Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ЯЗЫК OBJECT PASCAL И СИСТЕМА ПРОГРАММИРОВАНИЯ DELPHI

Учебное пособие

по курсу «Инструментарий систем программирования»
для студентов специальности 31 03 04 «Информатика»
дневной формы обучения

Минск 2003

УДК 681.3.06 (075)
ББК 32.973 я 73
В 68

Р е ц е н з е н т :
проректор Высшего государственного колледжа связи,
канд. физ.-мат. наук, доц. В. Н. Теслюк

Волосевич А.А.

В 68 Язык Object Pascal и система программирования Delphi: Учеб. пособие по курсу «Инструментарий систем программирования» для студентов специальности 31 03 04 «Информатика» дневной формы обучения / А.А. Волосевич. – Мн.: БГУИР, 2003. – 60 с.

ISBN 985-444-467-8.

Учебное пособие является дополнительным учебным материалом для студентов, изучающих программирование на языке Object Pascal с использованием системы Delphi. Рассматриваются синтаксические конструкции для реализации концепций объектно-ориентированного программирования. Даны примеры описания классов и построения собственных компонентов.

УДК 681.3.06 (075)
ББК 32.973 я 73

ISBN 985-444-467-8

Содержание

Введение

В настоящее время наиболее распространенными операционными системами для персональных компьютеров являются системы семейства Windows компании Microsoft. Для современного программиста совершенно необходимо овладеть навыками разработки программного обеспечения для достаточно сложной для программирования операционной системы. Одной из особенностей программирования для Windows является то, что при написании программ важная роль уделяется созданию и кодированию пользовательского интерфейса.

Для повышения производительности программистов и улучшения качества получаемых Windows-программ разработчиками программного обеспечения было предпринято несколько шагов. Первый шаг – отход от традиционного процедурного и модульного программирования в пользу объектно-ориентированного программирования и создание специализированных объектно-ориентированных библиотек. Второй шаг – использование интегрированных сред программирования, сочетающих редактор, компилятор, отладчик и объектно-ориентированную библиотеку. И, наконец, третий шаг – появление интегрированных сред, в которых разработка пользовательского интерфейса представляет собой визуальный процесс, автоматически сопровождаемый генерацией соответствующего программного кода. Подобные системы получили название *RAD-сред* (Rapid Application Development – быстрая разработка приложений). Delphi представляет собой одну из таких систем, выгодно сочетая простоту освоения и программную мощь.

Систему Delphi составляют:

1. Интегрированная среда разработки (IDE), включающая редактор кода, визуальный редактор интерфейса приложения, компилятор и отладчик.
2. Объектная библиотека VCL (в Delphi 6 добавлена библиотека CLX).
3. Компилятор командной строки.
4. Средства для работы с базами данных.
5. Файлы справки.
6. Файлы исходного кода объектной библиотеки VCL.

Система Delphi поставляется в трех вариантах:

а) основная версия (издание «Standard») рассчитана на новичков в Delphi и непрофессионалов;

б) второй уровень (издание «Professional») предназначен для профессиональных разработчиков и включает дополнительно к изданию «Standard» поддержку баз данных, расширенную поддержку программирования для Internet и некоторые дополнительные инструменты;

в) полное издание («Enterprise») ориентировано на разработчиков приложений масштаба предприятия.

1. Понятия «проект», «форма», «компонент»

При написании программ начального уровня для MS-DOS используется, как правило, один файл с исходным текстом программы. Для разработки даже простейших приложений Windows, как правило, необходимо несколько исходных файлов. Эти файлы логически объединены понятием проекта. *Проект* (Project) – набор файлов, необходимых для создания приложения.

Большинство приложений Windows выполняются в отдельных областях экрана, называемых *окнами*. Кроме того, приложения часто создают дополнительные окна в процессе работы. Специфика создания Windows-приложения в любой RAD-среде, в том числе и в Delphi, заключается в разбиении исходной задачи на две подзадачи: разработка интерфейсной части (как программа выглядит и как взаимодействует с пользователем) и разработка программной части (что программа делает). Окно на этапе разработки в IDE называется *формой* (Form).

Несмотря на разнообразие внешнего вида окон приложений, в них можно выделить некоторые стандартные элементы, характерные практически для любого окна (например, меню, кнопки, строка состояния, текстовые поля). На этапе разработки этим элементам соответствуют так называемые *компоненты* (Components). Компоненты подобны кирпичикам, из которых строится интерфейс приложения. Отметим, что кроме компонентов, предназначенных для построения интерфейса (так называемых *визуальных компонентов*), система Delphi предлагает большой набор *невизуальных компонентов*, которые на этапе выполнения приложения не видны, однако могут использоваться приложением для организации доступа к данным, отсчета временных промежутков, связи с Internet. Итак, компонент является частью формы, форма – это часть проекта. На этапе выполнения проекту соответствует приложение, форме – окно, а компоненту – один из стандартных элементов окна.

Рассмотрим подробнее состав проекта. Типичный проект, как правило, включает следующие файлы:

1) главный – текстовый файл с расширением `DPR`, содержащий главный программный блок; главный файл проекта подключает все остальные используемые файлы и содержит операторы для запуска приложения;

2) описания форм – текстовые файлы с расширением `DFM`, которые содержат описание внешнего вида формы и размещенных на ней компонентов, установленного на этапе проектирования; количество файлов описания форм равно количеству форм в проекте;

3) программные модули – текстовые файлы с расширением `PAS`, которые содержат исходный код на языке Object Pascal. Это может быть либо код программного описания формы, либо участок программы, выделенный в отдельный модуль; количество файлов программных модулей больше или равно количеству форм в проекте;

4) файлы ресурсов (в частности, файл ресурсов проекта) – двоичные файлы с расширением RES, содержат ресурсы Windows; файл ресурсов проекта включает пиктограмму приложения;

5) файл опций проекта – текстовый файл с расширением DOF, содержит настройки компилятора и проекта и используется, если эти настройки отличны от принятых по умолчанию.

Проект также может содержать графические файлы (расширение BMP, JPG), файлы оперативной подсказки (расширение HLP) и другие файлы.

Файлы программных модулей доступны для редактирования в IDE. Код программного описания форм создается Delphi автоматически, обычно редактируются в нем лишь участки, ответственные за обработку событий компонент. Главный файл проекта также генерируется автоматически. Необходимость редактирования его возникает крайне редко. Для просмотра этого файла можно использовать команду IDE `Project|View Source` . Изменить пиктограмму приложения возможно при помощи команды IDE `Project| Options| Application| Icon`.

2. Сравнение языков Object Pascal и Turbo Pascal

Язык Object Pascal является встроенным языком программирования системы Delphi и развитием языка Turbo Pascal, используя некоторые особенности диалекта этого языка. Поэтому Object Pascal будет рассмотрен в сравнении с версией Turbo Pascal 7.0.

Заметим, что для использования некоторых специфических особенностей Object Pascal предусмотрен так называемый *режим расширенного синтаксиса*. Для управления режимом используется директива компилятора `$X` (по умолчанию режим включен – `$X+`).

Алфавит Object Pascal совпадает с алфавитом Turbo Pascal. Остались прежними правила записи констант и идентификаторов. Набор ключевых слов и директив языка существенно расширен в сравнении с Turbo Pascal. Однако, так как в редакторе IDE все ключевые слова и стандартные директивы выделяются **полужирным** шрифтом, то запоминать их для того, чтобы не использовать в качестве идентификаторов программиста, нет необходимости.

В Object Pascal появилась новая форма записи комментариев. Теперь комментарием является также любой текст от двойной наклонной черты до конца строки. Старые способы записи комментариев сохранены. Новый способ не применим для записи директив компилятора:

```
A+B; // это комментарий
```

2.1. Типы данных

Набор типов данных претерпел изменения, связанные с переходом к 32-разрядной платформе программирования. Он также был дополнен некоторыми новыми типами, отсутствующими в Turbo Pascal.

Типы `Byte`, `Word`, `ShortInt`, `LongInt` по диапазону и формату соответствуют одноименным типам Turbo Pascal. Часто используемый тип `Integer` имеет формат 4 байта и представляет диапазон от `-2147483648` до `2147483647`. Появились новые целые типы: `SmallInt` (формат – 2 байта, диапазон от `–32768` до `32767`) и `Cardinal` (формат – 4 байта, диапазон `0...4294967295`). В Delphi 4 дополнительно введены типы `LongWord` (является синонимом `Cardinal`) и `Int64` (64-разрядное знаковое целое, до 18 значащих цифр).

Вещественные типы `Single`, `Double`, `Extended`, `Comp` аналогичны типам Turbo Pascal. Заметим, что операции с типами `Single`, `Double`, `Extended` и `Comp` поддерживаются аппаратно, на уровне микропроцессора. Учитывая популярность идентификатора `Real` среди Pascal-программистов, в Delphi 4 этот тип объявлен как синоним типа `Double`. Для обратной совместимости в Delphi 4 описан тип `Real48`, аналогичный по формату типу `Real` ранних версий Delphi и Turbo Pascal. Новый вещественный тип `Currency` рекомендован как тип для финансовых расчетов. Его формат – 8 байт, но он имеет всего четыре значащих цифры после запятой. Отметим, что внутренним представлением вещественных

констант в Object Pascal является тип Extended. Это может служить источником трудноуловимых ошибок, аналогичных приведенному примеру:

```
var A: Double;
...
A := 1.1;
if A = 1.1 then ...// в данном примере условие будет ложно!
```

Упражнение. Разберите, почему оператор if работает подобным образом. Почему оператор работает корректно, если присвоить переменной A значение 1.5 и сравнить его с константой 1.5 (сравните двоичные представления чисел 1.1 и 1.5)?

Для представления отдельных символов Object Pascal использует типы AnsiChar, WideChar и Char. Типы Char и AnsiChar являются синонимами. Они кодируют один из 256 символов таблицы ANSI, их формат – один байт. Тип WideChar занимает два байта и кодирует символы из кодовой таблицы Unicode.

Набор булевых типов данных был расширен для большей совместимости с программными библиотеками операционной системы. Кроме стандартного типа Boolean появились три новых типа: ByteBool, WordBool, LongBool. Их формат – соответственно один, два и четыре байта, нулевое значение интерпретируется как «ложь», любое другое – как «истина».

Для представления дат и времени в Object Pascal используется тип TDateTime. Фактически это синоним типа Double, т.е. вещественное число. Целая часть этого числа соответствует количеству дней, прошедших с 30 декабря 1899 года, дробная – части суток, прошедших с полуночи.

Object Pascal разделяет множество строковых типов на три вида: короткие строки, длинные и строки с завершающим нулем. *Короткие строки* объявляются при помощи ключевого слова string, за которым следует заключенное в квадратные скобки значение максимальной длины строки. Формат такого типа полностью совпадает с форматом аналогичного объявления в Turbo Pascal. Для объявления короткой строки длиной 255 символов возможно использование синонима ShortString.

Длинные строки объявляются при помощи ключевого слова string без указания максимальной длины строки. Максимальный размер длинной строки теоретически может составлять около 2 Гб. В переменной, объявленной как длинная строка, фактически хранится адрес начала строки в памяти. Символы длинной строки индексируются от 1 до N+1, где N – реальная длина строки. Доступ к отдельным символам организуется так же, как и для короткой строки, например St[2], St[2345], при этом символ с индексом N+1 всегда равен #0. Для получения значения длины строки следует использовать функцию Length, для изменения длины – процедуру SetLength. Для длинной строки в памяти хранятся количество символов в строке и счетчик ссылок на строку, которые являются 32-разрядными беззнаковыми целыми числами. Счетчик ссылок на строку позволяет экономно расходовать память при наличии разных переменных, содержащих одинаковые строки. Синонимом типа string является тип

AnsiString. Тип данных WideString – это длинная строка, состоящая из символов таблицы Unicode (код символа занимает два байта). Тип WideString не поддерживает счетчик ссылок на строку.

Для объявления *строки с завершающим нулем* (так называемый ANSIIZ формат) используется идентификатор PChar. Переменная типа PChar – это указатель на массив Char, который индексируется с нуля и заканчивается символом #0. Строки с завершающим нулем используются для совместимости с программами, написанными на языках C и C++. В режиме расширенного синтаксиса значения указателей PChar можно складывать и вычитать, массив символов с нижним индексом, равным нулю, и строковые константы совместимы с типом PChar.

Object Pascal поддерживает автоматическое приведение типов для длинных и коротких строк. Явное преобразование строк выполняется с помощью конструкций вида ShortString(S) и AnsiString(S).

В Object Pascal включена поддержка переменных *вариантного типа*. Такие переменные могут принимать значения различных типов – быть целым или вещественным числом, строкой, массивом, принимать значения булевых переменных. Идентификатором вариантного типа служит ключевое слово Variant. Для вариантных переменных поддерживается гибкое встроенное приведение типов. Следующий пример показывает это:

```
var V1, V2, V3, V4: Variant; //объявление переменных
. . .
V1 := 5; //целое число
V2 := 0.8; //вещественное число
V3 := '10.2'; //строка
V4 := V1+V2+V3; //целое число, равно 16
```

Будьте внимательны с автоматическим приведением типов. Обратимся к примеру:

```
var V1, V2, V3, V4: Variant; //объявление переменных
. . .
V1 := '100'; //строка
V2 := '10'; //строка
V3 := 1; //число
V4 := V1 + V2 + V3;
```

Чему будет равно V4? Эта переменная будет содержать число, и так как операция сложения левоассоциативна, то число это будет равно 10011.

Внутреннему представлению вариантной переменной соответствует структура VVarData. Ее подробное описание можно найти в интерактивной системе помощи. Отметим, что одно из полей этой структуры позволяет узнать текущий тип вариантной переменной.

Object Pascal содержит большое количество подпрограмм для работы с вариантными переменными. В качестве примера рассмотрим функцию VarArrayCreate, создающую массив произвольной размерности и произвольного типа и помещающую его в вариантную переменную:

```
var V: variant;
. . .
V := VarArrayCreate([1,10,5,7], varDouble);
```

Первый параметр функции VarArrayCreate – открытый массив целых, который задает размерность массива и границы каждой размерности (в нашем примере: две размерности, границы первой – от 1 до 10, второй – от 5 до 7). Второй параметр – тип значения элементов массива (в нашем примере – вещественные числа).

Операции с вариантными переменными выполняются сравнительно медленно, их присутствие в программе может понадобиться только для поддержки некоторых программных технологий (COM, ActiveX).

В Delphi 4 появилась возможность работать с *динамическими массивами*. Работу с такими массивами иллюстрирует следующий пример:

```
var A: array of Integer; {динамический массив объявляется без указания диапазона индексов}
. . .
   SetLength(A,10); {перед использованием надо задать длину массива в логических элементах при помощи процедуры SetLength}
   A[0] := 10; //индексы начинаются с нуля
   A[9] := -100;
   A := nil; //память, занимаемая массивом, освобождается
```

Отметим, что после первого использования процедуры SetLength все элементы созданного массива обнуляются. Если вызвать SetLength для уже созданного массива с целью увеличения его размера (SetLength(A,20)), то старые значения элементов в массиве сохранятся.

Имеется возможность создавать многомерные динамические массивы:

```
var B: array of array of integer;
. . .
  SetLength(B, 2);
  SetLength(B[0], 3);
  SetLength(B[1], 2);
  B[0,0] := 10;
  B[1,1] := -100;
```

Для массивов можно использовать следующие функции: Low(A) и High(A) – возвращают значения нижней и верхней границы индекса массива A, Length(A) – возвращает количество элементов массива. Для динамических

массивов функция Low всегда возвращает значение 0. Если для динамического массива не вызывалась процедура SetLength, то значение функции High будет равно -1. Функция Copy(A, Start, Count) возвращает динамический массив, являющийся частью динамического массива A. Так как переменные динамических массивов фактически являются указателями, то присваивание вида A := B (A, B – динамические массивы одного типа) приведет к тому, что обе переменные будут ссылаться на один участок памяти, и память для A не будет выделена. Внутри функции Copy создается динамический массив и возвращается ссылка на него, так что после оператора A := Copy(B, 0, 10) длина массива A будет равна 10, даже если она предварительно была установлена равной 100.

2.2. Переменные и константы

Типизированные константы при установке директивы компилятора $J+ становятся константами настоящими, т.е. их значение нельзя изменять. Начиная с Delphi 3, описание строковых констант можно размещать в секции resourcestring. Строковые константы, помещенные в этой секции, при использовании не отличаются от обычных, однако находятся в специальной части исполняемого файла, называемой *ресурсной строковой таблицей*. Это позволяет изменять такие константы в откомпилированных файлах при помощи специальных программ-редакторов ресурсов.

Все глобальные переменные в Object Pascal инициализируются нулевым значением (целые и вещественные числа получают значение 0, логические переменные – false, указатели – nil, строки инициализируются пустой строкой). Существует возможность задать начальное значение глобальной переменной при ее объявлении:

```
var X: Integer = 10;
```

Подчеркнем, что техника инициализации работает только для глобальных переменных, т.е. объявленных в основной программе или в модуле, но не в подпрограмме.

2.3. Процедуры и функции

В Object Pascal функции могут возвращать значения любого типа, за исключением файлового. При использовании расширенного синтаксиса возможен вызов функции как процедуры, т.е. без присвоения возвращаемого функцией значения.

При описании подпрограмм возможно задание специальных директив, регулирующих порядок передачи параметров. Директива register указывает на передачу параметров по возможности через регистры процессора. Этот способ является быстрым и используется в Object Pascal по умолчанию. Директива pascal задает традиционный способ передачи параметров через стек. Директива cdecl применяется при описании процедур, которые должны

быть совместимы с языком С. Для работы с системными библиотеками описываются процедуры с директивой `stdcall`. Директива `safecall` уведомляет компилятор о необходимости использования безопасного метода передачи параметров процедуре с дополнительным контролем. При использовании директив `register` и `pascal` параметры передаются, начиная с самого левого параметра. При использовании остальных директив передача параметров начинается с правого параметра:

```
function F(X: Integer): Byte; stdcall; //пример директивы
```

При использовании расширенного синтаксиса в функциях можно использовать внутреннюю переменную `Result`. Тип этой переменной совпадает с типом возвращаемого значения функции, описывать ее в блоке `var` функции не надо. Эта переменная может находиться как слева, так и справа от знака присваивания. Для задания функции возвращаемого значения достаточно присвоить это значение переменной `Result`:

```
function Sum(A, B: Integer): Integer;
begin
  Result := A + B; //эквивалентно Sum := A + B
end;
```

В качестве параметров в подпрограммы можно передавать так называемые *открытые массивы*, т.е. массивы, размер которых заранее неизвестен и уточняется непосредственно при вызове подпрограммы:

```
function Sum(const A: array of Integer): Integer;
var i: Integer;
begin
  Result := 0; //переменную Result необходимо инициализировать
  for i := Low(A) to High(A) do //для открытого массива Low(A)=0
    Result := Result + A[i]
end;
```

В приведенном примере в функцию `Sum` можно передать любой массив с элементами `Integer` или даже сконструировать такой массив непосредственно во время передачи:

```
k := Sum([0, 5, 6, X*3]);
```

В качестве фактического параметра вместо массива можно передать единственное значение с типом открытого массива. В подпрограмме это значение трактуется как массив из одного элемента:

```
k := Sum(5);
```

Особый случай представляет объявление `array of const`. При наличии такого объявления в процедуру можно передать открытый массив из элементов разных типов. Каждый элемент такого массива преобразуется в запись типа

TVarRec. У этой записи есть поле VType: Byte, при помощи которого можно узнать конкретный тип элемента:

```
function Example(const A: array of const): Integer;
. . .
  case A[i].VType of
    vtInteger:. . .
    vtExtended:. . .
```

В Delphi 4 появилась возможность объявлять подпрограммы с одинаковыми именами (*перегружать подпрограммы*). Для этого необходимо указать директиву overload после описания параметров процедуры или функции. Чтобы компилятор смог различить вызовы таких подпрограмм, у них должны отличаться либо количество, либо тип параметров:

```
procedure Ov1(X: Double); overload;
. . .
procedure Ov1(X: Integer); overload;
```

Если в программе появится вызов Ov1(1.6), это будет означать вызов первой процедуры, вызов Ov1(4) будет означать вызов второй процедуры.

Начиная с Delphi 4, стало возможным описывать в объявлении подпрограмм *параметры по умолчанию*:

```
procedure Proc(X: Integer; Y: Real = 3.14);
```

Для приведенного выше объявления возможны вызовы Proc(7) и Proc(7, 4.6). В первом случае X=7, Y=3.14, во втором случае X=7, Y=4.6. Если объявление содержит параметры по умолчанию, то они должны быть последними в списке параметров.

2.4. Модули

В Object Pascal модуль может содержать секции initialization и finalization. Секцию initialization составляют операторы, которые выполняются при подключении модуля. Секция finalization, которая может присутствовать только при наличии секции initialization, включает операторы, выполняемые при завершении программы при выгрузке модуля.

В составе Delphi имеется большое количество модулей. Основная часть их составляет объектную библиотеку VCL. Модуль Math содержит несколько десятков процедур и функций для математических и статистических расчетов. Модуль SysUtils включает процедуры и функции для работы с файлами, строками, переменными типа TDateTime, конвертирования типов и многие другие. Этот модуль подключается к большинству проектов автоматически.

Рассмотрим некоторые процедуры и функции модуля SysUtils. Процедура Beep генерирует звуковой сигнал. Функция Now возвращает текущую дату и время в формате TDateTime. Функции Date и Time возвращают соответственно

текущую дату и время. Функции DateTimeToStr, DateToStr, TimeToStr конвертируют значения переменной типа TDateTime в строку. Обратное преобразование выполняют функции StrToDateTime, StrToDate, StrToTime. Для получения номера дня недели по дате служит функция DayOfWeek, при этом значение 1 соответствует воскресенью. Получить величину в формате TDateTime по значению года, месяца и дня позволяет функция EncodeDate. EncodeTime кодирует время по переданным значениям часов, минут, секунд и миллисекунд. Обратные преобразования выполняют функции DecodeDate и DecodeTime. Функция IsLeapYear возвращает значение «истина», если переданное ей целое значение соответствует високосному году.

Функции IntToStr, FloatToStr, FloatToStrF, CurrToStr, StrToInt, StrToFloat используются для перевода числовых значений в строковые и наоборот.

Функция Format используется для так называемого *форматного вывода* (подобного применяемому в языке C). Аргументы функции – строка, содержащая *спецификаторы формата*, и открытый массив значений. Функция возвращает переданную строку, в которой спецификаторы формата заменены значениями из массива:

```
S:=Format('Задано %d и %d', [5,2]); //S='Задано 5 и 2'
```

Спецификаторы формата имеют вид

```
%[<индекс>:][-][<ширина>][.<точность>]<тип>,
```

где <индекс> – номер аргумента в массиве, к которому относится спецификатор; [-] – идентификатор выравнивания влево; <ширина> – ширина поля; <точность> – спецификатор точности; <тип> – обязательное поле, ниже перечислены его возможные значения:

d – десятичное число;

e – формат вида -d.dd..e+ddd; если задана <точность>, то она обозначает общее количество цифр;

f – формат вида -d.ddd; <точность> – количество знаков после запятой;

g – совпадает с e или f, в зависимости от того, что дает меньше цифр;

n – совпадает с f, дополнительно содержит разделители тысяч;

m – монетарный формат (добавляет $ или p.);

p – формат для указателей (XXXX:YYYY);

s – строка, если есть <точность>, то это количество символов строки;

x – шестнадцатеричный формат целых чисел.

3. Объектно-ориентированное программирование. Базовые понятия

Традиционное процедурное программирование подразумевает разработку программы методом последовательного уточнения и детализации. Исходная задача разбивается на несколько связанных подзадач, затем каждая подзадача делится на более мелкие, и так далее, вплоть до неких элементарных задач. *Объектно-ориентированное программирование* (ООП) предполагает представление исходной задачи как процесса взаимодействия некоторых вполне самостоятельных программных единиц, называемых *объектами*. Так, в среде Delphi при разработке приложения представляет объект, любой компонент также является объектом. Работа программиста в этом случае заключается в наиболее полном описании набора объектов задачи и кодировании их связей. Подобный стиль программирования позволяет существенно сократить время разработки сложных программ и программных комплексов, особенно при использовании готовых наборов объектов, так называемых *объектных библиотек*.

Наиболее распространенными языками, реализующими ООП, являются C++, Java, Turbo Pascal (начиная с версии 5.5), Object Pascal. При программировании на этих языках используются такие объектные библиотеки, как Turbo Vision, OWL, MFC, VCL и другие.

ООП основывается на трех принципах:

1. *Инкапсуляция* – объединение в одном программном типе, называемом *класс*, как данных, так и подпрограмм для их обработки. Данные класса хранятся в *полях класса*, подпрограммы для их обработки называются *методами класса*.

2. *Наследование* – возможность создавать новые классы на основе существующих. Новый класс называется *классом-потомком* (или *дочерним классом, производным классом*), старый – *классом-предком* (или *родительским классом, базовым классом*). Наследование предполагает, что либо дочерний класс сохраняет поля и методы родительского класса, либо при необходимости изменяет их или дополняет новыми. При помощи наследования можно строить так называемое *дерево классов* (или *иерархию классов*), последовательно уточняя описание класса и переходя от общих абстрактных понятий к частным.

3. *Полиморфизм* – особый вид замены методов при наследовании, при котором программный код, работавший с методами родительского класса, пригоден для работы с измененными методами дочернего класса.

Рассмотрим, как реализуется ООП в Object Pascal. Опишем некий класс для работы с домашними животными:

```
type TPet = class
       fName: string; // f – традиционный префикс для имени поля
       fAge: Integer;
       procedure SetAge(aAge: Integer);
       function GetName: string;
     end;
```

Дадим необходимые комментарии. Описание класса размещается в секции описания типов. Описать класс можно в глобальной секции типов программы, в секции типов в интерфейсной части модуля или в разделе реализации модуля, но не в подпрограмме. Для описания класса используется ключевое слово `class`. Вначале описываются поля класса. Полями класса `TPet` являются `fName` и `fAge`. После описания всех полей следует указание методов класса. Методы `TPet` – это процедура `SetAge` и функция `GetName`. Класс содержит только заголовки методов, реализация методов описывается отдельно. Если класс описан в модуле, реализация методов должна находиться в том же модуле в секции `implementation`. Для указания того, что процедура или функция является реализацией метода некоего класса, используют синтаксис `<имя класса>.<имя метода>`. При реализации методов класса полностью доступны поля и другие методы этого класса без указания каких–либо дополнительных спецификаторов:

```
procedure TPet.SetAge (aAge: Integer);
begin
  if aAge > 0 then fAge := aAge else fAge := 0
end;
function TPet.GetName: string;
begin
  Result := 'My name is ' + fName
end;
```

После того, как класс описан, можно объявить переменную класса, называемую *экземпляром класса* или *объектом*:

```
var Pet: TPet;
```

Перед непосредственной работой с объектом он должен быть особым образом инициализирован. Следующий код выполняет это:

```
Pet := TPet.Create;
```

После инициализации для работы с полями и методами объекта используется синтаксис вида `<имя объекта>.<имя поля или метода>`. Это напоминает работу с записью:

```
Pet.fName := 'Tuzik';
Pet.SetAge(5);
writeln(Pet.GetName+' I am ', Pet.fAge);
```

Хотя прямая работа с полями объекта возможна, в ООП принято для доступа к полям использовать методы, подобные `TPet.SetAge` и `TPet.GetName`. Считается, что поля представляют внутреннюю закрытую часть класса, которую создатель класса вправе изменять. Заголовки методов доступа к полям (*интерфейс класса*) при этом должны оставаться неизменными. Object Pascal

позволяет принудительно ограничить доступ к полям класса при помощи специальных директив.

В методе `TPet.SetAge` содержится обращение к полю `fAge`. Каждый объект определенного класса содержит свой независимый набор полей. Метод `SetAge` можно вызвать для нескольких объектов. Возникает вопрос: как метод определяет, с полем какого объекта он работает?

```
var P1, P2: TPet;
. . .
P1.SetAge(10); // SetAge работает с P1.fAge
P2.SetAge(4); // SetAge работает с P2.fAge
```

Для выявления конкретного объекта, с которым происходит работа, любому методу передается *скрытый параметр* `Self`, который указывает на объект, вызывающий метод. Тип параметра `Self` совпадает с типом класса. На уровне компилятора описание метода `SetAge` и работу с ним можно представить следующим образом:

```
procedure TPet.SetAge (aAge: Integer; Self: TPet);
begin
  if aAge > 0 then Self.fAge := aAge else Self.fAge := 0
end;
. . .
P1.SetAge(10, P1);
P2.SetAge(4, P2);
```

Итак, `Self` указывает на текущий объект, с которым происходит работа. Практически всегда в явном использовании `Self` нет необходимости. Одно из исключений – использование одинаковых идентификаторов для полей класса и параметров метода. Предположим, что класс `TPet` содержит поле `aAge`, а не `fAge`. Тогда корректная реализация метода `TPet.SetAge` должна выглядеть так:

```
procedure TPet.SetAge (aAge: Integer);
begin  //aAge – параметр метода, Self.aAge – поле объекта
  if aAge > 0 then Self.aAge := aAge else Self.aAge := 0
end;
```

4. Наследование классов. Перекрытие методов

Предположим, возникла необходимость в описании класса, представляющего домашних животных, способных издавать звуки. Подобный класс будет похож на класс `TPet`, но должен содержать некий дополнительный метод, ответственный за воспроизведение звука. При помощи наследования можно описать новый класс на основе уже существующего:

```
type TSpeakingPet = class(TPet)
       procedure Speak;
     end;
. . .
procedure TSpeakingPet.Speak;
begin
  Beep;
end;
```

Запись `TSpeakingPet = class(TPet)` означает, что класс `TSpeakingPet` является наследником класса `TPet`. Как наследник, `TSpeakingPet` содержит все поля и методы `TPet` и, кроме этого, добавляет собственный метод `Speak`.

В Object Pascal все классы имеют одного общего предка. Таким предком является класс `TObject`. Объявления `TPet = class` и `TPet = class(TObject)` являются эквивалентными. Мы использовали один из методов класса `TObject` для инициализации объекта `Pet` (метод `Create` в классе `TPet` не описан).

Объекты классов-потомков совместимы по присваиванию с объектами классов-предков. При этом действует следующее правило: объекту родительского класса можно присвоить объект дочернего класса, но не наоборот:

```
var MyPet: TPet; SPet: TSpeakingPet;
. . .
MyPet := SPet; // допустимо
SPet := MyPet; // ошибка компиляции
```

Обосновывается вышеуказанное правило следующим образом: так как дочерний класс может добавлять к родительскому новые поля, то при присваивании объекту дочернего класса объекта родительского класса некоторые из полей, возможно, не будут инициализированы. Это является недопустимым.

Новый дочерний класс может не только дополнять родительский полями и методами, но и замещать методы родительского класса своими. Объявление в дочернем классе метода с тем же именем, что и в родительском классе называется *перекрытием*. При этом сигнатуры методов, т.е. количество и тип параметров, могут отличаться.

Для иллюстрации перекрытия объявим два новых класса, порожденных от `TSpeakingPet`:

```
type TDog = class(TSpeakingPet)
```

```
      procedure Speak; // перекрытие метода TSpeakingPet.Speak
    end;
    TCat = class(TSpeakingPet)
      procedure Speak; // перекрытие метода TSpeakingPet.Speak
    end;
. . .
procedure TDog.Speak;
begin
  writeln('Wuf');
end;
procedure TCat.Speak;
begin
  writeln('Meau');
end;
```

Для вызова перекрытых методов ближайшего класса-предка применяется конструкция `inherited <имя метода класса-предка>`. Если имя и параметры вызываемого у предка метода совпадают с именем и параметрами вызывающего метода, достаточно записать `inherited`:

```
procedure TCat.Speak;
begin
  inherited; // вызов TSpeakingPet.Speak – звуковой сигнал
  writeln('Meau');
end;
```

5. Конструкторы и деструкторы

Object Pascal использует так называемую *ссылочную объектную модель*. Это означает, что все объекты размещаются в памяти динамически, объектные переменные фактически являются указателями на данные объекта в динамической памяти и имеют одинаковый размер (4 байта). Однако для доступа к объектам не используется разыменователь «^» (мы записываем `Pet.fName`, а не `Pet^.fName`, хотя подразумевается именно второе). Для размещения объектов в динамической памяти служит особый вид методов, называемых *конструкторами*. Для уничтожения объектов также предназначены особые методы – *деструкторы*.

Чтобы объявить метод как конструктор, используется ключевое слово `constructor`. Оно записывается вместо слова `procedure` в объявлении класса и при реализации метода (конструктор не может быть функцией). Компилятор автоматически добавляет к телу конструктора некий код, выделяющий для объекта участок в динамической памяти. Так как конструктор необходимо выполнить перед первым использованием объекта, то в тело конструктора обычно помещают операторы инициализации объекта, к примеру задание начальных значений для полей. В иерархии классов работа конструктора класса-потомка, как правило, начинается с вызова конструктора класса-предка для корректной инициализации полей предка. Отметим, что Object Pascal допускает существование в классе нескольких конструкторов. Традиционное имя для конструктора – `Create`. Класс `TObject` содержит конструктор `Create`, который обнуляет поля создаваемого объекта.

Добавим конструктор в класс `TPet`:

```
type TPet = class
       . . .
       constructor Create;
     end;
. . .
constructor TPet.Create;
begin
  fAge := 1;
  fName := 'Pet';
end;
```

Теперь оператор `Pet := TPet.Create` означает вызов не унаследованного конструктора `TObject.Create`, а собственного конструктора. Обратите внимание на синтаксис вызова конструктора: `<объектная переменная> := <имя класса>.<имя конструктора>`. Как метод, конструктор можно вызывать и в виде `<имя объекта>.<имя конструктора>` (`Pet.Create`). В первом случае конструктор работает как функция, которая выделяет память, размещает в ней объект соответствующего класса, выполняет тело конструктора и возвращает

ссылку на выделенную область. Вызов, подобный `Pet.Create`, означает просто выполнение тела конструктора (реинициализацию полей). Его можно применять только для тех объектов, которые уже размещены в памяти.

Для иллюстрации работы унаследованных конструкторов добавим конструктор в класс `TSpeakingPet`:

```
type TSpeakingPet = class(TPet)
       constructor Create; // перекрытие конструктора
     end;
. . .
constructor TSpeakingPet.Create;
begin
  inherited; // TPet.Create: fAge = 1, fName = 'Pet'
  fName := 'Speaking Pet';
end;
```

Для объявления деструкторов используется ключевое слово `destructor`. Задача деструктора – освободить память, которую занимал объект. Тело деструктора – подходящее место для финальных действий с объектом (освобождение динамической памяти, закрытие файлов и тому подобное). Традиционное имя для деструктора – `Desrtoy`. Даже если ваш класс не содержит собственного деструктора, вызывайте унаследованный деструктор `TObject.Destroy` при окончании работы с объектом, чтобы не допускать утечек памяти:

```
Pet := TPet.Create; // создание объекта
Pet.fName := 'Tuzik'; // работа с объектом
Pet.Destroy; // уничтожение объекта. Вызов TObject.Destroy
```

Деструктор можно вызвать только у объекта, а не у класса. Попытка вызвать деструктор у неинициализированного объекта может привести к исключительной ситуации. Рекомендуется вместо прямого обращения к деструктору использовать метод `TObject.Free`, который вызывает деструктор `Destroy`, если объект инициализирован (указатель на объект отличен от `nil`). Следует иметь ввиду, что метод `Free` вызывает не просто деструктор, а именно `Destroy`. Для корректной работы метода `Free` в унаследованных классах деструктор должен называться `Destroy` (и быть виртуальным).

6. Реализация полиморфизма

Рассмотрим пример. Предположим, что имеется иерархия классов TPet→TSpeakingPet→(TDog, TCat). В программе необходимо создать массив из объектов класса TSpeakingPet или его наследников и для всех элементов массива вызвать метод Speak:

```
var Zoo: array [1..3] of TSpeakingPet;
. . .
  Zoo[1] := TSpeakingPet.Create;
  Zoo[2] := TDog.Create; // корректно по правилам присваивания
  Zoo[3] := TCat.Create; // для объектов
  for i := 1 to 3 do
    Zoo[i].Speak;
```

Ожидается, что результатом работы последнего цикла будут звуковой сигнал и вывод строк ‘Wuf‘ и ‘Meau’. Однако в результате будет получена лишь последовательность из трех звуковых сигналов.

Еще один пример. Опишем процедуру для обработки объектов класса TCryingPet:

```
procedure Proc(A: TSpeakingPet);
begin
. . .
  A.Speak;
end;
```

Хотя в Proc возможно передать не только объекты класса TCyingPet, но и любого дочернего класса, вызываться в теле процедуры будет метод TSpeakingPet.Speak.

Подобные расхождения возникают из-за того, что нами до сих пор использовался единственный вид методов – *статический*. Адрес статического метода известен на этапе компиляции. Он определяется классом объекта и не зависит от того, с каким классом будет связан объект на этапе выполнения. Нам необходимо, чтобы адрес метода вычислялся непосредственно в период выполнения по тому типу, который в этот момент имеет объект. Для этого необходимо применить *виртуальные методы*.

Работа с виртуальными методами происходит по следующей схеме. Каждый объект наряду со значениями своих полей хранит указатель на специальную *таблицу виртуальных методов* (virual method table – VMT). Таблица виртуальных методов индивидуальна и единственна для каждого класса. В ней хранятся адреса всех виртуальных методов класса (как собственных, так и унаследованных). Связь между объектом и VMT класса осуществляется во время начальной инициализации объекта – первого вызова конструктора. Виртуальные методы идентифицируются по константе-смещению в VMT. Во время

выполнения программы из экземпляра объекта извлекается указатель на VMT и, используя константу-смещение, вычисляется адрес необходимого метода.

Для объявления виртуального метода используется директива `virtual`. Методы, объявленные с директивой `dynamic` (*динамические методы*), функционально эквивалентны виртуальным. Иная внутренняя организация хранения адресов динамических методов позволяет сократить накладные расходы памяти, но несколько замедляет работу с ними.

Исходный метод, объявленный как виртуальный (динамический) в классе–предке, перекрывается в классе–потомке методом с тем же именем и параметрами, что и исходный, и помечается директивой `override`. Если хотя бы одно из этих требований не соблюдено, связь между виртуальными методами в иерархии классов теряется.

Отредактируем классы `TSpeakingPet`, `TDog` и `TCat`, сделав метод `Speak` виртуальным:

```
type TSpeakingPet = class(TPet)
       procedure Speak; virtual;
     end;
     TDog = class(TSpeakingPet)
       procedure Speak; override;
     end;
     TCat = class(TSpeakingPet)
       procedure Speak; override;
     end;
```

Если внести подобные изменения в описания классов, то вышеприведенные примеры заработают корректно. При реализации методов директивы `virtual`, `dynamic`, `override` не указываются.

Типичным примером полиморфизма в Delphi является объявление метода `Repaint` в классе `TControl`, предке всех визуальных компонентов. Благодаря тому, что этот метод объявлен как виртуальный, форма может прорисовать любой размещенный на ней визуальный компонент.

Рассмотрим еще один пример, который требует применения виртуальных методов. Пусть имеется некий абстрактный класс `TFigure` для представления геометрических фигур. Его наследники `TCircle` и `TSquare` представляют собой соответственно круг и квадрат. Для рисования себя все три класса используют метод `Draw`, для стирания – метод `Hide`. Эти методы различны у всех классов. Кроме этого, `TFigure` содержит метод `MoveTo` для перемещения фигуры. Он состоит из вызова `Hide`, изменения координат фигуры и вызова `Draw`. Логика работы `MoveTo` полностью сохраняется для классов `TCircle` и `TSquare`. Однако если сделать методы `Draw` и `Hide` статическими, `MoveTo` придется дословно переписать в классах–потомках, так как `TFigure.MoveTo` содержит вызовы `TFigure.Draw` и `TFigure.Hide`.

В этом примере для методов TFigure.Draw и TFigure.Hide реализация не важна. Главное то, *что* они делают, а не *как*. Можно оформить реализацию этих методов в виде пустых процедур. Лучшее решение состоит в указании директивы abstract, объявляющей метод *абстрактным*, не нуждающимся в реализации:

```
type TFigure = class
       . . .
       procedure Draw; virtual; abstract;
     end;
```

Директива abstract применяется только для виртуальных и динамических методов в корневых классах объектных иерархий. Попытка создать экземпляр класса, содержащего абстрактные методы, вызовет предупреждение, а обращение к абстрактному методу сгенерирует исключительную ситуацию.

Одним из ключевых понятий системы Windows является *сообщение*. Сообщение – это сигнал приложению от операционной системы о том, что произошло некоторое событие (к примеру, нажатие пользователем клавиши, перемещение мыши, запуск приложения). Каждый класс сообщений идентифицируется уникальной числовой константой. Object Pascal позволяет создавать методы для обработки сообщений. Схема внутренней реализации таких методов напоминает динамические методы. Метод обработки сообщений это всегда процедура с единственным var-параметром. За заголовком метода следуют директива message и номер сообщения Windows, которое этот метод будет обрабатывать. Этот номер обычно представляется мнемонической константой:

```
type TCls = class
       . . .
       procedure WmUser(var M: TMessage); message wm_User;
     end;
```

Модуль Messages содержит константы для сообщений, а также большое количество типов, идентифицирующих конкретные сообщения.

7. Свойства

Развитие концепций ООП привело к появлению понятия *свойства* (property). Работа в программе со свойствами объекта происходит так же, как с полями объекта. Разница между полем и свойством заключается в следующем: обращение к свойству компилятор транслирует в вызов метода или обращение к полю, следовательно, при работе со свойствами могут выполняться некоторые действия (к примеру, при обращении к свойству «цвет» некоего объект, он перерисовывается указанным цветом); в отличие от полей свойства не занимают места в памяти.

Добавим свойства в класс `TPet`:

```
type TPet = class
      . . .
      property Age: Integer read fAge write SetAge;
      property Name: string read GetName write fName;
    end;
```

Для объявления свойства используется ключевое слово `property`. Далее следует имя свойства и указывается его тип. После директивы `read` указывается имя поля или метода для чтения свойства. Аналогично, после директивы `write` указывается имя поля или метода для записи свойства. Считается, что свойство записывается, когда ему присваивается некое значение, в противном случае свойство читается. Тип полей, используемых после `read` и `write`, должен совпадать с типом свойства. Метод, используемый для чтения простого свойства, должен быть функцией без параметров, тип возвращаемого значения которой совпадает с типом свойства. Метод для записи – процедура с одним параметром, имеющим тип свойства. Динамические методы и методы для обработки сообщений не могут использоваться для чтения и записи свойств. Принято имена методов чтения свойств начинать с префикса `Get`, имена методов записи свойств – с префикса `Set`. Объявление свойства может следовать только после объявления полей и методов, которые используются для чтения и записи. Работу со свойствами демонстрирует следующий фрагмент программы:

```
var Pet: TPet;
. . .
Pet.Name := 'Fish'; // оттранслируется в Pet.fName := 'Fish'
Pet.Age := 1;       // Pet.SetAge(1);
writeln(Pet.Age, Pet.Name); // writeln(Pet.fAge, Pet.GetName)
```

То, что свойства не являются полями, накладывает определенные ограничения на их использование. Свойства нельзя передавать в качестве `var`-параметров в подпрограммы, к ним нельзя применить операцию взятия адреса.

Если опустить директиву `read`, можно получить свойство только для записи. Если опустить `write`, получим свойство только для чтения. Однако какая-то из директив должна присутствовать.

Использование свойств является «хорошим тоном» ООП. Как было сказано выше, следует избегать прямой работы с полями класса. В пользу использования свойств говорит тот факт, что в объектной библиотеке VCL весь доступ к полям классов и компонентов организован через свойства.

Если класс имеет несколько «почти одинаковых» свойств, то для их обработки можно использовать одинаковые методы:

```
type TExample = class
       . . .
       property Prop1: Byte index 0 read Get write Set;
       property Prop2: Byte index 1 read Get write Set;
       property Prop3: Byte index 2 read Get write Set;
     end;
```

Свойства `Prop1`, `Prop2`, `Prop3` называются *индексированными*. Об этом говорит использование директивы `index` с последующей уникальной целой константой. Для чтения всех индексированных свойств применяется единственный метод-функция, содержащий один целый параметр. Запись всех индексированных свойств осуществляется также одним методом-процедурой, первый параметр которого – целое число. Применение полей для чтения и записи индексированных свойств невозможно:

```
var Ex: TExample;
. . .
Ex.Prop1 := 100; // Ex.Set(0, 100)
k := Ex.Prop3; // k := Ex.Get(2)
```

Object Pascal позволяет объявлять так называемые *свойства-массивы*, представляющие индексированное множество свойств:

```
type TEx = class
       . . .
       property Props[i: Integer]: Byte read Get write Set;
     end ;
```

Как и для индексированных свойств, для доступа к свойствам-массивам возможно использование только методов. Первый параметр этих методов должен совпадать по типу с типом индекса в квадратных скобках. Ниже приведен пример работы со свойством `Props` и указано, во что оттранслируются обращения к нему:

```
for i := 1 to 5 do
  Ex.Props[i] := 0; //транслируются в Ex.Set(i, 0);
```

Тип индекса свойства-массива не ограничен диапазоном (к примеру, индекс может быть строкового или вещественного типа). Допускается работа только с элементами свойства-массива, а не со всем свойством целиком.

Свойства-массивы могут быть многомерными. В этом случае количество необходимых параметров у методов чтения и записи увеличивается на соответствующее число.

Если при описании свойства-массива добавить в конце директиву `default`, то такое- свойство становится *основным свойством класса*, что упрощает работу с ним:

```
property Props[i: Integer]: Byte read Get write Set; default;
. . .
for i := 1 to 5 do
  Ex[i] := 0; //вместо Ex.Props[i] := 0
```

Только свойство-массив может быть основным свойством класса. У класса может быть только одно основное свойство.

8. Видимость атрибутов класса

Под *атрибутом класса* далее понимается поле, метод или свойство класса.

Как было указано выше, идеология ООП подразумевает, что не все данные и методы класса равноправны при использовании: одна их часть представляет внутреннюю организацию класса и должна быть скрыта от пользователя, другая служит интерфейсом класса и является общедоступной.

Чтобы ограничить доступ к атрибуту класса, предусмотрено использование специальных директив ограничения видимости (мы пока их не использовали). Они применяются не отдельно к каждому атрибуту, а разделяют объявление класса на секции:

1. `Private`. Все атрибуты из данной секции доступны для использования только в том модуле, который содержит объявление класса. В этой секции обычно размещаются поля и методы, описывающие внутренние особенности реализации класса.

2. `Protected`. Атрибуты этой секции могут использоваться вне пределов модуля с объявлением класса, но только потомками класса. В данную секцию обычно помещают виртуальные методы чтения и записи свойств.

3. `Public`. Атрибуты из секции `public` не имеет ограничений на использование. В эту секцию помещают методы и свойства, составляющие интерфейс класса.

4. `Published`. Атрибуты секции `published` (публикуемые), как и атрибуты из секции `public`, не имеют ограничений на использование. Компилятор генерирует для них специальную информацию, позволяющую работать с ними в IDE и Инспекторе объектов. В этой секции объявлено большинство свойств компонент. Тип поля из секции `published` может быть только классом, причем поддерживающим RTTI. RTTI (Run-time type information, *информация о типе периода времени выполнения*) – дополнительная информация об атрибутах класса, хранимая в классе и доступная во время выполнения программы. Класс с поддержкой RTTI либо откомпилирован с директивой `$M+`, либо является потомком класса, откомпилированного с этой директивой. Публикуемые свойства не могут быть свойствами-массивами. Тип публикуемого свойства может быть порядковым, множеством в пределах `Integer`, классом с поддержкой RTTI, `Variant`, вещественным типом, любым строковым типом. Для таких свойств обязательно наличие директивы как `read`, так и `write`.

Если класс не содержит директив ограничения видимости, считается, что все его атрибуты имеют видимость `public`. Для классов с поддержкой RTTI (к примеру, класс `TPersistent`) по умолчанию принимается директива `published`. Порядок директив в описании класса произволен, допускается даже повторение директив. В отличие от C++, внутри модуля, содержащего описание класса, все директивы ограничения видимости не действуют.

Для публикуемых свойств возможно указание спецификаторов default и stored, которые управляют процессом сохранения значения публикуемых свойств в DFM файле.

Директива default служит для указания для свойства значения по умолчанию, которое должен устанавливать конструктор класса. Именно начальное значение отображается в Инспекторе объектов, и только если значение свойства отлично от принятого по умолчанию, оно сохранится в DFM-файле:

```
type TC = class
        . . .
        published
        property P: Integer read fP write fP default 5;
      end;
constructor TC.Create;
begin
  P := 5;
end;
```

Директива stored используется для указания того, будет ли свойство запоминаться в DFM-файле. В объявлении свойства за директивой stored должны следовать либо имя логического поля данного класса, либо логическая константа, либо имя метода-функции класса, не имеющего параметров и возвращающего логическое значение. Свойство запоминается в DFM-файле, если значение, следующее за stored, равно true:

```
type TC = class
        . . .
        function NeedStore: Boolean;
        published
        property P: Integer
                  read fP write fP stored NeedStore default 5;
      end;
```

Для свойств существует возможность изменения ограничения видимости в классе-потомке с protected на public или published. Для этого достаточно указать ключевое слово property и имя свойства в новой секции, не указывая тип свойства и методы или поля для чтения и записи. При необходимости можно указать новое значение свойства по умолчанию:

```
type TArrow = class(TGraphicControl)
        . . .
        published
        property Height default 20;
      end;
```

9. Работа с классами. Обработчики событий

Рассмотрим специальный тип данных, называемый *метакласс*. Этот тип введен для манипулирования классами, а не отдельными объектами.

Описание метакласса и объявление соответствующей переменной выглядят следующим образом:

```
type TPetClass = class of TPet;
var PetRef: TPetClass;
```

На объявление метакласса указывает конструкция `class of <имя класса>`. Что можно присваивать переменной `PetRef`? Ей присваивается либо имя класса, объявленного после `class of`, либо его дочерних классов:

```
PetRef := TPet;
. . .
PetRef := TDog;
```

Значением переменной типа метакласс фактически является указатель на VMT соответствующего класса. Так, в приведенном примере `PetRef` вначале содержит указатель на VMT класса `TPet`, затем указатель на VMT класса `TDog`.

В Object Pascal имеется предопределенный тип `TClass`:

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

Так как `TObject` является предком для всех объектов, то переменной типа `TClass` можно присваивать значение любого класса.

Ценность метаклассов состоит в возможности создавать программы, которые манипулируют классами, не существующими в момент написания программы. Примером этого служит работа с формой. Когда мы конструируем форму, мы создаем новый класс, порожденный от `TForm`. Класс `TForm` содержит виртуальный конструктор. Метод `TApplication.CreateForm` ответственен за создание и визуализацию формы. Его заголовок имеет вид `TApplication.CreateForm(T: class of TForm; var F: TForm)`. В теле метода создается объект `F` класса `T`. Благодаря ссылке на класс и виртуальному конструктору метод `CreateForm` может создать объект не только класса `TForm`, но и любого дочернего класса.

9.1. Классовые методы

В предыдущих примерах для вызова метода класса использовался экземпляр класса. Object Pascal позволяет описать в классе такие методы, для вызова которых нет необходимости создавать объект. Эти методы называются *классовыми* (class method). Для их объявления достаточно указать слово `class` непосредственно перед `procedure` или `function`:

```
type TSomeClass = class
```

```
      class procedure Hello;
    end;
class procedure TSomeClass.Hello;
begin
  writeln('Hello. This is class method')
end;
```

Так как в момент вызова классового метода экземпляр класса может и не существовать, то в теле такого метода запрещено обращение к полям объекта и обычным методам. Однако классовый метод может вызывать другие классовые методы и конструктор класса.

Для вызова классового метода возможно применение как синтаксиса <имя класса>.<имя классового метода>, так и <имя объекта>.<имя классового метода>. Для вызова классового метода можно применять метакласс:

```
type TSomeClassRef = class of TSomeClass;
var C: TSomeClass;
    R: TSomeClassRef;
. . .
  TSomeClass.Hello; // все вызовы корректны
  C := TSomeClass.Create; //дают одинаковый результат
  C.Hello;
  R := TSomeClass;
  R.Hello;
```

Указатель `Self` для классового метода содержит адрес VMT класса. Разрешено объявление виртуальных классовых методов. Для классовых методов действуют обычные правила видимости атрибутов класса.

9.2. Указатели на методы. Обработчики событий

В Object Pascal, как и в Turbo Pascal, есть процедурные типы:

```
type Proc = procedure (k: Integer);
var P: Proc;
```

Однако, хотя метод `TPet.SetAge` имеет такую же сигнатуру, как и при объявленнии типа `Proc`, присваивание `P := Pet.SetAge`, где `Pet` – экземпляр `TPet`, является недопустимым. Причина этого в том, что методу всегда передается неявный параметр `Self`. Процедурные типы для методов называются *указателями на метод* (method pointer) и объявляются следующим образом:

```
type Proc = procedure (k: Integer) of object;
var P: Proc;
```

Конструкция `of object` указывает на объявление указателя на метод. Теперь возможно присваивание вида `P := Pet.SetAge`. Переменная типа указа-

тель на метод занимает 8 байт и содержит адрес метода и ссылку на объект (значение Self для конкретного объекта).

Указатели на метод используются для создания и манипулирования *событиями*. Любое событие объекта представляет собой свойство типа указатель на метод. IDE Delphi автоматически помещает все такие свойства класса, объявленные в секции published, на закладку Events Инспектора объектов. Тип простейших событий с единственным параметром Sender описан следующим образом:

```
type TNotifiEvent = procedure(Sender: TObject) of object;
```

В качестве примера рассмотрим класс-компонент TEdit. Он обладает событием OnClick. Для реализации работы с ним класс содержит поле FOnClick и свойство OnClick:

```
type TEdit = class(TCustomEdit)
       . . .
       FOnClick: TNotifiEvent;
       property OnClick: TNotifiEvent
                         read FOnClick write FOnClick;
     end;
```

Если в процессе разработки мы размещаем на форме компонент Edit1: TEdit и пишем для него обработчик события OnClick, то фактически мы пишем новый метод класса формы TForm1.Edit1Click. При создании формы в начале выполнения приложения метод формы связывается со свойством компонента следующим образом (это происходит без нашего участия):

```
Edit1.OnClick := Form1.Edit1Click; // Form1 – объект TForm1
```

При щелчке на компоненте Edit1 системой вызывается метод TEdit.Click. В нем производятся проверка наличия обработчика события OnClick и вызов его, если он существует:

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

Функция Assigned возвращает значение «истина», если ее аргумент не равен nil.

Подобный подход, когда все методы обработки событий компонент сосредоточены в одном классе (TForm1), называется *делегированием*. Делегирование позволяет избежать порождения от существующих классов-компонентов многочисленных потомков, содержащих дополнительные методы для обработки событий.

10. Приведение и контроль типов объектов

Любой класс содержит некую дополнительную информацию, размещаемую в памяти непосредственно перед VMT. Эти данные позволяют во время выполнения программы контролировать (type checking) и приводить (type casting) объектные типы.

Для контроля типов используется оператор `is`. Выражение `<объект> is <класс>` возвращает «истину», если `<объект>` принадлежит классу `<класс>` или потомкам этого класса (`if Obj1 is TPet then . . .`).

Для приведения типов используется оператор `as` в следующей форме:

```
(Obj1 as TPet).SetAge(10);
```

Допустима традиционная конструкция `TPet(Obj1).SetAge(10)`, однако вариант с `as` является более безопасным. В случае неудачи он генерирует обрабатываемую исключительную ситуацию, а жесткое приведение типов `TPet(Obj1)` может привести к краху приложения.

Приведение и контроль типов можно использовать для написания универсальных обработчиков событий. Предположим, на форме имеются компоненты `TEdit` и `TButton`. Необходимо при щелчке на компоненте `TEdit` установить свойство `Text='Hi'`, при щелчке на `TButton` – свойство `Caption = 'Hello'`:

```
procedure TForm1.Edit1Click(Sender: TObject);
begin
  if Sender is TEdit then
    TEdit (Sender).Text := 'Hi'
  else if Sender is TButton  then
        TButton(Sender).Caption := 'Hello'
end;
```

Приведенный обработчик события `TEdit.Click` решает эту задачу. Этот обработчик нужно назначить событию `TButton.Click`, выбрав его в выпадающем списке на закладке `Events` в Инспекторе объектов. В примере использовался тот факт, что любой обработчик события имеет параметр `Sender`, указывающий на объект, который инициировал событие.

11. Исключительные ситуации

Термин *исключительная ситуация* (exception) обозначает любую ошибку или ошибочное условие, возникающие в процессе выполнения программы. Примерами исключительных ситуаций являются деление на 0, запись в файл, открытый только для чтения, ошибки при преобразовании типов. Традиционный подход для контроля и обработки подобных ошибок заключается в использовании условных операторов. Однако они загромождают программу, делают ее менее ясной.

Object Pascal предлагает новый подход: при исключительной ситуации порождается объект особого класса, при помощи этого объекта информация об исключительной ситуации передаётся специальному обработчику, написанному программистом.

Каждому типу исключительной ситуации соответствует некий класс. Эти классы образуют иерархию, корнем которой является класс `Exception`. Приведем некоторые классы и соответствующие им исключительные ситуации:

`EAbort` – «безмолвная» исключительная ситуация, без сообщений; генерируется вызовом процедуры `Abort`;

`EDivByZero` – деление целого числа на 0;

`EInOutError` – ошибка при доступе к файлу;

`EOutOfMemory` – нехватка памяти;

`EMathError` – общая ошибка при работе с вещественными числами;

`EZeroDivide` – деление вещественного числа на 0;

`EConvertError` – ошибка при преобразовании типов.

Полный список классов содержит справочная система Delphi. Основные классы исключительных ситуаций описаны в модуле `SysUtils`.

Для генерации объекта исключительной ситуации применяется предложение `raise`. Вот как выглядит создание объекта в случае нехватки памяти:

```
raise EOutOfMemory.Create('Мало памяти');
```

Обратите внимание: вызывается конструктор класса, соответствующего исключительной ситуации. Для гибкой обработки исключительных ситуаций класс `Exception` предлагает несколько конструкторов. Мы использовали простейший, параметр которого устанавливает свойство `Message` класса `Exception`.

Обработчики исключительных ситуаций реализуются при помощи *защищенного блока* `try-except`. Синтаксис блока следующий:

```
try
  //операторы, которые могут вызвать исключительную ситуацию
except
  //операторы обработки исключительных ситуаций
end;
```

При нормальном ходе программы выполняются операторы, размещенные между try и except, а затем – размещенные после end. Если какой–либо из операторов между try и except вызвал исключительную ситуацию, управление сразу передается в часть except-end. Предполагается, что в ней сосредоточена обработка исключительной ситуации. После обработки исключительной ситуации выполняются операторы, размещенные за end. Конструкции try-except могут быть вложенными.

Для распознавания исключительной ситуации в части except-end служит последовательность блоков обработки вида on <класс исключительной ситуаций> do <оператор>. К примеру:

```
try
  //работа с вещественными числами
except
  on EZeroDivide do . . . ; //обрабатываем ошибку деления на 0
  on EMathError do . . . ; //обработка других матем. ошибок
end;
```

Если для класса исключительной ситуации существует блок обработки, выполняется часть блока после do и управление передается за блок try-except. Порядок блоков обработки имеет значение: вначале проводится распознавание частных исключительных ситуаций, затем – более общих (как и в приведенном примере).

Если необходимо, чтобы нераспознанные исключительные ситуации были как-то обработаны, можно поместить операторы их обработки в секцию except-end после слова else:

```
except
  on EZeroDivide do . . . ; //обрабатываем ошибку деления на 0
  on EMathError do . . . ; //обработка других матем. ошибок
  else //обработка нематематических ошибок
end;
```

Приведем пример функции, генерирующей исключительную ситуацию:

```
function StrToPersent(const S: string): Integer;
begin
  Result := StrToInt(S);
  if (Result < 0) or (Result > 100) then
    raise EConvertError.Create(S + 'is not a valid value')
end;
```

Функция StrToPersent может генерировать исключительную ситуацию класса EConvertError. В следующей функции она распознается и обрабатывается:

```
function IncPercent(const S: string): string;
begin
```

```
    try
      Result := IntToStr(StrToPercent(S) + 1);
    except
      on EConvertError do Result := '0';
    end
  end;
```

Если мы хотим возложить часть обработки исключительной ситуации на некий внешний блок, то исключительную ситуацию можно сгенерировать повторно, используя `raise`:

```
on EZeroDivide do
begin
 . . . //частично обработали мы
  raise //пусть обрабатывает внешний обработчик
end;
```

Если необходимо получить доступ к объекту, описывающему исключительную ситуацию, то используется следующая форма блока обработки: `on <идентификатор объекта>: <класс исключительной ситуации> do <оператор>;`

```
on E: EConvertError do
begin
  Result := '0';
  ShowMessage(E.Message)
end;
```

Освобождение созданного объекта исключительной ситуации происходит автоматически. Переменную, используемую в блоке `on` (в примере это `E`), нигде описывать не надо.

Для корректного освобождения ресурсов приложения часто используется блок `try-finally`. Синтаксис блока следующий:

```
try
  // операторы, которые могут вызвать исключительную ситуацию
finally
  // эти операторы выполняются всегда
end;
```

При возникновении исключительной ситуации в части `try-finally` управление передается на часть `finally-end`. При нормальной работе выполняются все операторы между `try` и `end`.

Если возникшая исключительная ситуация программистом не обработана, вызывается обработчик события `OnExcept` объекта `Application`. Если этот обработчик не установлен, выводится стандартное окно с текстовым сообщением об ошибке.

12. Иерархия классов VCL

Объектная библиотека VCL насчитывает около 200 классов, составляющих сложное дерево иерархии. Основными классами, «стволом дерева», являются следующие:

```
TObject → TPersistent → TComponent → TControl → (TGraphic-
Control, TWinControl)
```

Класс `TObject` является общим предком всех классов. Он содержит конструктор `Create`, виртуальный деструктор `Destroy`, метод `Free` для освобождения объекта, методы для работы с сообщениями системы `Dispatch` и `DefaultHandler` и большое число классовых методов для работы с информацией о классе. Эти методы позволяют узнать предок класса (`ClassParent`), имя класса (`ClassName`), адрес метода в секции `published` по имени метода (`MethodAddress`), размер объекта данного класса (`InstanceSize`) и многое другое. Виртуальные методы `TObject` хранятся по отрицательным смещениям таблицы VMT, так что «первым» методом в VMT будет первый виртуальный метод производного класса. Классы `Exeption`, `TList`, `TPrinter` порождены непосредственно от класса `TObject`.

Особенностью класса `TPersistent` является то, что он и его потомки содержат методы для сохранения и чтения данных своих полей. `TPersistent` содержит также методы для копирования данных из полей одного объекта в поля другого. Так как объекты являются всего лишь указателями на данные в памяти, то код

```
var A, B: TSomeClass;
. . .
A := B;
```

приведет к тому, что и `A` и `B` будут ссылаться на один участок памяти, а данные, связанные с объектом `A`, будут потеряны. Чтобы этого не произошло, следует использовать метод `Assign`, который позволяет вызвавшему объекту присваивать данные, связанные с другим объектом, и метод `AssignTo`, копирующий данные вызвавшего объекта в другой объект. На уровне `TPersistent` данные методы объявлены как абстрактные, классы–потомки должны переопределить их. Класс `TPersistent` скомпилирован с директивой `$M+`, а значит, у него и у всех его потомков атрибуты по умолчанию имеют видимость `published`. Классы `TString`, `TCanvas`, `TGraphic`, `TPicture` порождены непосредственно от `TPersistent`.

Рассмотрим подробнее класс `TStrings`. Он относится к *классам общего назначения*. Задачей классов общего назначения является реализация типичных структур данных – списков, потоков, коллекций. Такие классы можно использовать самостоятельно, но задумывались они как вспомогательные элементы

крупных классов VCL. Большая часть классов общего назначения описана в модуле `Classes`.

Класс `TStrings` служит для представления списка строк. Этот класс является абстрактным, непосредственно оперировать его методами нельзя. Многие компоненты имеют свойства, тип которых представляет класс, порожденный от `TStrings`. Класс `TStringList` является наследником `TStrings` и допускает прямое использование.

Перечислим основные свойства класса `TStrings`:

`Count: Integer` – число элементов в списке;

`Strings[Index: Integer]: string` – массив строк, основное свойство, элементы нумеруются с нуля;

`Objects[Index: Integer]: TObject` – массив объектов, это свойство позволяет ассоциировать с каждой хранимой строкой некий объект;

`Text: string` – позволяет интерпретировать массив строк как одну большую строку с разделителями `#13#10`.

Основные методы класса `TStrings` приведены ниже:

`Add` – добавляет строку в массив и возвращает её позицию;

`AddObject(const S: string; AObject: TObject): Integer` – добавляет в массив строку `S` и ассоциирует с ней объект `AObject;`

`AddStrings(St: TStrings)` – добавляет в массив список строк;

`Append` – аналог `Add`, но является процедурой;

`Clear` – удаляет все элементы списка;

`Delete(Index: Integer)` – удаляет строку с индексом `Index` и ассоциированный с ней объект. `Delete` и `Clear` не вызывают у удаляемых объектов деструктор;

`Equals(String: TStrings): Boolean` – сравнивает два списка;

`Exchange(Ind1. Ind2: Integer)` – меняет два элемента списка местами;

`GetText: PChar` – возвращает весь список в виде строки с завершающим нулем;

`IndexOf(const S: string ): Integer` – позиция строки `S` в списке или –1;

`Insert(Index: Ineger; const S: string )` – вставка в список строки в заданную позицию;

`LoadFromFile(const Name: string)`, `SaveToFile(const Name: string)` – загрузка или сохранение списка в файле;

`Move(CurIndex, NewIndex: Integer)` – перемещение элемента списка;

`SetText(Text: PChar)` – загрузка списка из одной строки с завершающим нулем.

Класс `TStringList` добавляет к `TStrings` несколько новых свойств, методов и событий.

Свойства класса `TStringList`:

`Duplicates` – определяет, разрешено ли использование дубликатов в списке. Возможные значения: `DupIgnore` – дубликаты игнорируются, `DupAccept` –

дубликаты разрешаются, DupError – попытка добавить дубликат вызывает ошибку;

Sorted: Boolean – при установке в true строки автоматически сортируются по алфавиту.

Методы класса TStringList:

Find(const S: string; var Index: Integer): Boolean – ищет строку S, если она найдена, возвращает true и индекс строки;

Sort – сортирует строки списка по алфавиту.

События класса TStringList:

OnChange – генерируется после внесения изменений в список;

OnChanging – генерируется перед внесением изменений в список.

Класс TComponent происходит от TPersistent и является предком визуальных и невизуальных компонентов. Объекты класса TComponent называются *компонентами*. На уровне TComponent реализована поддержка возможности редактирования свойств компонентов в Инспекторе объектов. Компоненты также поддерживают возможность владения другими компонентами, при этом компонент-владелец становится ответственным за уничтожение принадлежащих ему компонентов. Некоторые из свойств класса TComponent перечислены ниже:

Owner – компонент, который владеет данным;

ComponentState – текущее состояние компонента;

ComponentCount – количество компонентов, принадлежащих данному;

Components – массив принадлежащих компонентов;

ComponentIndex – текущая позиция в этом массиве;

Name – имя компонента, используется как имя поля в классе формы;

Tag – число типа Integer.

Класс TComponent определяет виртуальный конструктор Create с одним параметром – именем компонента-владельца создаваемого компонента.

Методы TComponent позволяют найти компонент по имени в массиве Components (FindControl), получить список всех компонент, владельцем которых является данный, и так далее. Класс TScreen и большинство классов невизуальных компонентов порождены непосредственно от TComponent.

Объект Screen класса TScreen инкапсулирует информацию об экране. Этот объект создается автоматически в начале работы приложения. Большинство свойств класса TScreen доступно только для чтения. Некоторые из свойств перечислены ниже:

ActiveForm – форма приложения, имеющая фокус ввода;

ActiveControl – элемент управления, имеющий фокус;

Cursor – вид курсора на экране;

Cursors – массив доступных курсоров;

FormCount – число форм, отображённых на экране;

Forms – массив видимых форм программы;

Fonts: TStrings – имена системных шрифтов для вывода текста на экран;

Width, Height – ширина и высота экрана в пикселях.

Наследником класса TComponent является класс TApplication. Задача этого класса – инкапсулирование поведения приложения в целом. В любом приложении существует автоматически создаваемый объект Application этого класса.

Основные свойства класса TApplication приведены ниже:

MainForm: TForm – главная форма приложения;

ExeName – строка, содержит пути и имя приложения;

Title – название приложения на панели задач;

HelpFile – имя файла-справки для приложения;

Icon – значок приложения на панели задач;

Hint – текст всплывающей подсказки;

HintColor – цвет окна всплывающей подсказки;

Handle – дескриптор приложения.

Некоторые события класса Tapplication:

OnActivate – приложение стало активным;

OnDeactivate – приложение теряет фокус при переключении приложений;

OnException – возникла исключительная ситуация, которая не обработалась;

OnIdle – приложение простаивает.

Приведем некоторые из методов класса Tapplication:

Terminate – завершает работу приложения;

ProcessMessages – обрабатывает системные сообщения Windows; используется в теле длинных циклов для устранения эффекта «замораживания»;

ShowException(E: Exception) – выводит окно с описанием исключительной ситуации E;

Minimize – свёртывает все окна приложения;

Restore – развёртывает все окна приложения;

MessageBox – показывает простейшее диалоговое окно с текстом;

CreateForm – создает и отображает форму.

От класса TComponent происходит класс TControl – общий предок визуальных компонентов. Класс TControl содержит свойства позиционирования компонента (Top, Left, Widht, Height, Align), свойства клиентской области (ClientRect, ClientWidth, ClientHeight), свойства внешнего вида (Visible, Enabled, Font, Color), строковые свойства (Caption, Text, Hint), свойства мыши (Cursor, DragCursor, DragMode). На уровне класса TControl появляются обработчики событий мыши (OnClick, OnDblClick, OnMouseDown, OnMouseMove, OnMouseUp) и событий перетаскивания Drag&Drop (OnDragOver, OnDragDrop, OnEndDrag). Каждый компонент класса TControl имеет родительский компонент класса TWinControl, указываемый в свойстве Parent и отвечающий за отображение компонента. Если свойство Parent не установлено, компонент не отображается.

Класс TWinControl является предком для компонентов, инкапсулирующих стандартные элементы управления, которые могут получать фокус (поля ввода,

кнопки, списки и другие). Компоненты класса TWinControl могут быть родительскими (Parent) для других визуальных компонентов. Основные свойства класса предназначены для управления внешним видом компонента и фокусом ввода:

Handle – дескриптор, целое число для идентификации объекта системой;

Ctrl3D – определяет, выглядит ли компонент объемным;

HelpContext – раздел справочной системы для оперативной подсказки;

Controls – список элементов управления, для которых компонент является родительским;

TabStop, TabOrder – управляют перемещениями по компонентам на форме при нажатии клавиши Tab.

Класс TWinControl содержит методы управлением фокусом (Focused, CanFocus, SetFocus), методы позиционирования и выравнивания, методы создания и уничтожения окна (CreateParams, CreateWnd, DestroyWnd). Метод ControlAtPos возвращает дочерний компонент для указанной точки, метод FindNextControl позволяет найти компонент, получающий фокус после данного компонента.

Компоненты класса TWinControl поддерживают обработку событий клавиатуры (OnKeyDown, OnKeyUp, OnKeyPress) и событий управления фокусом (OnEnter, OnExit).

Класс TGraphicControl используется для визуальных компонентов, которые отображаются (рисуются) системой Delphi, а не Windows. Это позволяет экономно расходовать системные ресурсы. Потомки класса TGraphicControl (к примеру, TLabel) не могут получать фокус. Класс TGraphicControl содержит свойство Canvas – «холст» для рисования компонента и виртуальный метод Paint, содержащий операторы для рисования компонента и переопределяемый потомками класса.

В качестве примера работы с описанными классами рассмотрим, как создать компонент во время выполнения программы:

```
procedure TForm1.Button1Click(Sender: TObject);
var B: TButton;
begin
  B := TButton.Create(Self); //владелец новой кнопки – форма
  B.Top := 20; //кнопка невидима, настроим ее внешний вид
  B.Left := 40;
  B.Parent := Self; //сейчас кнопка отобразилась на форме
end;
```

13. Создание и использование DLL

Библиотека динамической компоновки (dynamic link lybrary, DLL) – выполняемый модуль Windows, код и ресурсы которого могут использоваться приложениями и другими DLL. Файлы библиотек динамической компоновки обычно имеют расширение `DLL`. Применение DLL оправдано, если приложение разрабатывается на нескольких языках программирования, программа содержит национальные ресурсы (к примеру, строки сообщений), существует несколько приложений, использующих сходный набор функций.

13.1. Структура DLL

Структура DLL схожа со структурой обычной программы. Она начинается с ключевого слова `library`, за которым следует имя библиотеки:

```
library MyDLL;
```

Все подпрограммы, которые будут вызываться из других приложений, объявляются в DLL с директивой `stdcall`. Вызов подпрограмм, объявленных без директивы `stdcall`, может быть затруднен в приложениях, написанных не в Delphi:

```
procedure MyProc(A: Integer; B: Char);stdcall;
```

Подпрограммы, экспортируемые из DLL, должны быть перечислены в специальном разделе `exports` исходного кода:

```
exports MyProc, MyFunct;
```

Таких разделов может быть несколько и в любом месте исходного кода DLL, главное, чтобы экспортируемые подпрограммы были описаны перед упоминанием в разделе `exports`.

Каждая экспортируемая из DLL подпрограмма идентифицируется для распознавания двумя ключами: *числовым индексом* и *именем экспорта*. По умолчанию имя экспорта совпадает с именем подпрограммы (большие и малые буквы в имени экспорта различаются), а индекс – с порядковым номером подпрограммы в разделе `exports`. Однако такой порядок можно обойти, явно указав имя экспорта и (или) числовой индекс:

```
exports
  MyProc1,
  MyProc2 name 'MyDinProc',
  MyProc3 index 3, //целое число от 1 до 2,147,483,647
  MyProc4 index 4 name 'MyPr@1'; //в имени допустимы спецсимволы
```

Будьте внимательны при явном указании индекса – его уникальность компилятором не отслеживается.

Рассмотрим пример DLL. Для получения некой «заготовки» DLL в IDE можно использовать следующую последовательность команд: File|New…|DLL. Далее наполняем эту «заготовку» содержимым:

```
library MathLib;
uses SysUtils; //модуль Classes нами не используется
function Max(X,Y: Integer): Integer;stdcall;
begin
  if X < Y then Result := Y else Result := X
end;
function Min(X,Y: Integer): Integer;stdcall;
begin
  if X < Y then Result := X else Result := Y
end;
exports Min, Max;
begin
end.
```

После компиляции этого кода (Ctrl+F9) в рабочем каталоге получим файл Mathlib.dll.

Между begin..end в коде DLL можно поместить операторы инициализации. Они выполняются при загрузке библиотеки DLL в память.

13.2. Импорт подпрограмм из DLL

Различают следующие виды импорта подпрограмм DLL – статический и динамический.

При статическом импорте DLL загружается в память в момент старта приложения и находится там до окончания его работы. При этом в исходном коде приложения размещены явные ссылки на подпрограммы DLL. Следующие примеры демонстрируют три возможных вида этих ссылок:

1. По имени подпрограммы в исходном коде DLL:

```
procedure MyProc1;stdcall;external 'MYLIB.DLL'.
```

2. По имени экспорта:

```
procedure MyProc2;stdcall;external 'MYLIB.DLL' name
'MyDinProc2'.
```

3. По числовому индексу:

```
procedure MyProc3;stdcall;external 'MYLIB.DLL' index 3.
```

Рекомендуется использовать импорт по имени экспорта, он более нагляден, хотя и работает немного медленнее, чем импорт по числовому индексу.

При динамическом импорте подпрограмм DLL загружается в память и выгружается самим приложением по мере надобности. Динамическая загрузка происходит следующим образом. Вначале вызывается функция `LoadLibrary (LibFileName: PChar): HModule`, где `LibFileName` – имя файла DLL. Если указано имя файла без маршрута, то DLL ищется в каталоге, из которого запущено приложение, затем – в текущем каталоге, системном каталоге (`Windows\System32`), каталоге `Windows`, каталогах, указанных в системной переменной `PATH`. Функция возвращает системный дескриптор DLL (число) или код ошибки от 0 до 32:

```
var LibHandle: THandle; //встроенный тип для дескрипторов
begin
. . .
  LibHandle := LoadLibrary('MathLib.dll');
  if LibHandle < 32 then . . .//обработка ошибки
. . .
end;
```

После загрузки DLL для получения адресов подпрограмм используется функция `GetProcAddress(Module: HModule; ProcName: PChar): Pointer`, где `ProcName` – имя подпрограммы. Если мы хотим произвести поиск подпрограмм по индексу, то два младшие байта указателя `PChar` должны содержать индекс, а два старшие – ноль. Если подпрограмма не найдена, при использовании имени `GetProcAddress` возвращает `nil`, при использовании индекса значение `GetProcAddress` может быть отлично от `nil`.

При динамическом импорте DLL в вызывающей программе необходимо объявить соответствующие процедурные типы. Приведем пример динамического импорта:

```
type TMin = functuion (X, Y: Integer): Integer;stdcall;
. . .
var Min: TMin
. . .
  @Min := GetProcAddress(LibHandle, 'Min');
  A := Min(B,C);
```

После завершения работы с библиотекой она освобождается вызовом функции `FreeLibrary(LibModule: HModule): BOOL`, где `LibModule` – дескриптор DLL. Функции `LoadLibrary`, `GetProcAddress`, `FreeLibrary` объявлены в модуле `Windows`.

Достоинствами динамического импорта DLL являются эффективное использование ресурсов памяти и возможность контролировать и обрабатывать ситуации, когда необходимой подпрограммы в DLL не оказалось.

13.3. Ресурсы в DLL

DLL может содержать не только подпрограммы, но и ресурсы, в частности, описание одной или нескольких форм. Для помещения формы в DLL во время разработки DLL в IDE необходимо выполнить команду `File|New form`. Затем новая форма настраивается необходимым образом (как правило, форма представляет диалоговое окно). Далее в исходный файл DLL добавляется процедура, которая показывает эту форму:

```
procedure ShowTheForm(AOwner: Tcomponent);stdcall;
begin
  Form1 := TForm1.Create(AOwner);
  Form1.ShowModal;
  Form1.Free
end;
. . .
exports ShowTheForm;
```

Эта процедура используется в основной программе:

```
procedure    ShowTheForm(AOwner    :TComponent);stdcall;external
'MYDLL.DLL';
. . .
ShowTheForm(Form1);
```

13. 4. Некоторые замечания

Если DLL экспортирует подпрограммы, использующие в качестве параметров значения типа `Variant`, длинные строки и динамические массивы, эта DLL и использующее ее приложение должны подключать модуль `ShareMem` (причем первым в списке `uses`) и иметь доступ к библиотекам `DELPHIMM.DLL` или `BORLANDMM.DLL` (поставляются в комплекте с Delphi).

Если в подпрограмме DLL возникла исключительная ситуация, то соответствующий объект передается в вызывающее приложение. Обработка исключительной ситуации возможна, если DLL использует модуль `SysUtils`.

Если в DLL объявлены свои глобальные переменные и константы, то доступ к ним из вызывающей программы возможен только через процедурный интерфейс.

Для главной программы, использующей DLL, обычно создается дополнительный интерфейсный модуль. В секции `interface` модуля указываются требуемые подпрограммы с директивой `stdcall`, а в секции `implementation` перечисляются с директивой `external` экспортируемые из DLL процедуры.

14. Разработка пользовательских компонентов

Одной из ключевых особенностей Delphi, хорошо иллюстрирующей преимущества объектно-ориентированного программирования, является возможность создания и использования новых пользовательских компонентов. При этом благодаря наследованию процесс создания компонента никогда не начинается «с нуля», а благодаря полиморфизму компонент органически встраивается в IDE и библиотеку VCL.

Создание компонента – невизуальный процесс. Он включает следующие стадии:

1. Написание программного кода компонента.
2. Создание значка компонента для *Палитры компонентов* и файла справки компонента (необязательный этап).
3. Инсталляция компонента.

Процесс создания компонента в различных версиях Delphi может иметь небольшие отличия. Следующее описание опирается на версию Delphi 3.

14.1. Компонент TNewLabel

Проиллюстрируем этапы создания компонента на примере `TNewLabel` – надписи, которая имеет отдельное свойство для изменения цвета шрифта.

Для получения модуля с «заготовкой» программного кода будущего компонента служит команда IDE `File|New|Component`. Заполним в появившемся диалоге следующие поля: `Ancestor Type` – `TLabel`, `Class Name` – `TNewLabel`, `Pallete Page` – `Samples`, `Unit File Name` – `'c:\work\newlabel'`.

Дадим необходимые комментарии. `Ancestor Type` – класс предка. Если новый компонент дополняет возможности существующего, указывается класс существующего компонента (как в нашем случае). Если компонент по содержанию радикально новый, то указываем: `TComponent` – для невизуальных компонентов; `TCustomControl` – для компонентов, которые должны получать фокус; `TGraphicControl` – для рисуемых компонентов; `TCustomXXX` (`TCustomLabel`) – если компонент дополняет возможности существующего, но желательно уменьшить количество свойств и событий, видимых в Инспекторе объектов. `Class Name` – имя класса нового компонента. `Pallete Page` – страница Палитры компонентов для размещения нового компонента. Можно указать как существующую, так и новую страницу, которая будет автоматически создана. `Unit File Name` – имя файла с программным кодом компонента.

Модуль-заготовка компонента, создаваемый IDE, содержит каркас описания класса компонента и процедуру `RegisterComponents` для регистрации компонента в IDE. Наша задача – наполнить этот модуль содержимым.

Наш компонент должен содержать новое свойство, ответственное за цвет надписи. Назовем это свойство `LabelColor`. Его тип, очевидно, `TColor`. Так как оно должно быть видимо в Инспекторе объектов, объявим его в секции

published. Для установки и чтения свойства используем методы SetLabelColor и GetLabelColor, описанные в секции private:

```
unit NewLabel;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls;
type
  TNewLabel = class(TLabel)
  private
    procedure SetLabelColor(AValue: TColor);
    function GetLabelColor: TColor;
  published
    property LabelColor: TColor
                         read GetLabelColor write SetLabelColor;
  end;
procedure Register;
implementation
procedure TNewLabel.SetLabelColor;
begin
  if AValue <> LabelColor then Font.Color := AValue;
end;
procedure TNewLabel.GetLabelColor;
begin
  Result := Font.Color;
end;
procedure Register;
begin
  RegisterComponents('Samples', [TNewLabel]);
end;
end.
```

Обратите внимание на синтаксис реализации методов и на разнесение методов по разделам видимости.

Разобраться с атрибутами классов компонентов, а также с разделами их видимости помогает *Браузер объектов*. Вызвать его можно после компиляции проекта, используя команду View|Browser. Браузер показывает дерево классов в левой части, правая часть содержит сгруппированные по разделам видимости атрибуты текущего класса. Для изучения компонентов используется также исходный код библиотеки VCL, поставляемый с редакцией Delphi Enterprise.

14.2. Пакеты

Процесс инсталляции компонента связан с понятием пакета. *Пакет* (package) – это расширенный вариант DLL, который может использоваться как программой, так и средой разработки Delphi. Фактически пакет – это несколько скомпилированных модулей (*.DCU) и, возможно, некоторые ресурсы, специальным образом объединенные. Файлы пакетов имеют расширение BPL.

Исходный файл пакета имеет расширение DPK и является обычным текстовым файлом. Рассмотрим его структуру на примере:

```
package Sample;
{$R 'COMP.DCR'} //подключение ресурсов (значок компонента)
{$DESCRIPTION 'Sample Components'} //описание пакета
requires
  vcl50;
contains
  Comp in 'Comp.pas';
end.
```

Директива package обозначает начало исходного файла пакета, директива requires – начало списка пакетов, требуемых для компиляции исходного. Директива contains обозначает начало списка имен модулей, из которых состоит пакет. Имена модулей разделяются запятыми, каждый модуль представлен либо именем, либо именем модуля и именем файла в виде строки.

Как правило, редактирование исходного текста пакета вручную не производится. Для этого применяется Менеджер пакетов из IDE Delphi.

При помощи директивы компилятора {$DesignOnly On}, помещенной в исходный файл пакета, он может быть помечен как *пакет разработки* (design time packages). Такой пакет нельзя присоединить к приложению, а можно использовать только при разработке приложения в IDE. По умолчанию директива выключена. При помощи директивы компилятора {$RunOnly On} пакет помечается как *пакет времени выполнения* (runtime packages). Такой пакет нельзя загрузить в IDE. Директивы не являются взаимоисключающими. По умолчанию установлено {$DesignOnly Off} и {$RunOnly Off} (пакет можно использовать и при разработке приложения и при выполнении).

Управление пакетами можно осуществить, вызвав диалоговую панель Packages (Project|Options|Packages или Component|Install Packages). Панель содержит две части. В первой находится список используемых IDE пакетов дизайна. Мы можем временно отключить использование пакета, а значит, и входящих в него компонент, убрать пакет из IDE (кнопка Remove) или добавить новый скомпилированный пакет с компонентами (кнопка Add). Состав компонентов, входящих в пакет, можно изучить при помощи кнопки Components.

Вторая часть диалоговой панели управляет генерацией кода с подключением пакетов времени выполнения. Установка флага `Build with runtime packages` позволяет сгенерировать приложение, которое не включает в себя код используемых компонентов и классов, а берет его во время выполнения из пакетов, перечисленных в диалоговом окне строкой ниже. Такое приложение будет малым по размеру (простейшее – около 12 Кбайт), но для своей работы оно будет требовать указанные в строке диалога пакеты. Главный пакет, содержащий основные компоненты, называется в Delphi 3 `VCL30.BPL` и имеет размер около 3,5 Мбайт.

Вернемся к нашему компоненту и завершим его установку. Для этого требуются создание DCU–файла для модуля компонента, включение этого файла в существующий пакет или создание нового пакета, компиляция и инсталляция полученного пакета. Совокупность этих действий выполняется при вызове команды `Component|Install Component`. После выполнения команды в появившемся диалоговом окне можно выбрать инсталляцию в новый или существующий пакет. Выберем инсталляцию в новый пакет (`Into new package`), укажем имя модуля (`Unit file name`), имя пакета (`Package file name`, должно отличаться от имени модуля), описание пакета (`Package description`, заполнять не обязательно). После закрытия окна произойдут немедленная компиляция и установка пакета. На Палитре компонентов появится новый компонент.

Более сложный путь предполагает создание пакета вручную. Для этого выполняется команда `File|New|Package`. Появившееся окно содержит кнопки для добавления или удаления модулей в пакет, компиляции пакета, установки пакета и компонент на палитру.

Если компонент активно редактируется в процессе работы, желательно тестировать его, не прибегая к инсталляции. Пусть компонент хранится в файле `c:\work\NewLabel.pas`. Создадим маленький тестовый проект, содержащий форму и кнопку, при нажатии на которую на форме появляется компонент. Для этого достаточно в секции `uses` модуля формы подключить `NewLabel.pas`, а в обработчике нажатия кнопки создать и отобразить компонент:

```
procedure TForm1.Button1Click(Sender: TObject);
var L: TNewLabel;
begin
 L := TNewLabel.Create(Self); //создаем компонент
 L.Caption := 'Hi-Hi'; //настраиваем его
 L.LabelColor := clRed;
 L.Parent := Form1; //теперь отображаем
end;
```

14. 3. Компонент TClock

Рассмотрим еще один пример. Создадим компонент «Цифровые часы» (`TClock`) на основе текстовой метки. Для того чтобы не перегружать Инспектор

объектов свойствами, выберем в качестве класса-предка TCustomLabel. Изначально у такого компонента вообще не будет видимых в Инспекторе объектов событий и очень мало свойств. Это происходит потому, что полезные свойства и события в классе TCustomLabel скрыты в секции protected, их нужно перенести в секцию published. Работа компонента TClock будет опираться на компонент TTimer. Основные свойства этого компонента: Enabled – включение таймера, Interval – через сколько миллисекунд происходит событие таймера OnTimer.

Наш компонент будет иметь поле для хранения объекта класса TTimer; создавать этот объект, настраивать его свойства и событие мы будем в конструкторе TClock.Create. Кроме этого, к модулю компонента необходимо подключить модуль extctrls, содержащий класс TTimer. Компонент TClock будет обладать собственным событием OnChange, которое происходит каждую секунду при включенном таймере:

```
unit Clock;
interface
uses Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, extctrls;
type
  TClock = class(TCustomLabel)
  private
    fTimer: TTimer; //содержит объект-таймер
    fOnChange: TNotifyEvent; //хранит обработчик события
    procedure TimerTick(Sender: TObject); //обновляет показания
    procedure SetEnabled(AValue: Boolean);
    function GetEnabled: Boolean;
  public
    constructor Create(AOwner: TComponent);override;
  published
    property Enabled: Boolean
                  read GetEnabled write SetEnabled default True;
    property Font; //перенесено из секции protected
    property OnMouseMove; //перенесено из секции protected
    property OnChange: TNotifyEvent
                      read FOnChange write FOnChange;
  end;
. . .
constructor TClock.Create;
begin
  inherited;
  fTimer := TTimer.Create(Self); //создаем таймер
  fTimer.Interval := 1000; //настраиваем его
```

```
  fTimer.OnTimer := TimerTick;
  fTimer.Enabled := True;
  TimerTick(Self); //обновим показания часов
end;
procedure TClock.TimerTick;
begin
  Caption := TimeToStr(Time); //выведем текущее время
  if Assigned(fOnChange) then fOnChange(Self);
end;
procedure TClock.SetEnabled;
begin
  if AValue <> fTimer.Enabled then
  begin
    fTimer.Enabled := AValue; //добираемся до таймера
    if fTimer.Enabled then TimerTick(Self); //обновим показания
  end
end;
procedure TClock.GetEnabled;
begin
  Result := fTimer.Enabled
end;
end;
```

Этот пример продемонстрировал наследование от классов вида `TCustomXXX`, перенос свойств из одной области видимости в другую и создание собственных обработчиков событий.

14. 4. Компоненты, наследники TGraphicControl

Следующий пример – рисуемый компонент, наследник класса `Tgraphic-Control`. Напоминаем, что класс `TGraphicControl` имеет виртуальный метод `Paint`. Его можно перекрывать для собственного рисования компонента. В методе `Paint` доступно свойство класса `Canvas`. Опишем компонент для вывода горизонтальной стрелки:

```
unit Arrow;
. . .
type
  TDirection = (drLeft, drRight);
  TArrow = class(TGraphicControl)
  private
    fDirection: TDirection;
    procedure SetDirection(ADirection: TDirection);
```

```
  protected
    procedure Paint;override;
  public
    constructor Create(AOwner: TComponent);override;
  published
    property Height default 20;
    property Width default 60;
    property Color;
    property Direction: TDirection
             read fDirection write SetDirection default drRight;
  end;
constructor TArrow.Create;
begin
  inherited;
  Height := 20;
  Width := 60;
  Direction := drRight
end;
procedure TArrow.SetDirection;
begin
  if ADirection <> fDirection then
  begin
    fDirection := ADirection;
    Invalidate
  end
end;
procedure TArrow.Paint;
var dx, dy: Integer;
begin
  inherited;
  dx := Width div 6;
  dy := Height div 2;
  with Canvas do
  begin
    if csDesigning in ComponentState then
    begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle(0, 0, Width, Height)
    end;
    Pen.Style := psSolid;
```

```
      Pen.Color := Color;
      case Direction of
        drRight:   PolyLine([Point(Width-dx,0),   Point(Width,dy),
Point(Width-dx,Height), Point(Width,dy), Point(0,dy)]);
        drLeft:        PolyLine([Point(dx,0),        Point(0,dy),
Point(dx,Height), Point(0,dy), Point(Width,dy)]);
      end
    end
  end;
```

Обратим внимание на следующие моменты. Для задания направления стрелки введен перечисляемый тип `TDirection`. Работа с таким типом полностью поддерживается Инспектором объектов. Свойства `Width` и `Height` перенесены из секции `protected` в секцию `published` с указанием нового значения по умолчанию. В методе `Paint` использовано свойство `ComponentState` класса `TComponent`, которое позволяет контролировать, в каком режиме находится компонент. В данном случае `ComponentState` используется для рисования пунктирной линии на границе компонента в режиме проектирования формы.

14.5. Компоненты с обработкой сообщений

Последний пример компонента демонстрирует использование методов обработки сообщений. В примере обрабатываются внутренние сообщения Delphi о проходе указателя мыши над компонентом. Реализована «активная кнопка», заголовок которой становится полужирным, когда на кнопку указывает мышь:

```
  unit ActiveButton;
  type
    TActiveButton = class(TButton)
    protected
    procedure MouseEnter(var Msg: TMessage);message cm_mouseEnter;
    procedure MouseLeave(var Msg: TMessage);message cm_mouseLeave;
    end;
  procedure TActiveButton.MouseEnter(var Msg: TMessage);
  begin
    Font.Style := Font.Style + [fsBold]
  end;
  procedure TActiveButton.MouseLeave(var Msg: TMessage);
  begin
    Font.Style := Font.Style - [fsBold]
  end;
```

15. Взаимодействие приложений. Работа с сообщениями

В этом разделе будут описаны несколько приемов совместной работы приложений, созданных в Delphi, и внешних программ. Приведенная информация не претендует на полноту и для более глубокого изучения необходима дополнительная литература.

15.1. Запуск внешних программ

Для запуска внешних программ из приложения, созданного в Delphi, можно использовать следующую функцию:

```
function WinExec(CmdLine: PChar; CmdShow: Integer): Integer;
```

Параметры функции означают следующее:

`CmdLine` – имя исполняемого файла с параметрами. Если это имя задано без пути, то файл ищется по правилам, определенным для поиска DLL;

`CmdShow` – вид окна запускаемого приложения (одна из констант `SW_RESTORE`, `SW_MAXIMIZE` и т. п.).

Если запуск программмы успешен, функция возвращает значение большее, чем 31. Меньшее значение говорит об ошибке (0 – мало памяти, 2 – файл не найден и т. п.). Примеры использования функции:

```
WinExec('nc', SW_RESTORE); // запуск Norton Commander
WinExec('command.com', SW_RESTORE); // запуск сеанса DOS
```

Альтернативной возможностью является использование функции `ShellExecute(Wnd: HWnd; Operation, FileName, Parameters, Directory: PChar; CmdShow: Integer): Integer`. По сравнению с `WinExec` она обладает более широкими возможностями, к примеру, может не только запустить приложение, но и напечатать документ. Смысл параметров функции следующий:

`Wnd` – дескриптор родительского окна (обычно указывают свойство `Handle`);

`Operation` – операция (`'open'` – открыть файл, `'print'` – распечатать файл, `'explore'` – открыть папку в Проводнике Windows);

`FileName` - имя файла или папки;

`Parameters` – параметры для файла (или `nil`);

`Directory` – каталог по умолчанию;

`CmdShow` – аналогично `WinExec`.

Для возвращаемого `ShellExecute` значения справедливо то же, что и для `WinExec`. Приведем примеры использования функции:

```
ShellExecute(Handle, 'print', 'file.doc', nil, nil, SW_RESTORE)
//запуск редактора Word и печать файла file.doc
ShellExecute(Handle, 'explore', 'c:\Work', nil, nil, SW_RESTORE)
//откроет папку c:\Work в Проводнике
```

Функция FindExecutable(FileName,Directory,Buffer:PChar): THandle позволяет получить имя *exe*-файла, связанного с программой, указанной в параметре FileName, и заносит ответ в параметр Buffer:

```
var Answ: array[0..254] of Char; //переменная для записи ответа
. . .
FindExecutable('file.doc',nil,Answ); //Answ ='C:\…\winword.exe'
```

15.2. Базовые моменты технологии COM

Под *технологий COM* (Component Object Mode) понимается объектно-ориентированный механизм интеграции приложений. По сути, эта технология позволяет одним приложениям предоставить для работы свои внутренние объекты, а другим приложениям управлять этими объектами. *COM-сервер* – это приложение, объекты которого доступны другим приложениям (все такие приложения зарегистрированы в реестре, так что их можно найти по уникальным именам). *COM-контроллер* (или *COM-клиент*) – приложение, которое обращается к объектам COM-сервера.

Рассмотрим простейший пример кода COM-клиента, который работает с COM-сервером Microsoft Word:

```
procedure TForm1.Button1Click(Sender: TObject);
var MSWord: Variant; //переменная для доступа к COM-серверу
begin
  MSWord := CreateOLEObject('Word.Basic');
  //создание объекта для доступа к COM-серверу
  MSWord.FileNewDefault; //создаем новый документ
  MSWord.Insert('Hello from Delphi'); //вставляем в него текст
  MSWord.FileSaveAs('C:\Work\hello.doc'); //сохраняем в файле
  MSWord := UnAssigned; //освобождаем объект
end;
```

Функция CreateOLEObject объявлена в модуле COMObj. Методы, которые мы вызываем у MSWord, являются не методами Object Pascal, а методами VBA, они расположены в другом приложении и могут иметь непаскалевский синтаксис. Приведенный пример несколько устарел. Начиная с Delphi 3, для подобных целей используются так называемые *интерфейсные типы* (interface types).

15.3. Сообщения. Обработка и перехват сообщений

Дадим некоторые определения:

Дескриптор (Handle) – уникальный целочисленный идентификатор, позволяющий системе однозначно определить некий объект (окно, файл, ресурс);

Сообщение (Message) – системная структура данных, содержащая несколько числовых полей. Служит для взаимодействия системы и приложений, а так-

же приложений между собой и с отдельными частями одного приложения. Системные сообщения Windows однозначно определяются своим номером (одно из полей структуры).

В Delphi дескрипторам соответствуют собственные типы `THandle`, `HDC`, `HWND` (фактически по формату совпадают с типом `Integer`), а сообщениям системы – тип `TMsg`.

Сделаем небольшое отступление и рассмотрим основные этапы создания программы для Windows без VCL. Такую программу можно написать в Delphi, этот процесс напоминает создание консольного приложения, для которого понадобятся два модуля – `windows` и `messages`. Первый является интерфейсным модулем к набору DLL, содержащих системные функции Windows, а второй содержит именованные константы для различных сообщений (чтобы не употреблять их числовые идентификаторы) и несколько полезных типов данных. Подробно программа воспроизводиться не будет, так как ее текст достаточно большой (40–60 строк). Структура программы совпадает со структурой любого Windows-приложения:

1) регистрация класса окна;

2) создание окна;

3) цикл обработки сообщений (взять сообщение из очереди – оттранслировать – направить в систему, продолжать цикл до получения сообщения о выходе из программы);

4) оконная функция, в которую, в конце концов, попадают все сообщения и которая содержит оператор `case` для распознавания конкретных сообщений.

Все моменты здесь ключевые, подробнее они изучаются при прохождении курса «Системное программирование».

Попробуем разобраться, в каких местах Delphi скрыты эти ключевые моменты. В простейшей программе, созданной в Delphi, как правило, существует несколько элементов, соответствующих окнам в терминах Windows (кнопка, окно формы, список и т.д.). Все такие элементы являются потомками класса `TWinControl` и теоретически могут получать сообщения от системы. При этом сообщение системы проходит по следующей цепочке:

1) вызывается функция окна `MainWndProc`;

2) `MainWndProc` вызывает метод, адрес которого находится в свойстве `WindowProc`. По умолчанию это свойство содержит адрес виртуальной процедуры `WndProc`;

3) `WndProc` реализует обработку некоторых сообщений, после такой обработки вызывается метод `Dispatch`;

4) `Dispatch` ищет динамический метод-обработчик сообщения по идентификатору сообщения;

5) если `Dispatch` не нашел обработчик, он вызывает метод `DefaultHandler`, который передает сообщения обработчику сообщений по умолчанию из Windows.

Все вышеуказанные методы работают с сообщениями не типа `TMsg`, а типа `TMessage`. Кроме объявления `TMessage` в модуле `messages` содержатся объявле-

ния типов, идентичных TMessage по длине, но интерпретирующих данные типа в виде различных структур для упрощения доступа к параметрам сообщения.

Сам цикл обработки сообщений инкапсулирован в методе TApplication.ProcessMessage. В ходе работы этого метода вызывается обработчик события TApplication.OnMessage

Теперь опишем возможности перехвата и реакции на сообщения:

1. Перекрытие виртуального метода WndProc окна (формы, кнопки). При этом не забывать вызвать перекрытый обработчик.

2. Написание своего обработчика сообщений (с ключевым словом message).

3. Перекрытие виртуального метода DefaultHandler (редкий подход)

4. Написание обработчика события OnMessage для перехвата сообщений, поставленных в очередь приложения.

Рассмотрим следующий пример. Будем отлавливать щелчки правой кнопки мыши на форме. В проекте используются вышеописанные возможности перекрытия:

```
  type
    TForm1 = class(TForm)
      Label1: TLabel;
      procedure FormMouseDown(Sender: TObject; Button: TMouseBut-
ton; Shift: TShiftState; X, Y: Integer);
      procedure FormCreate(Sender: TObject);
    public
      procedure WndProc(var Message: TMessage); override;
      procedure    WmLButDown(var    Message:    TWMMouse);message
WM_LBUTTONDOWN;
      procedure DefaultHandler(var Message); override;
      procedure AppMessage(var Msg: TMsg; var Handled: Boolean);
    end;
  . . .
  procedure TForm1.WndProc(var Message: TMessage);
  begin
    if Message.Msg = wm_LButtonDown then ShowMessage('WndProc');
    inherited
  end;
  procedure TForm1.WmLButDown(var Message:TWMMouse);
  begin
    ShowMessage('WmLButDown');
    inherited
  end;
  procedure TForm1.DefaultHandler(var Message);
  begin
```

```
    if TMessage(Message).Msg = wm_LButtonDown then ShowMes-
sage('DefHandler');
    inherited;
  end;
  procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
  begin
    if  Button = mbLeft then ShowMessage('OnMouseDown')
  end;
  procedure TForm1.AppMessage(var Msg: TMsg; var Handled: Boo-
lean);
  begin
    if Msg.message = WM_LBUTTONDOWN then begin
      Label1.Caption := 'OnMessage';
      Handled := False;
    end;
  end;
```

Обратите внимание на последовательность, с которой происходит обработка сообщений: `OnMessage` – `WndProc` – `WmLButDown` – `DefaultHandler` – `OnMouseDown`. Так же обратите внимание на использование специальных типов и констант для распознавания сообщений.

К перехватчикам сообщений (`WmLButDown`) предъявляются следующие требования: перехватчик сообщения всегда является процедурой, которая должна иметь единственный `var`-параметр; после имени процедуры указываются директива `message` и один числовой параметр (или константа), соответствующий обрабатываемому сообщению; в перехватчике, как правило, вызывается унаследованный метод.

Кроме системных сообщений компоненты Delphi определяют много собственных сообщений, которые также можно обработать перехватчиками сообщений.

15.4. Посылка сообщений. Управление окнами

Приведем несколько функций, при помощи которых можно работать с окнами и компонентами, используя сообщения. Во всех функциях, как правило, одним из параметров выступает дескриптор окна. Если компонент порожден от класса `TWinControl`, то он имеем соответствующее свойство `Handle`. Иначе можно попытаться узнать дескриптор через вызов функции:

```
function FindWindow(ClassName, WndName: PChar): HWnd
```

Здесь `ClassName` – имя класса окна, `WndName` – его заголовок. Для форм и компонент Delphi эти параметры определяются просто (`TForm1`, `Form1`). Для других окон может помочь входящая в состав Delphi утилита WinSight32.

Имея дескриптор, можно использовать, например, такие функции:

`function CloseWindow(hWnd: HWnd): BOOL` – сворачивает окно;
`function EnableWindow(hWnd: HWnd; en: BOOL): BOOL` – делает окно доступным или недоступным.

Для посылки сообщений можно использовать функцию:

```
function SendMessage(hWnd: HWnd; Msg, wParam: word; lParam: Lon-
gint): Longint,
```

которая осуществляет посылку сообщения `Msg` с параметрами `wParam` и `lParam` окну с дескриптором `hWnd`. Функция ждет, пока сообщение не будет обработано.

Функция

```
function PostMessage(hWnd: HWnd; Msg,wParam: word; lParam: Lon-
gint): Longint
```

по параметрам и действию аналогична `SendMessage`, но передает сообщение не окну, а ставит его в очередь сообщений приложения.

Метод класса `TControl`

```
function Perform(Msg: Cardinal; WParam, LParam: Longint): Lon-
gint;
```

посылает сообщение тому компоненту, у которого вызывается (через вызов его оконной функции).

Литература

1. Архангельский А. Программирование в Delphi 6. СПб.: Бином, 2001.
2. Архангельский А. Object Pascal в Delphi. СПб.: Бином, 2002.
3. Бобровский С. Delphi 6 и Kylix. Библиотека программиста. СПб.: Питер, 2002.
4. Гофман В., Хомоненко А. Delphi 6 в подлиннике. СПб.: BHV, 2001.
5. Елманова Н., Тенцер А., Трепалин С. Delphi 6 и технологии COM. СПб.: Питер, 2002.
6. Карпов Б. Delphi: Специальный справочник. СПб.: Питер, 2001.
7. Кэнту М. Delphi 6 для профессионалов. СПб.: Питер, 2002.
8. Лишнер Р. Delphi: Справочник. СПб.: Питер, 2001.
9. Озеров В. Delphi. Советы программистов. М.: Символ-Плюс, 2002.
10. Пачеко К., Тейксейра С. Borland Delphi 6: Руководство разработчика. М.: Вильямс, 2002.
11. Стивенс Р. Delphi. Готовые алгоритмы. М.: ДМК, 2001.

Св. план 2002, поз. 53

Учебное издание

Волосевич Алексей Александрович

ЯЗЫК OBJECT PASCAL И СИСТЕМА ПРОГРАММИРОВАНИЯ DELPHI

Учебное пособие

по курсу «Инструментарий систем программирования»
для студентов специальности 31 03 04 «Информатика»
дневной формы обучения

Редактор Н.А. Бебель
Корректор Е.Н. Батурчик
Компьютерная верстка Т.В. Шестакова

Подписано в печать 18.03.2003. Формат 60x84 1/16. Бумага офсетная.
Печать ризографическая. Гарнитура «Таймс». Усл.-печ. л. 3,72.
Уч.-изд. л. 3,3. Тираж 100 экз. Заказ 719.

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП № 156 от 30.12.2002.
Лицензия ЛВ № 509 от 03.08.2001.
220013, Минск, П. Бровки, 6