Язык Delphi

ГЛАВА

5

Стив Тейксейра

В ЭТОЙ ГЛАВЕ...

- **И это все о платформе .NET**
- **Комментарии**
- **Процедуры и функции**
- **Переменные**
- **Константы**
- **Операторы**
- **Типы данных Delphi**
- **Пользовательские типы данных**
- **Приведение и преобразование типов**
- **Строковые ресурсы**
- **Условные операторы**
- **Циклы**
- **Процедуры и функции**
- **Область видимости**
- **Модули и пространства имен**
- **Пакеты и сборки**
- **Объектно-ориентированное программирование**
- **Использование объектов Delphi**
- **Структурная обработка исключений**

В этой главе рассматривается язык Object Pascal, лежащий в основе Delphi, представлено введение в основы языка Delphi, включая его правила и конструкции. Затем рассматривается несколько более сложных аспектов языка Delphi, таких как применение классов и обработка исключений. Предполагается, что читатель имеет некоторый опыт работы с другими высокоуровневыми языками. Поэтому здесь не рассматриваются концепции, связанные с языками программирования вообще и с Delphi в частности. Завершив изучение этой главы, читатель будет ознакомлен с такими концепциями программирования, как применение переменных, типов, операторов, циклов, оператора `case`, исключений и объектов в Delphi, а также их взаимоотношением с платформой .NET Framework. Для наглядности элементы языка Delphi будут сравниваться с соответствующими элементами таких наиболее широко известных языков платформы .NET, как C# и Visual Basic .NET.

И это все о платформе .NET

Язык Delphi 8 позволяет создавать приложения, выполняющиеся в контексте платформы .NET Framework от *Microsoft*. Следовательно, достоинства и возможности компилятора Delphi 8 полностью зависят от возможностей и достоинств базовой платформы .NET Framework. Эта сентенция могла бы дезориентировать тех, кто переходит к платформе .NET из мира базового кода. По существу, базовый компилятор кода может сделать только то, на что он способен, т.е. его возможности ограничены тем, что предусмотрел для него производитель компилятора. При разработке на платформе .NET возможно буквально все, от тривиального сложения двух целых чисел до создания компилятором кода, который управляет элементами и типами самой платформы .NET Framework.

Вместо *базового кода* (native code) компилятор .NET, такой как Delphi 8, создает код в формате промежуточного языка Microsoft (MSIL), являющийся наиболее низкоуровневым представлением команд приложения.

НА ЗАМЕТКУ

Принципы JIT-компиляции платформы .NET Framework обсуждаются в главе 2, “Обзор среды .NET Framework”. Сейчас, вероятно, наилучшее время, чтобы вернуться к этой информации.

Комментарии

Язык Delphi поддерживает три типа комментариев: с использованием фигурных скобок, скобки и звездочки, а также двойной наклонной черты. Примеры каждого типа комментариев приведены ниже.

```
{  Комментарий с использованием фигурных скобок }
(* Комментарий с использованием скобок и звездочек *)
// Комментарий с использованием двойной наклонной черты
```

Поведение первых двух типов комментариев практически идентично. Компилятор считает комментарием все, что находится между символом (или парой символов) начала комментария и символом завершения комментария. Комментарий в виде двой-

ной косой черты продолжается только до конца строки, а все, что находится после конца строки, комментарием не считается.

НА ЗАМЕТКУ

Вложение комментариев того же самого типа недопустимо. Хотя вложение комментариев различных типов внутрь друг друга вполне допустимо, пользоваться этой возможностью не рекомендуется. Примеры вложенных комментариев приведены ниже.

```
{ (*  Допустимо *) }
(* {  Допустимо } *)
(* (* Недопустимо *) *)
{ {   Недопустимо } }
```

Еще одна методика исключения фрагментов кода, особенно удобная, когда в исключаемом коде встречаются комментарии различных типов, подразумевает использование директивы компилятора `$IFDEF`. В примере, приведенном ниже, директива `$IFDEF` используется для комментирования фрагмента кода, уже содержащего комментарий.

```
{$IFDEF DONTCOMPILEME}
// здесь мог находиться большой блок кода
{$ENDIF}
```

Поскольку идентификатор `DONTCOMPILEME` (не компилировать) не определен, код внутри директивы `$IFDEF` компилироваться не будет, а следовательно, он окажется закомментирован.

Процедуры и функции

Поскольку процедуры и функции — это довольно обширная тема для любых языков программирования, не будем пока углубляться в данный вопрос, а отметим лишь малоизвестные и новые возможности в этой области.

НА ЗАМЕТКУ

Функции (function), не возвращающие значения (на языке C# возвращающие тип `void`), называются *процедурами* (procedure), а функции, возвращающие значение определенного типа, остаются функциями. Зачастую термин *функция* используется для описания не только процедур и функций, но и *методов* (method), которые фактически являются функциями, расположенными внутри класса.

Круглые скобки при вызове функций

Одной из наименее известных особенностей языка Delphi является то, что при вызове процедуры или функции, которой никакие параметры не передаются, круглые скобки вовсе необязательны. Следовательно, приведенный ниже синтаксис вызова функций вполне допустим.

```
Form1.Show;
Form1.Show();
```

Безусловно, эта возможность не из тех, которые сильно удивляют. Однако программисты, вынужденные делить свое рабочее время между языками Delphi и C#, где круглые скобки необходимы, оценят ее по достоинству, ибо им не придется задумываться о различии синтаксиса вызова функций в разных языках.

Перегрузка

Язык Delphi поддерживает концепцию *перегрузки функций* (overloading), которая позволяет иметь несколько одноименных функций или процедур с разными списками параметров. Все перегруженные методы и функции должны быть объявлены с применением директивы `overload`, как показано ниже.

```
procedure Hello(I: Integer); overload;
procedure Hello(S: string); overload;
procedure Hello(D: Double); overload;
```

Обратите внимание: правила перегрузки методов класса немного отличаются от правил перегрузки обычных процедур и функций. Они рассматриваются в разделе "Перегрузка методов" этой главы.

Значения параметров по умолчанию

Язык Delphi обладает еще одной весьма полезной возможностью: *значениями параметров по умолчанию* (default value parameter). Это позволяет установить принимаемое по умолчанию значение параметра процедуры или функции. Такое значение будет использовано в тех ситуациях, когда вызов процедуры или функции осуществляется без указания значения данного параметра. В объявлении процедуры либо функции принимаемое по умолчанию значение параметра указывается после знака равенства, следующего после его имени, как показано в примере ниже.

```
procedure HasDefVal(S: string; I: Integer = 0);
```

Процедура `HasDefVal()` может быть вызвана двумя способами. В первом случае — как обычно, с указанием обоих параметров.

```
HasDefVal('hello', 26);
```

Во втором случае можно задать значение только параметра `S`, а для параметра `I` использовать значение по умолчанию.

```
HasDefVal('hello');  // для I используется значение по умолчанию
```

Используя значения параметров по умолчанию, следует придерживаться правил, приведенных ниже.

- Параметры, имеющие значения по умолчанию, должны располагаться в конце списка параметров. Параметр без значения по умолчанию не должен встречаться в списке после параметра, имеющего значение по умолчанию.
- Значения по умолчанию могут присваиваться таким параметрам обычных типов, как перечисление, строковый тип, число с плавающей запятой, указатель или набор. Классы, интерфейсы, динамические массивы, ссылки на процедуры и классы также допустимы, но только в том случае, когда значение по умолчанию равно `nil`.
- Значения по умолчанию могут передаваться только по значению либо как константа с модификатором `const`. Они не могут быть ссылками (`var`, `out`) или нетипизированным параметром.

Одним из важнейших преимуществ применения значений параметров по умолчанию является простота расширения функциональных возможностей уже существующих процедур и функций при соблюдении обратной совместимости. Предположим, на рынок программных продуктов был выпущен модуль, ключевым звеном которого является функция `AddInts()`, складывающая две целые величины.

```
function AddInts(I1, I2: Integer): Integer;
begin
  Result := I1 + I2;
end;
```

Допустим, что впоследствии программе понадобилось добавить возможность сложения трех чисел. Однако замена существующей функции другой, способной складывать три числа, приведет к необходимости исправить и перекомпилировать весь существующий код, в котором происходит вызов этой функции. Но при использовании значений параметров по умолчанию возможности функции `AddInts()` расширяются без необходимости перекомпиляции. Для этого достаточно изменить объявление функции следующим образом.

```
function AddInts(I1, I2: Integer; I3: Integer = 0);
begin
  Result := I1 + I2 + I3;
end;
```

СОВЕТ

Как правило, выбор средства добавления новых возможностей без нарушения совместимости с прежней версией сводится к применению перегрузки функций или значений по умолчанию. Применение перегрузки не только немного эффективней, но и позволяет обеспечить совместимость с другими языками .NET, поскольку значения параметров по умолчанию не поддерживаются языком C# или управляемым C++.

Переменные

У некоторых программистов бытует мнение: “Если мне понадобится еще одна переменная — я опишу ее прямо здесь, посреди кода, в том месте, где она понадобилась”. Для таких языков, как C# или Visual Basic .NET, это вполне допустимо. Но в языке Delphi все переменные обязательно должны быть описаны в соответствующем разделе `var`, расположенном в начале процедуры, функции или программы. Предположим, на языке C# код функции выглядит так.

```
public void foo()
{
  int x = 1;
  x++;
  int y = 2;
  float f;
  //... и так далее ...
}
```

В Delphi аналогичная функция должна выглядеть следующим образом.

```
procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  //... и так далее ...
end;
```

Чувствительность к регистру символов

С точки зрения чувствительности к регистру символов язык Delphi подобен языку Visual Basic .NET, но отличается от языка C#. Использование строчных и прописных символов позволяет сделать код более удобочитаемым, однако решение об их применении зависит от личных предпочтений разработчика. Стиль, принятый в этой книге, подразумевает их применение. Если имя идентификатора состоит из нескольких слов, то первый символ каждого слова начинается прописной буквой (верблюжья нотация). Например, приведенное ниже имя процедуры воспринять довольно сложно.

```
procedure thisprocedurenamemakesnosense;
```

Имя процедуры в следующем виде значительно понятней.

```
procedure ThisProcedureNameIsMoreClear;
```

Возникает вполне резонный вопрос: в чем же тогда преимущество и удобство языка Delphi? Однако вскоре будет продемонстрировано, что столь строгая структура кода делает его понятным, удобочитаемым и, как следствие, более простым в отладке и сопровождении, чем у других языков, которые полагаются на общепринятые соглашения, а не на жесткие правила.

Обратите внимание: Delphi позволяет объявить в одной строке несколько переменных одинакового типа (равно как и формальных параметров). Для этого применяется следующий синтаксис.

```
ИмяПеременной1, ИмяПеременной2: НекоторыйТип;
```

Применение данной возможности позволяет сделать код намного более компактным и удобочитаемым, чем в таком языке, как C#, где тип каждой переменной или параметра должен быть задан отдельно.

Объявление переменной в Delphi состоит из имени, двоеточия и типа переменной. Не забывайте: локальные переменные *всегда* инициализируются *отдельно* от своего объявления.

Глобальные переменные в Delphi можно инициализировать внутри блока `var`. Приведенный ниже пример демонстрирует синтаксис такого объявления.

```
var
  i: Integer = 10;
  S: string  = 'Hello world';
  D: Double  = 3.141579;
```

НА ЗАМЕТКУ

Предварительная инициализация (preinitialization) возможна только для глобальных переменных и недопустима для переменных, являющихся локальными для процедур или функций.

Автоматическая инициализация нулевым значением

Среда CLR автоматически инициализирует все переменные нулевым значением. Таким образом, при запуске приложения или вызове функции всем переменным целочисленных типов будет присвоено значение `0`, переменным типа числа с плавающей запятой — значение `0.0`, объектам — значение `nil`, строкам — пустая строка и т.д. Следовательно, инициализировать переменные в исходном коде нулевым значением больше не обязательно.

Константы

Для определения констант в языке Delphi применяется директива `const`, действие которой аналогично ключевому слову `const` языка C#. Ниже приведен пример объявления трех констант на языке C#.

```
public const float ADecimalNumber = 3.14;
public const int i = 10;
public const String ErrorString ="Danger, Danger, Danger!";
```

Основное различие между константами в языках C# и Delphi заключается в том, что в Delphi (как и в Visual Basic .NET) не требуется указывать тип объявляемой константы. Компилятор Delphi автоматически выделяет необходимую память, основываясь на инициализирующем значении (а при использовании таких скалярных констант, как целое число, он просто подставляет значение в нужные места в программе, совсем не выделяя память). Объявления тех же констант в Delphi выглядят следующим образом.

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Danger, Danger, Danger!';
```

Хоть и необязательно, но в объявлении константы можно указать ее тип. Это позволит компилятору лучше контролировать типы данных таких констант.

```
const
  ADecimalNumber: Double = 3.14;
  I: Integer = 10;
  ErrorString: string = 'Danger,Danger,Danger!';
```

Типизированные константы

Типизированные константы (typed constant) имеют одно преимущество перед нетипизированными типами. Для нетипизированных констант не производится строгий контроль типов данных во время компиляции, что не позволяет применить к ним вызов методов базовых объектов. Например, следующий код вполне допустим.

```
const
  I: Integer = 19710704;
```

```
  S: string;
begin
  S := I.ToString;
```

А вот следующий код откомпилирован не будет.

```
const
  I = 19710704;
  S: string;
begin
  S := I.ToString;
```

Язык Delphi допускает применение в объявлениях констант (`const`) и переменных (`var`) функций, вычисляемых во время компиляции. К этим функциям относятся: `Ord()`, `Chr()`, `Trunc()`, `Round()`, `High()`, `Low()`, `Abs()`, `Pred()`, `Succ()`, `Length()`, `Odd()`, `Round()`, `Trunc()` и `SizeOf()`. Таким образом, все объявления в следующем примере абсолютно корректны.

```
type
  A = array[1..2] of Integer;;
const
  w: Word = SizeOf(Byte);
var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: Longint = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: Char = Chr(46);
```

ВНИМАНИЕ!

Для обеспечения совместимости с прежней версией компилятор снабжен флажком, который позволяет присваивать типизированным константам значения точно так же, как и переменным. Флажок Assignable typed constants (Присваиваемые типизированные константы) расположен на вкладке Compiler (Компилятор) диалогового окна Project Options (Параметры проекта). Вместо него можно воспользоваться директивой компилятора `$WRITEABLECONST` (или `$J`). Но этот режим (официально) считается неприемлемым, а во избежание случайного применения он по умолчанию отключен.

НА ЗАМЕТКУ

Подобно языкам C# и Visual Basic .NET, язык Delphi не имеет препроцессора, как у языка C. Таким образом, в языке Delphi отсутствует концепция макрокоманд, а следовательно, отсутствует и эквивалент директивы препроцессора `#define`, используемой в языке C для описания констант. Хотя язык Delphi и позволяет выполнять условную компиляцию с помощью директивы компилятора `$define`, ее нельзя применять для определения констант, как в языке C. Там, где в языке C применялась директива `#define`, в языке Delphi рекомендуется объявлять константы (`const`).

Операторы

Оператор (operator) — это один или несколько символов кода, с помощью которых выполняются определенные действия с данными различных типов. Простейшим примером могут служить операторы сложения, вычитания, умножения и деления арифметических типов данных; другим примером может быть оператор доступа к определенному элементу массива. Здесь рассматриваются операторы языка Delphi и их корреляция с аналогами в C# и CLR.

Операторы присвоения

Тем, кто не обладает опытом работы на языке Pascal, в Delphi трудней всего будет привыкнуть к оператору присвоения. Чтобы присвоить значение переменной, используется оператор `:=`, аналогичный оператору `=` в языках C# и Visual Basic .NET. Иногда программисты называют его оператором *возвращения значения* (get). Давайте рассмотрим пример.

```
Number1 := 5;
```

Это выражение можно прочитать так: "переменная `Number1` *получает* значение `5`" или "переменной `Number1` *присваивается* значение `5`".

Операторы сравнения

Тем, кто уже работал с Visual Basic .NET, операторы сравнения Delphi покажутся знакомыми, поскольку они фактически идентичны. Во всех языках программирования эти операторы довольно похожи, поэтому в данном разделе они рассматриваются очень кратко.

На языке Delphi оператор `=` используется для логического сравнения двух выражений или значений. В языке C# ему соответствует оператор `==`, пример применения которого приведен ниже.

```
if (x == y)
```

На языке Delphi этот же оператор выглядит следующим образом.

```
if x = y
```

НА ЗАМЕТКУ

Не забывайте, что на языке Delphi оператор `:=` используется для присвоения значения, а оператор `=` — для сравнения значений двух операндов.

В Delphi оператор "не равно" выглядит как `<>`. Он аналогичен оператору `!=` языка C#. Чтобы проверить неравенство двух значений, применяется следующий код.

```
if x <> y then ВыполнитьДействия
```

Логические операторы

Delphi использует ключевые слова `and` и `or` в качестве логических операторов "и" и "или" (в языке C# для этого применяются соответственно операторы `&&` и `||`). Операторы `and` и `or` применяются в основном как элементы оператора `if` или цикла. Например, так.

```
if (Условие 1) and (Условие 2) then
  ВыполнитьДействия;

while (Условие 1) or (Условие 2) do
  ВыполнитьДействия;
```

Логический оператор Delphi "не" выглядит как `not` (он аналогичен оператору `!` в языке C#). Оператор `not`, в основном, применяется в составе оператора `if`. Например, так.

```
if not (условие) then (ВыполнитьДействия); // если условие ложно,
                                           // тогда...
```

Список операторов Delphi и соответствующих им операторов языков C# и Visual Basic .NET приведен в табл. 5.1.

Таблица 5.1. Операторы присвоения, сравнения и логические операторы

Оператор	*Delphi*	*C#*	*Visual Basic .NET*
Присвоение	:=	=	=
Сравнение	=	==	= или Is*
Неравенство	<>	!=	<>
Меньше, чем	<	<	<
Больше, чем	>	>	>
Меньше или равно	<=	<=	<=
Больше или равно	>=	>=	>=
Логическое and	and	&&	And
Логическое or	or	||	Or
Логическое not	not	!	Not
Логическое xor	xor	^	Xor

* Оператор сравнения `Is` в языке Visual Basic .NET используется только для объектов, а для сравнения всех остальных типов данных применяется оператор =.

Арифметические операторы

Большинство арифметических операторов Delphi должно быть знакомо всем, поскольку они идентичны операторам, используемым в других языках. Список арифметических операторов Delphi и соответствующих им операторов языков C# и Visual Basic .NET приведен в табл. 5.2.

Таблица 5.2. Арифметические операторы

Оператор	*Delphi*	*C#*	*Visual Basic .NET*
Сложение	+	+	+
Вычитание	-	-	-
Умножение	*	*	*
Деление с плавающей точкой	/	/	/
Деление целых чисел	div	/	\
Деление по модулю	mod	%	Mod
Возведение в степень	Отсутствует	Отсутствует	^

Как можно заметить в таблице, языки Delphi и Visual Basic .NET для деления целых чисел и чисел с плавающей точкой используют разные математические операторы, в отличие от языка C#. Оператор `div` автоматически отсекает остаток при делении двух целочисленных выражений.

НА ЗАМЕТКУ

Не забывайте использовать правильный оператор деления для соответствующих типов выражений. Компилятор Delphi выдаст сообщение об ошибке при попытке деления двух чисел с плавающей точкой при помощи оператора `div` или попытке присвоения результата деления двух целых чисел при помощи оператора `/`, способного создать число с плавающей точкой. Например, так.

```
var
  i: Integer;
  d: Double;
begin
  i := 4 / 3;        // Эта строка приведет к ошибке компиляции
  d := 3.4 div 2.3;  // Эта строка тоже приведет к ошибке
end;
```

Чтобы присвоить целочисленной переменной результат деления двух целых чисел, полученный с помощью оператора `/`, его следует преобразовать в целое число, используя функцию `Trunc()` или `Round()`.

Побитовые операторы

Побитовые операторы (bitwise operator) позволяют изменять отдельные биты значения переменной. Чаще всего побитовые операторы используются для сдвига битов влево или вправо, их инверсии, а также побитовых операций "и", "или" и "исключающее или" (`xor`) между двумя числами. Операторы *сдвига влево* `shl` (shift-left) и *сдвига вправо* `shr` (shift-right) языка Delphi аналогичны операторам << и >> языка C#. Запомнить остальные побитовые операторы Delphi тоже достаточно легко — это `and`, `not`, `or` и `xor`. Список побитовых операторов приведен в табл. 5.3.

Таблица 5.3. Побитовые операторы

Оператор	*Delphi*	*C#*	*Visual Basic .NET*
И	`and`	`&`	`And`
Не	`not`	`~`	`Not`
Или	`or`	`|`	`Or`
Исключающее или	`xor`	`^`	`Xor`
Сдвиг влево	`shl`	`<<`	Отсутствует
Сдвиг вправо	`shr`	`>>`	Отсутствует

Процедуры инкремента и декремента

Процедуры инкремента и декремента упрощают код, применяемый для добавления или вычитания единицы из целочисленной переменной. Язык Delphi не предоставляет таких широких возможностей, как постфиксные и префиксные операторы `++` и `--` языка C#. Тем не менее в языке Delphi для этого применяются процедуры `Inc()` и `Dec()`.

Процедуры `Inc()` и `Dec()` можно вызывать и с одним, и с двумя параметрами. Ниже приведен пример их вызова с одним параметром.

```
Inc(variable);
Dec(variable);
```

Ниже значение переменной `variable` увеличивается и уменьшается на `3`.

```
Inc(variable, 3);
Dec(variable, 3);
```

Сравнение операторов инкремента и декремента различных языков приведено в табл. 5.4.

Таблица 5.4. Операторы инкремента и декремента

Оператор	*Delphi*	*C#*	*Visual Basic .NET*
Инкремент	`Inc()`	`++`	Отсутствует
Декремент	`Dec()`	`--`	Отсутствует

Обратите внимание: в языке C# операторы `++` и `--` возвращают значение, в то время как процедуры `Inc()` и `Dec()` этого не делают. Функции Delphi `Succ()` и `Pred()` возвращают значение, однако они не изменяют значение переданного параметра.

Операторы присвоения с действием

Весьма удобные *операторы присвоения с действием* (do-and-assign operator), присущие языку C#, в языке Delphi отсутствуют. Это такие операторы, как присвоение с суммой `+=` и присвоение с умножением `*=`. Они выполняют арифметическую операцию (в данном случае — сложение и умножение) перед операцией присвоения. В Delphi этот тип операций следует выполнять, используя два разных оператора. Например, следующий код C#

```
x += 5;
```

на языке Delphi выглядит так.

```
x := x + 5;
```

Типы данных Delphi

Одним из самых больших достоинств языка Delphi является строгий контроль типов данных, обеспечивающий его превосходную совместимость с платформой .NET. В частности, это означает, что все реальные переменные, передаваемые в качестве параметров в функции или процедуры, должны абсолютно точно соответствовать типу формальных параметров в объявлении данной функции или процедуры, что несколько неудобно с точки зрения неявного преобразования типов. В целом, строгий контроль типов данных языка Delphi позволяет предотвратить попытку заткнуть квадратной пробкой круглую дыру. В конце концов, проще всего исправить те ошибки, о которых компилятор сообщает сразу.

Объекты, объекты повсюду

Одной из наиболее важных концепций базовых типов данных версии компилятора Delphi for .NET является то, что типы значений могут быть неявно преобразованы в классы. Преобразование типа значения в объект и обратно в языках платформы .NET называется *упаковкой* (boxing) и *распаковкой* (unboxing). Целые числа, строки, числа с плавающей точкой и все остальные типы, не реализованные в компиляторе как базовые, могут быть приведены в соответствие с типами значений либо платформы .NET Framework, либо библиотек RTL или VCL от *Borland*. В отличие от Delphi Win32, эти типы значений могут иметь свои собственные процедуры и функции, в дополнение к методам, доступным в соответствующих классах, создаваемых при неявной упаковке и распаковке типов. Это допускает своеобразный синтаксис, непривычный для основных компиляторов (хотя, возможно, знакомый тем, кто работал с такими языками, как Java или SmallTalk).

```
var
  S: string;
  I: Integer;
begin
  I := 42;
  S := I.ToString;   // возможен вызов метода класса Integer!
end;
```

НА ЗАМЕТКУ

Тип `String` при разработке используется достаточно часто. Это вовсе не означает, что другие типы применяются редко, но описанию строк посвящена отдельная глава, 11, “Работа с классами String и StringBuilder”, поэтому не будем рассматривать их подробно здесь.

Сравнение типов данных

Delphi поддерживает большинство базовых типов, доступных в среде CLR. Перечень базовых типов языка Delphi и соответствующих им типов C# и CLR приведен в табл. 5.5. Здесь также указано их соответствие спецификации CLS.

Таблица 5.5. Сравнение типов данных Delphi, C# и CLR

Тип переменной	*Delphi*	*C#*	*CLR*	*Соответствие CLS*
8-битовое целое со знаком	`ShortInt`	`sbyte`	`System.SByte`	Нет
8-битовое целое без знака	`Byte`	`byte`	`System.Byte`	Да
16-битовое целое со знаком	`SmallInt`	`short`	`System.Int16`	Да
16-битовое целое без знака	`Word`	`ushort`	`System.UInt16`	Нет
32-битовое целое со знаком	`Integer`	`int`	`System.Int32`	Да
32-битовое целое без знака	`Cardinal`	`uint`	`System.UInt32`	Нет
64-битовое целое со знаком	`Int64`	`long`	`System.Int64`	Да
64-битовое целое без знака	`UInt64`	`ulong`	`System.UInt64`	Нет
4-байтовое вещественное	`Single`	`float`	`System.Single`	Да
8-байтовое вещественное	`Double`	`double`	`System.Double`	Да
Десятичное с фиксированной точкой	Отсутствует	`decimal`	`System.Decimal`	Да
Десятичное с фиксированной точкой Delphi	`Currency`	Отсутствует	Отсутствует	Нет
Дата и время	`TDateTime`*	Отсутствует	`System.DateTime`	Да
Вариант	`Variant`, `OleVariant`	Отсутствует	Отсутствует	Нет
1-байтовый символ	`AnsiChar`	Отсутствует	Отсутствует	Нет
2-байтовый символ	`Char`, `WideChar`	`char`	`System.Char`	Да
Строка фиксированной длины	`ShortString`	Отсутствует	Отсутствует	Нет
Динамическая 1-байтовая строка	`AnsiString`	Отсутствует	Отсутствует	Нет
Динамическая 2-байтовая строка	`String`, `WideString`	`string`	`System.String`	Да
Логический	`Boolean`	`bool`	`System.Boolean`	Отсутствует

* Тип `TDateTime` — это запись (record), являющаяся оболочкой для типа `System.DateTime`, включающей методы и перегруженные операторы. В результате она обладает возможностями, подобными типу `System.DateTime`, а также совместимостью с типом `TDateTime` Delphi Win32.

Символьные типы

Delphi поддерживает три символьных типа данных.

- `WideChar` — соответствующий спецификации CLS 2-байтовый символ Unicode.
- `Char` — псевдоним типа `WideChar`. В версии Delphi для Win32 тип `Char` соответствовал типу `AnsiChar`.
- `AnsiChar` — старый добрый стандартный 1-байтовый символ ANSI.

Не забывайте, что тип `Char` (или любой другой подобный тип) более не является гарантированно однобитовым. Поэтому, чтобы выяснить настоящий размер строки, используйте функцию `SizeOf()`.

НА ЗАМЕТКУ

Стандартная функция `SizeOf()` возвращает размер в байтах переменной любого типа или экземпляра любого класса.

Типы вариантов

Класс `Variant` (вариант) содержит реализацию типа, служащего своего рода контейнером для множества других типов данных. Это значит, что вариант (переменная типа `Variant`) способен во время выполнения изменять свой тип. Например, приведенный ниже код вполне допустим.

```
var
  V: Variant;
  I: IInterface;
begin
  V := 'Delphi is Great!'; // Вариант содержит строку,
  V := 1;                  // теперь вариант содержит целое число,
  V := 123.34;             // теперь число с плавающей запятой,
  V := True;               // теперь логическое значение,
  V := I;                  // а теперь интерфейс.
end;
```

Варианты поддерживают широкое разнообразие типов, включая такие сложные типы данных, как массив и интерфейс. Приведенный ниже код из модуля `Borland.Vcl.Variants` демонстрирует разнообразие поддерживаемых типов, известных под общим названием тип `TVarType`.

```
type
  TVarType = Integer;

const                  // Decimal | Другие названия этого типа
  varEmpty    = $0000; // = 0     | Unassigned, Nil
  varNull     = $0001; // = 1     | Null, System.DBNull
  varSmallInt = $0002; // = 2     | I2, System.Int16
  varInteger  = $0003; // = 3     | I4, System.Int32
  varSingle   = $0004; // = 4     | R4, System.Single
  varDouble   = $0005; // = 5     | R8, System.Double
  varCurrency = $0006; // = 6     | Borland.Delphi.System.Currency
```

```
  varDate     = $0007; // = 7     | Borland.Delphi.System.TDateTime
  varString   = $0008; // = 8     | WideString, System.String
  varError    = $000A; // = 10    | Exception, System.Exception
  varBoolean  = $000B; // = 11    | Bool, System.Boolean
  varObject   = $000C; // = 12    | TObject, System.Object
  varDecimal  = $000E; // = 14    | System.Decimal
  varShortInt = $0010; // = 16    | I1, System.SByte
  varByte     = $0011; // = 17    | U1, System.Byte
  varWord     = $0012; // = 18    | U2, System.UInt16
  varLongWord = $0013; // = 19    | U4, System.UInt32
  varInt64    = $0014; // = 20    | I8, System.Int64
  varUInt64   = $0015; // = 21    | U8, System.UInt64
  varChar     = $0016; // = 22    | WideChar, System.Char
  varDateTime = $0017; // = 23    | System.DateTime;

  varFirst    = varEmpty;
  varLast     = varDateTime;

  varArray    = $2000; // = 8192  | System.Array, Dynamic Arrays
  varTypeMask = $0FFF; // = 4095
  varUndefined = -1;
```

При необходимости (например, во время присвоения) вариант может быть неявно приведен к другому типу данных. Например, так.

```
var
  V: Variant;
  I: Integer;
begin
  V := '1'; // V содержит строку '1'
  I := V;   // Неявное преобразование в Integer. Теперь I содержит 1
end;
```

Преобразование типов вариантов

Преобразование в тип `Variant` можно осуществить явно. Например, так.

```
Variant(X);
```

Теперь результат выражения `X` будет получен в виде варианта, и может представлять собой такой тип данных, как целое число, число с плавающей точкой, денежный тип, строка, символ, дата и время, класс или логический тип.

Вариант можно также преобразовать и в другие типы данных. Давайте рассмотрим пример.

```
V := 1.6;
```

Здесь `V` — переменная типа `Variant`, а следующие выражения представляют различные преобразования типов, возможные для него.

```
S := String(V);  // S содержит строку '1.6'
I := Integer(V); // I округлено до ближайшего целого, т.е. до 2
B := Boolean(V); // B содержит True, поскольку V не равно 0
D := Double(V);  // D содержит значение 1.6
```

Полученные результаты обусловлены правилами преобразования типов, принятыми для типа Variant. Более подробная информация об этих правилах приведена в руководстве *Delphi Language Guide*.

Заметим, что в предыдущем примере явное приведение типов не являлось неизбежным. Показанный ниже код по своим возможностям идентичен предыдущему.

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

Когда в выражении используется несколько вариантов, для неявного приведения их типов компилятор может добавить большое количество вспомогательного кода. В таких случаях, если хранимые в вариантах типы данных известны, для ускорения операций и уменьшения объема кода можно воспользоваться явным преобразованием типов.

Варианты в выражениях

Варианты можно использовать в выражениях со следующими операторами: +, =, *, /, div, mod, shl, shr, and, or, xor, not, :=, <>, <, >, <= и >=. Стандартные функции Inc(), Dec(), Trunc() и Round() также допустимы в выражениях с использованием вариантов.

Когда варианты используются в выражениях, Delphi принимает решение о том, как именно выполнять операторы на основании текущего типа, содержащегося в варианте. Например, если варианты V1 и V2 содержат целые числа, то результатом выражения V1+V2 станет их сумма. Но если они содержат строки, то результатом будет их объединение. А что произойдет, если типы данных различны? В этом случае Delphi использует некоторые правила, позволяющие определить, какие именно преобразования необходимо выполнить. Так, если V1 содержит строку '4.5', а V2 — вещественное число, то V1 будет преобразовано в число 4.5 и добавлено к значению V2. Давайте рассмотрим пример.

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // Строковый тип
  V2 := '50';   // Строковый тип
  V3 := 200;    // Тип Integer
  V1 := V1 + V2 + V3;
end;
```

Исходя из описанных выше правил, можно предположить, что в результате будет получено целое значение 350. Но это не так. Поскольку вычисление выражений выполняется слева направо, то при первом сложении (V1 + V2) складываются две строки, и в результате должна получиться тоже строка, имеющая значение '10050'. А уже затем полученный результат будет преобразован в целое значение и просуммирован с третьим целочисленным операндом, в результате чего будет получено значение 10250.

В Delphi для успешного выполнения вычислений данные типа Variant всегда преобразуются в самый высокий тип данных, присутствующих в выражении. Но если в операции участвуют два варианта, для которых Delphi не в состоянии подобрать под-

ходящий тип, то создается и передается исключение EVariantTypeCast. Вот простейший пример такой ситуации.

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  V1 := V1 / V2;  // Передача исключения.
end;
```

Как уже упоминалось, явное преобразование во время компиляции — это хорошая идея, если участвующие в выражении типы данных известны. Давайте рассмотрим следующую строку кода.

```
V4 := V1 * V2 / V3;
```

Прежде, чем результат такого уравнения будет получен, для каждой операции понадобится выполнить несколько циклов, в ходе которых функции времени выполнения попробуют выяснить типы, совместимые с содержимым вариантов. Затем соответствующие типы данных будут преобразованы. Такой подход приводит к увеличению объема дополнительного кода и служебной информации. Безусловно, наилучшим решением является отказ от применения вариантов, но при необходимости можно также явно осуществить преобразование вариантов, чтобы избежать дополнительных операций во время компиляции.

```
V4 := Integer(V1) * Double(V2) / Integer(V3);
```

Помните: этот подход подразумевает, что типы данных, в которые варианты могут быть успешно преобразованы, известны заранее.

Пустое значение и значение Null

Два значения вариантов заслуживают отдельного обсуждения. Первое, Unassigned, свидетельствует о том, что значение варианту пока не присвоено. Это начальное значение варианта, которое компилятор устанавливает при входе переменной в область видимости. Второе значение, Null, отличается от значения Unassigned тем, что оно представляет реально существующее значение, которое равно Null. Это отличие особенно важно при работе с базами данных, где отсутствие значения и значение Null — абсолютно разные вещи. Более подробная информация о работе с базами данных приведена в части IV, "Применение ADO.NET для разработки баз данных".

Еще одно отличие этих значений заключается в том, что любая попытка выполнения операции с вариантом, содержащим значение Unassigned, приводит к получению значения, преобразовываемого в 0 для числовых операций или пустой строки для строковых операций. Но для вариантов, содержащих значение Null, все происходит иначе. Когда в уравнении встречается вариант, содержащий значение Null, происходит передача исключения EVariantTypeCast.

Если варианту необходимо присвоить значение или сравнить его с ним, то используется один из двух предопределенных в модуле System.Vcl.Variants специальных вариантов — Unassigned и Null, которые вполне применимы для присвоения и сравнения.

Управление поведением вариантов

Присваивая свойству `Variant.NullStrictOperations` значение `True`, можно контролировать передачу исключений при выполнении арифметических операций с вариантами, содержащими значение `Null`. Другие флаги типа `Variant` контролируют поведение вариантов, содержащих значение `Null`, при преобразовании в типы `String` и `Boolean`.

```
NullEqualityRule: TNullCompareRule;
NullMagnitudeRule: TNullCompareRule;
NullAsStringValue: string;
NullStrictConvert: Boolean;
BooleanToStringRule: TBooleanToStringRule;
BooleanTrueAsOrdinalValue: Integer;
```

Пользовательские типы данных

Таких типов, как целые числа, строки и вещественные числа, зачастую недостаточно для адекватного представления данных, с которыми приходится работать при решении реальных задач. В подобных случаях приходится создавать свои собственные типы данных, которые лучше соответствуют текущей задаче. В Delphi пользовательские типы данных обычно принимают форму *записей* (record) или *классов* (class). При объявлении пользовательских типов данных используется ключевое слово `Type`.

Массивы

Язык Delphi позволяет создавать *массивы* (array) переменных любого типа. Например, ниже объявлена переменная, представляющая собой массив из восьми целых чисел.

```
var
  A: Array[0..7] of Integer;
```

Этот оператор эквивалентен следующему объявлению языка C#.

```
int A[8];
```

А также следующему объявлению языка Visual Basic .NET.

```
Dim A(8) as Integer
```

Массивы Delphi обладают специальным свойством, отличающим их от массивов других языков — они не обязаны начинаться с определенного номера элемента. Следовательно, вполне возможно объявлять массив из трех элементов, начинающийся с элемента под номером 28, как в следующем примере.

```
var
  A: Array[28..30] of Integer;
```

Поскольку массивы Delphi не обязаны начинаться с нулевого или первого элемента, при их переборе в цикле `for` следует принимать некоторые меры предосторожности. Для этого компилятор предоставляет две встроенные функции — `High()` и `Low()`, возвращающие верхнюю и нижнюю границы заданного массива. Код станет более ус-

тойчивым к ошибкам и возможным изменениям в описании массива, если в циклах `for` перебора элементов будут использованы эти функции, как показано ниже.

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // Не используйте в циклах
    A[i] := i;                   // жестко заданный код!
end;
```

При определении многомерных массивов размерности в описании отделяются запятой.

```
var
  // Двумерный массив целых чисел:
  A: array[1..2, 1..2] of Integer;
```

Для доступа к элементам такого массива используют индексы, задаваемые через запятую.

```
I := A[1, 2];
```

Динамические массивы

Динамический массив (dynamic array) — это массив, память для которого выделяется динамически. При объявлении динамического массива его размер не указывают.

```
var
  // Динамический массив строк:
  SA: array of String;
```

Прежде, чем динамический массив можно будет использовать, необходимо с помощью процедуры `SetLength()` задать его размер, что приведет к выделению необходимой памяти.

```
begin
  // Выделить в памяти место для 33 элементов:
  SetLength(SA, 33);
```

Как только память будет выделена, с элементами динамического массива можно работать как с элементами любого другого массива.

```
  SA[0] := 'Pooh likes hunny';
  OtherString := SA[0];
```

НА ЗАМЕТКУ

Динамические массивы всегда начинаются с нулевого элемента.

Динамический массив — это тип данных, продолжительностью существования которого управляет среда .NET. Поэтому об освобождении занимаемой им памяти заботиться не нужно, она автоматически освобождается механизмом *сборщика мусора* (garbage collector), когда динамический массив покидает область видимости. Но иногда возникает необходимость удалить динамический массив из памяти прежде, чем он

покинет область видимости (например, если он использует слишком много памяти). Для этого достаточно просто присвоить массиву значение nil.

```
SA := nil;  // Освобождение памяти, выделенной для массива SA
```

Обратите внимание: присвоение массиву SA значения nil освобождает не занятую им память, а только ссылку на массив. Однако, поскольку ссылаться на массив, обозначенный как SA, может несколько переменных, сборщик мусора среды .NET освободит занятую им память только при следующем сборе мусора после освобождения последней ссылки на массив SA.

Для работы с динамическими массивами применяется семантика ссылок, а не значений, как у обычных массивов. Вот маленький тест: чему равен элемент A1[0] после выполнения следующего фрагмента кода?

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

Правильный ответ — 26, поскольку присвоение A2 := A1 не создает новый массив, а приводит к тому, что элементы массива A1 и A2 ссылаются на один и тот же участок памяти. Следовательно, изменение любого элемента массива A2 приведет к изменению соответствующего элемента массива A1 (впрочем, точнее было бы сказать, что это просто один и тот же элемент). Если же необходимо создать именно копию массива A1, то придется воспользоваться стандартной функцией Copy().

```
A2 := Copy(A1);
```

После выполнения этой строки кода массивы A2 и A1 окажутся совершенно независимыми массивами, первоначально содержащими одинаковые данные. Теперь изменения каждого из них не будут влиять на другой. Кроме того, вовсе не обязательно копировать весь массив, функция Copy() позволяет задать номер первого копируемого элемента и их количество, как показано ниже.

```
// Копировать только два элемента, начиная с первого:
A2 := Copy(A1, 1, 2);
```

Динамические массивы тоже могут быть многомерными. Чтобы определить такой массив, добавьте дополнительное описание array of для каждой дополнительной размерности.

```
var
  // Двумерный динамический массив целых чисел:
  IA: array of array of Integer;
```

При выделении памяти для многомерного динамического массива функции SetLength() следует передать дополнительный параметр.

```
begin
  // IA будет массивом целых чисел размерностью 5 x 5
  SetLength(IA, 5, 5);
```

Обращение к элементам многомерного динамического массива ничем не отличается от обращения к элементам обычного массива.

```
IA[0, 3] := 28;
```

Для доступа к многомерному массиву приемлем также синтаксис в стиле C.

```
IA[0][3] := 28;
```

Записи

Пользовательские структуры, определяемые в языке Delphi, называются *записями* (record). Они эквивалентны структурам (`struct`) языка C# и типам (`Type`) языка Visual Basic .NET. Например, определение записи в Delphi эквивалентно следующим определениям языков C# и Visual Basic .NET.

```
{ Delphi }
Type
  MyRec = record
    i: Integer;
    d: Double;
  end;
```

```
/* C# */
public struct MyRec
{
  int i;
  double d;
}
```

```
'Visual Basic
Type MyRec
  i As Integer
  d As Double
End Type
```

При работе с записями для доступа к их полям применяется символ точки (точечный оператор). Например, так.

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Для записей также доступны методы, перегрузка операторов и реализация интерфейсов. В предыдущих версиях компилятора Delphi эти возможности не поддерживались. Более подробная информация по этой теме, а также о типе `class` приведена далее в этой главе. Однако пример синтаксиса применения этих возможностей для записей представлен ниже.

```
IBlah = interface
  procedure bar;
end;
```

```
Foo = record(IBlah)  // Запись реализует интерфейс IBlah
  AField: Integer;
  procedure bar;
  class operator Add(a, b: Foo): Foo;  // Перегрузка оператора +
end;
```

Наборы

Наборы (set) — это уникальный тип данных Delphi, который не имеет эквивалентов в языках C# и Visual Basic .NET. Наборы представляют собой эффективное и удобное средство представления коллекций чисел, символов или других перечислимых значений. Для объявления нового типа набора используется ключевое слово `set of` с указанием перечислимого типа или диапазона допустимых значений. Например, так.

```
type
  TCharSet = set of AnsiChar;   // Допустимы элементы: #0 - #255

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;      // Любая комбинация членов TEnum

  TSubrangeSet = set of 1..10;  // Допустимы элементы: 1 - 10
```

Обратите внимание: набор может содержать не более 256 элементов. Кроме того, в объявлении набора после ключевого слова `set of` можно указывать только перечислимые типы данных. Таким образом, следующие объявления некорректны.

```
type
  TIntSet = set of Integer;     // Слишком много элементов
  TStrSet = set of string;      // Неперечислимый тип данных
```

Внутренне наборы хранят свои элементы как отдельные биты, что делает их весьма эффективными с точки зрения скорости обработки и использования памяти.

НА ЗАМЕТКУ

При переносе кода Win32 на платформу .NET имейте в виду, что символы в мире .NET — 2-байтовые, а в Win32 — 1-байтовые. Это означает, что объявление набора символов `set of Char` будет преобразовано компилятором .NET в набор `AnsiChar`, который потенциально может изменить смысл кода. Поэтому компилятор выдаст предупреждение, рекомендующее использовать объявление набора `AnsiChar` явно (`set of AnsiChar`).

Применение наборов

При присвоении набору значения из одного или нескольких элементов используются квадратные скобки. Приведенный ниже пример демонстрирует объявление переменных типа набора, а также присвоение им значений.

```
type
  TCharSet = set of AnsiChar;      // Допустимы элементы: #0 - #255
  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday,
           Saturday, Sunday);
  TEnumSet = set of TEnum;         // Любая комбинация членов TEnum
var
  CharSet: TCharSet;
```

```
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10;       // Допустимы элементы: 1 - 10
  AlphaSet: set of 'A'..'z';       // Допустимы элементы:'A' -'z'
begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];                  // Пусто, нет элементов
end;
```

Операторы для работы с наборами

Язык Delphi предоставляет несколько операторов для работы с наборами. Эти операторы можно использовать для определения принадлежности к набору, добавления и удаления элементов наборов, а также пересечения наборов.

Принадлежность

Для определения принадлежности элемента к тому или иному множеству используется оператор `in`. Например, приведенный ниже код позволяет удостовериться в принадлежности символа "`S`" набору `CharSet`.

```
if 'S' in CharSet then
  // выполнить действия
```

В следующем примере проверяется отсутствие элемента `Monday` в наборе `EnumSet`.

```
if not (Monday in EnumSet) then
  // выполнить действия;
```

Добавление и удаление элементов

Для добавления и удаления элементов из переменных типа набор используются операторы `+` и `-` или процедуры `Include()` и `Exclude()`.

```
Include(CharSet, 'a');           // добавить в набор 'a'
CharSet := CharSet + ['b'];      // добавить в набор 'b'
Exclude(CharSet, 'x');           // удалить из набора 'z'
CharSet := CharSet - ['y', 'z']; // удалить из набора 'y' и 'z'
```

СОВЕТ

По возможности используйте для добавления и удаления одиночных элементов набора процедуры `Include()` и `Exclude()`, а не операторы `+` и `-`, поскольку первый подход эффективней.

Пересечение

Оператор `*` используется для вычисления пересечения двух множеств. В результате операции `Set1 * Set2` получается набор, содержащий все элементы общие для обоих наборов (`Set1` и `Set2`). Приведенный ниже код позволяет эффективно определять содержание в наборе нескольких заданных элементов.

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  //выполнить действия
```

Небезопасный код

Читатели, знакомые с предыдущими книгами по Delphi для Win32, вероятно, сейчас подумали: "А что же случилось с указателями?". Хотя указатели все еще поддерживаются языком, с точки зрения платформы .NET они считаются небезопасными, поскольку обеспечивают прямой доступ к памяти. Следовательно, чтобы использовать указатели в приложениях, компилятор необходимо уведомить о разрешении работы с *небезопасным кодом* (unsafe code). Для применения небезопасного кода необходимо предпринять следующие действия.

1. В модуль, содержащий небезопасный код, следует включить директиву `{$UNSAFECODE ON}`.
2. Функции с небезопасным кодом следует пометить ключевым словом `unsafe`.

Приведенный ниже небезопасный код успешно компилируется в Delphi.

```
{$UNSAFECODE ON}

procedure RunningWithScissors; unsafe;
var
  A: array[0..31] of Char;
  P: PChar;            // PChar является небезопасным типом
begin
  A := 'safety first'; // заполнить символьный массив
  P := @A[0];          // указатель на первый элемент
  P [0] := 'S';        // изменить первый элемент
  MessageBox.Show(A);  // отобразить измененный массив
end;
```

Обратите внимание: для доступа к адресу расположенных в памяти данных применяется *оператор обращения к адресу* (address of operator) `@`.

СОВЕТ

Небезопасный код абсолютно неприемлем в среде .NET, поскольку небезопасное приложение не подлежит осмотру инструментом .NET PEVerify, а следовательно, может быть запрещено системой безопасности. Однако небезопасный код может оказаться полезен при переходе от Win32 к платформе .NET, ведь достаточно большое приложение Win32 будет весьма сложно перевести на платформу .NET сразу. Таким образом, возможность применения небезопасного кода позволит переводить приложение по частям, оставляя прежний код небезопасным до тех пор, пока все приложение не будет полностью переведено на безопасный код.

Таким образом, код этой части главы следует компилировать с использованием директив `$UNSAFECODE` и `unsafe`.

Указатели

Указатель (pointer) — это переменная, содержащая местоположение (адрес) участка памяти. В этой главе уже встречался указатель, когда рассматривался тип `PChar`.

Встроенным типом указателя языка Delphi является `Pointer`, называемый также *нетипизированным указателем* (untyped pointer), поскольку он содержит только адрес места в памяти, и компилятору ничего не известно о данных, располагающихся по этому адресу. Применение таких "указателей вообще" противоречит концепции строгого контроля типов данных, поэтому работать рекомендуется лишь с указателями на данные конкретного типа.

НА ЗАМЕТКУ

Для нетипизированных указателей в среде .NET используется тип `System.IntPtr`.

Типизированные указатели объявляются в разделе `Type` программы с использованием символа `^` (оператора указателя) или ключевого слова `Pointer`. Эти указатели позволяют компилятору отслеживать, с каким именно типом данных происходят действия, и не выполняются ли при этом некорректные операции. Примеры объявлений типизированных указателей приведены ниже.

```
Type
  PInt = ^Integer;        // PInt - указатель на тип Integer
  Foo = record            // Тип Record
    GobbledyGook: string;
    Snarf: Double;
  end;
  PFoo = ^Foo;            // PFoo - указатель на объект типа foo
var
  P: Pointer;             // Нетипизированный указатель
  P2: PFoo;               // Экземпляр PFoo
```

НА ЗАМЕТКУ

Программисты на языках C/C++ могут заметить схожесть оператора `^` языка Delphi и оператора `*` языков C/C++. Тип `Pointer` языка Delphi соответствует типу `void *` языков C/C++.

Не забывайте, что переменная типа `Pointer` всегда содержит только адрес области памяти. О выделении памяти для структуры, на которую будет ссылаться такой указатель, программист должен позаботиться сам. Предыдущие версии Delphi имели множество функций, позволявших разработчику выделять и освобождать память. Однако теперь, поскольку память на платформе .NET прямо распределяется достаточно редко, оно выполняется только при использовании класса `System.Runtime.InteropServices.Marshal`. Приведенный ниже код демонстрирует применение этого класса для создания и освобождения участка памяти, а также копирования в него данных из массива и обратно.

```
{$UNSAFECODE ON}

type
  TArray = array[0..31] of Char;

procedure ArrayCopy; unsafe;
var
  A1: TArray;
```

```
  A2: array of char;
  P: IntPtr;
begin
  A1 := 'safety first'; // заполнить символьный массив
  SetLength(A2, High(TArray) + 1);
  P := Marshal.AllocHGlobal(High(TArray) + 1);
  try
    Marshal.Copy(A1, 0, P, High(TArray) + 1); // Копировать из A1
                                              // в участок памяти
    Marshal.Copy(P, A2, 0, High(TArray) + 1); // Копировать из
                                              // участка в A2
    MessageBox.Show(A2);       // отобразить измененный массив
  finally
    Marshal.FreeHGlobal(P);
  end;
end;
```

Если необходимо получить доступ к данным, на которые ссылается определенный указатель, то можно воспользоваться оператором ^, следующим за именем этой переменной. Такой метод называется *разрешением указателя* (dereferencing), а также *косвенным доступом*, *взятием значения*, *разыменованием* и *ссылкой*. Ниже приведен пример работы с указателями.

```
procedure PointerFun; unsafe;
var
  I: Integer;
  PI: ^Integer;
begin
  I := 42;
  PI := @I;                    // указывает на I
  PI^ := 24;                   // изменить I
  MessageBox.Show(I.ToString);
end;
```

Компилятор Delphi использует для указателей строгий контроль соответствия типов. Так, в приведенном ниже примере переменные `a` и `b` не совместимы по типу.

```
var
  a: ^Integer;
  b: ^Integer;
```

В эквивалентном описании на языке C эти переменные вполне совместимы.

```
int *a;
int *b
```

Язык Delphi создает уникальный тип для каждого объявления *указателя на тип* (pointer-to-type). Поэтому, если значение переменной `a` предполагается присвоить переменной `b`, то необходимо создать именованный тип, как показано ниже.

```
type
  PtrInteger = ^Integer;  // Создать именованный тип
var
```

```
a: PtrInteger;
b: PtrInteger;   // Теперь a и b совместимы по типу
```

НА ЗАМЕТКУ

Если указатель не ссылается ни на что (его значение равно `0`), то о таком указателе говорят, что его значение равно `Nil`, а сам указатель называют *нулевым* или *пустым* указателем.

Строки с завершающим нулевым символом

Как уже упоминалось в этой главе, Delphi поддерживает три разных типа строк с завершающим нулевым символом: `PChar`, `PAnsiChar` и `PWideChar`. Как и следует из их названия, эти три основных типа строк Delphi завершаются нулем. В среде .NET тип `PChar` является псевдонимом типа `PWideChar`, а в Win32 — псевдонимом `PAnsiChar`. В этой главе каждый строковый тип обобщенно рассматривается как `PChar`. В Delphi тип `PChar` существует главным образом для совместимости с прежними версиями. Тип `PChar` определяется как указатель на строку с завершающим нулевым символом. Поскольку тип `PChar` является неуправляемым, небезопасным указателем, память для него автоматически не выделяется, и в среде .NET он не обслуживается.

В среде Win32 разновидности типа Delphi `PChar` считаются совместимыми с типом `String`. Однако в среде .NET эти типы не совместимы, что делает их применение гораздо менее популярным.

Вариантные записи

Язык Delphi поддерживает также *вариантные записи* (variant record), обеспечивающие хранение разнотипных данных в одной и той же области памяти. Не путайте эти записи с рассмотренным выше типом `Variant` — вариантные записи позволяют независимо получать доступ к каждому из перекрывающихся полей данных. Те, кто знаком с языком C, могут представить себе вариантные записи как аналог концепции `union` в структурах языка C. Приведенный ниже код показывает вариантную запись, в которой поля типа `Double`, `Integer` и `Char` занимают одну и ту же область памяти.

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
    case Integer of
      0: (D: Double);
      1: (I: Integer);
      2: (C: Char);
  end;
```

НА ЗАМЕТКУ

Правила языка Delphi запрещают размещать в вариантной части записи какие-либо типы данных с управляемым временем жизни. Сюда относятся классы, интерфейсы, варианты, динамические массивы и строки.

Ниже приведен эквивалент тех же объявлений типов, но на языке C.

```
struct TUnionStruct
{
  char * StrField;
  int IntField;
  union u
  {
    double D;
    int i;
    char c;
  };
};
```

Поскольку вариантные записи подразумевают прямое распределение памяти, они также считаются небезопасными типами данных.

НА ЗАМЕТКУ

Используя атрибуты `StructLayout` и `FieldOffset`, разработчик может управлять распределением памяти для записей в среде .NET.

Классы и объекты

Класс (class) — это тип данных значения, способный содержать данные, свойства, методы и операторы. Поскольку объектная модель Delphi обсуждается в разделе "Использование объектов Delphi" этой главы, здесь остановимся только на синтаксисе объявления классов. Класс объявляется следующим образом.

```
Type
  TChildObject = class(TParentObject)
  public
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Это объявление эквивалентно следующему объявлению на языке C#.

```
public class TChildObject: TParentObject
{
  public int SomeVar;
  public void SomeProc() {};
}
```

Методы определяются подобно обычным процедурам и функциям (речь о которых пойдет далее в этой главе). Единственное отличие заключается в добавлении перед именем метода имени класса и точки.

```
procedure TChildObject.SomeProc;
begin
  { здесь располагается код процедуры }
end;
```

Символ точки (.) в Delphi аналогичен оператору "." в языках C# и Visual Basic .NET, применяемому при обращении к членам класса.

Псевдонимы типов

Язык Delphi позволяет присваивать новые имена уже существующим типам данных, т.е. создавать их *псевдонимы* (alias). Например, если обычному типу `Integer` необходимо присвоить новое имя `MyReallyNiftyInteger`, то можно использовать следующий код.

```
type
  MyReallyNiftyInteger = Integer;
```

Новый тип аналогичен оригиналу. Это означает, что везде, где использовался тип `Integer`, можно применять тип `MyReallyNiftyInteger`.

Но можно создать и *строго типизированный* (strongly typed) псевдоним (т.е. псевдоним, не совместимый с оригиналом), для чего используется дополнительное ключевое слово `type`.

```
type
  MyOtherNeatInteger = type Integer;
```

При этом новый тип будет преобразовываться в оригинальный при таких операциях, как присвоение, но как параметр он будет несовместим с оригиналом. Следующий код синтаксически корректен.

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

Но код, приведенный ниже, не может быть откомпилирован из-за ошибки совместимости типов.

```
procedure Goon(var Value: Integer);
begin
  // Некоторый код
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Ошибка: тип переменной M не совместим с Integer
```

Помимо усиленного контроля за совместимостью типов данных, для строго типизированных псевдонимов компилятор создает также информацию о *типах времени выполнения* (runtime type). Это позволяет создавать уникальные редакторы свойств для простых типов. Более подробная информация по этой теме приведена в главе 8, “Mono — межплатформенный проект среды .NET”.

Приведение и преобразование типов

Приведение типов (typecasting) — эта технология, способная заставить компилятор рассматривать переменную одного типа как переменную другого типа. По-

скольку Delphi является языком со строгим контролем типов данных, его компилятор весьма требователен в отношении соответствия типов формальных параметров функции и реальных параметров, передаваемых ей при вызове. Поэтому в некоторых случаях переменную одного типа необходимо привести к другому типу, чтобы требования компилятора были удовлетворены. Предположим, необходимо присвоить символьное значение переменной типа `Word`.

```
var
  c: char;
  w: Word;
begin
  c := 's';
  w := c;   // здесь компилятор выдаст сообщение об ошибке
end.
```

Здесь переменную `c` необходимо привести к типу `Word`. Фактически приведение типа указывает компилятору на то, что программист точно знает, что он делает, требуя преобразовать один тип данных в другой.

```
var
  c: Char;
  w: Word;
begin
  c := 's';
  w := Word(c);   // Теперь компилятор доволен
end.
```

НА ЗАМЕТКУ

Использовать приведение типов можно только при совпадении размеров данных обоих типов. Привести тип `Double` к типу `Integer` невозможно, поэтому для подобного преобразования придется воспользоваться функцией `Trunc()` или `Round()`. Но, чтобы преобразовать целое число в вещественное, можно применить обычный оператор присвоения: `FloatVar := IntVar`. Кроме того, для явного или неявного приведения типов можно воспользоваться операторами преобразования, определенными для класса (более подробная информация о перегрузке операторов приведена далее).

Оператор `as` языка Delphi также предоставляет широчайшие возможности приведения типов объектов. Функции оператора `as` идентичны стандартным функциям приведения типов, за исключением того, что при отказе он возвращает *нулевое значение* (`null`), а не передает исключение.

Строковые ресурсы

Используя директиву `resourcestring`, строковые ресурсы можно поместить непосредственно в исходный код Delphi. *Строковые ресурсы* (string resource) — это литеральные строки (как правило, представляющие собой сообщения программы пользователю), которые физически расположены в ресурсных файлах, присоединенных к приложению, или в отдельной библиотеке, а не внедрены в исходный код программы. В частности, подобное отделение строк от исходного кода упрощает перевод приложе-

ния на другой язык (локализацию): для этого достаточно просто присоединить к приложению строковые ресурсы на необходимом языке без перекомпиляции самого приложения. Строковые ресурсы описываются в директиве `resourcestring` в виде пар *идентификатор = строковый литерал*, как показано ниже.

```
resourcestring
  ResString1 = 'Resource string 1';
  ResString2 = 'Resource string 2';
  ResString3 = 'Resource string 3';
```

В исходном коде строковые ресурсы применяются подобно функции, возвращающей строку.

```
resourcestring
  ResString1 = 'hello';
  ResString2 = 'world';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

Условные операторы

В этом разделе операторы `if` и `case` языка Delphi сравниваются с соответствующими конструкциями языков C# и Visual Basic .NET. Предполагается, что читатель уже знаком с подобными операторами, а потому не будем тратить время на подробное объяснение их назначения и функционирования.

Оператор `if`

Оператор `if` (условного перехода) позволяет проверить, выполняется ли некоторое условие, и, в зависимости от результатов этой проверки, выполнить тот или иной блок кода. В качестве примера ниже приведен простейший фрагмент кода с использованием оператора `if` и его аналоги на языках C# и Visual Basic .NET.

```
{ Delphi }
if x = 4 then y := x;

/* C# */
if (x == 4) y = x;

'Visual Basic
If x = 4 Then y = x
```

НА ЗАМЕТКУ

При наличии в операторе `if` нескольких условий поместите каждое из них в скобки. Это сделает код как минимум более ясным и удобочитаемым. Например, так.

```
if (x = 7) and (y = 8) then
```

Избегайте записи, подобной следующей (поскольку компилятор поймет ее неправильно).

```
if x = 7 and y = 8 then
```

В языке Delphi ключевые слова `begin` и `end` (называемые также *логическими скобками*) используются аналогично фигурным скобкам (`{` и `}`) в языке C#. Ниже приведен пример применения подобной конструкции в операторе `if`. Здесь при справедливости условия выполняется несколько строк кода.

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

Используя конструкцию `if..else`, можно объединить проверку нескольких условий и выполнение нескольких блоков кода.

```
if x =100 then
  SomeFunction
else if x = 200 then
  SomeOtherFunction
else begin
  SomethingElse;
  Entirely;
end;
```

Оператор case

Оператор `case` языка Delphi подобен оператору `switch` языка C#. Он позволяет осуществить выбор одного варианта из нескольких возможных без использования сложных конструкций, состоящих из нескольких вложенных операторов `if..else if..else`. Пример оператора `case` приведен ниже.

```
case SomeIntegerVariable of
  101 : DoSomething;
  202 : begin
          DoSomething;
          DoSomethingElse;
        end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```

НА ЗАМЕТКУ

Проверяемая в операторе `case` переменная обязана принадлежать к перечислимому типу. Использование других типов данных, в частности типа `String`, не допускается. Другие языки .NET, например C#, позволяют использовать строки в операторе `switch`.

На языке C# оператор `switch` выглядит следующим образом.

```
switch (SomeIntegerVariable)
{
  case 101: DoSomeThing(); break;
  case 202: DoSomething();
            DoSomethingElse(); break
  case 303: DoAnotherThing(); break;
  default : DoTheDefault();
}
```

Циклы

Цикл (loop) представляет собой конструкцию, которая позволяет выполнять повторяющиеся действия. Циклы языка Delphi во многом подобны циклам других языков, поэтому не будем останавливаться на деталях их построения и работы. В этом разделе описываются все конструкции циклов, существующие в языке Delphi.

Цикл `for`

Цикл `for` идеален для организации повторения некоторых действий заранее определенное количество раз. Ниже приведен код цикла `for` (хоть и не слишком полезного), в котором к значению переменной `X` десять раз прибавляется значение счетчика цикла `I`.

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
    inc(X, I);
end.
```

На языке C# данный пример выглядит так.

```
void main()
{
  int x = 0;
  for(int i=1; i<=10; i++)
    x += i;
}
```

На языке Visual Basic .NET этот пример имеет следующий вид.

```
Dim X As Integer
For I = 1 To 10
  X = X + I
Next I
```

Цикл while

Цикл while позволяет повторять выполнение блока кода до тех пор, пока заданное условие остается истинным. Условие проверяется *перед* выполнением блока кода, поэтому классическим примером использования данной конструкции является выполнение некоторых действий с файлом (например, чтение его данных), продолжающееся до тех пор, пока не будет достигнут его конец. Ниже приведен пример цикла, в котором при каждой итерации одна строка считывается из файла и выводится на экран.

```
program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // текстовый файл
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  try
    while not EOF(f) do begin
      readln(f, S);
      writeln(S);
    end;
  finally
    CloseFile(f);
  end;
end.
```

Цикл while языка Delphi аналогичен циклу while языка C# и циклу Do..While языка Visual Basic .NET.

Цикл repeat..until

Цикл repeat..until выполняет ту же задачу, что и цикл while, но несколько иначе. С его помощью блок кода выполняется до тех пор, пока заданное условие не станет истинным; причем вначале выполняется блок кода, а уже *затем* проверяется истинность условия. Этот цикл аналогичен циклу do..while языка C#.

В приведенном ниже примере значение счетчика X увеличивается до тех пор, пока оно не превысит 100.

```
var
  x: Integer;
begin
  X := 1;
  repeat
    inc(x);
  until x > 100;
end.
```

Оператор Break

Вызов оператора Break внутри цикла while, for или repeat..until организует немедленный выход из цикла. Подобный подход полезен в тех случаях, когда во время выполнения цикла возникает обстоятельство, требующее немедленного его завершения. Этот оператор аналогичен оператору Break языка C# и оператору Exit языка Visual Basic .NET. В следующем примере выполнение цикла прекращается после пяти итераций.

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do begin
    MessageBeep(0);          // подача звукового сигнала
    if i = 5 then Break;
  end;
end;
```

Оператор Continue

Вызов оператора Continue прерывает выполнение тела цикла и осуществляет переход к началу следующей итерации, пропуская при этом все оставшиеся операторы блока. В частности, в приведенном ниже примере при первой итерации второй оператор вывода строки выполнен не будет.

```
var
  i: Integer;
begin
  for i := 1 to 3 do begin
    writeln(i, '. Before continue');  // до Continue
    if i = 1 then Continue;
    writeln(i, '. After continue');   // после Continue
  end;
end;
```

Процедуры и функции

Всем программистам хорошо знакомы основные положения использования процедур и функций. *Процедура* (procedure) представляет собой отдельную часть программы, которая решает определенную задачу, а затем возвращает управление в точку вызова. *Функция* (function) работает практически точно так же, за одним исключением — она возвращает некоторое значение, которое может быть использовано в вызвавшем функцию коде.

В листинге 5.1 приведена небольшая программа, в которой используются процедура и функция.

Листинг 5.1. Пример применения процедуры и функции

```
 1:  program FuncProc;
 2:
```

```
 3:  {$APPTYPE CONSOLE}
 4:
 5:  procedure BiggerThanTen(I: Integer);
 6:  { Выводит на экран сообщение, если I больше 10 }
 7:  begin
 8:    if I > 10 then
 9:      writeln('Funky.');
10:  end;
11:
12:  function IsPositive(I: Integer): Boolean;
13:  { Возвращает True, если I больше или равно 0 }
14:  begin
15:    Result := I >= 0;
16:  end;
17:
18:  var
19:    Num: Integer;
20:  begin
21:    Num := 23;
22:    BiggerThanTen(Num);
23:    if IsPositive(Num) then
24:      writeln(Num, ' Is positive.')
25:    else
26:      writeln(Num, ' Is negative.');
27:  end.
```

Переменная `Result`

Локальная переменная `Result` (результат) функции `IsPositive()` имеет специальное назначение. В каждой функции Delphi существует локальная переменная с этим именем, предназначенная для размещения возвращаемого значения. Обратите внимание: в отличие от языка C#, выполнение функции не прекращается при присвоении значения переменной `Result`.

Если необходимо имитировать поведение оператора `return` языка C#, то непосредственно после присвоения значения переменной `Result` можно вызывать оператор `Exit`. Оператор `Exit` приведет к немедленному выходу из текущей функции.

Вернуть значение из функции можно, присвоив его имени функции внутри кода самой функции. Это стандартный синтаксис языка Delphi, сохранившийся со времен языка Borland Pascal. При использовании в теле функции ее имени будьте внимательны, поскольку существует принципиальное различие между использованием имени функции слева от оператора присвоения и использованием его в любом другом месте кода. Имя функции слева от знака равенства означает присвоение функции возвращаемого значения, а имя функции в любом другом месте внутри кода функции привет к ее рекурсивному[2] вызову!

Имейте в виду, что неявное применение переменной `Result` недопустимо при сброшенном флажке Extended Syntax (Расширенный синтаксис), расположенном во вкладке Compiler (Компилятор) диалогового окна Project Options, либо при указании директивы компилятора `{$EXTENDEDSYNTAX OFF}` (или `{$X-}`).

[2] Рекурсия — вызов себя самого. – *Прим. ред.*

Передача параметров

Язык Delphi позволяет передавать параметры в функции и процедуры либо по значению, либо по ссылке. Передаваемый параметр может иметь любой встроенный или пользовательский тип либо являться открытым массивом (они рассматриваются в этой главе далее). Параметр также может быть константой, если его значение в процедуре или функции не изменяется.

Передача параметров по значению

Передача параметров *по значению* (by value) — это стандартный режим передачи параметров. Когда параметр передается по значению, создается его локальная копия, которая и передоставляется для обработки в процедуру или функцию. Давайте рассмотрим пример.

```
procedure Foo(I: Integer);
```

При вызове процедуры `Foo()` будет создана копия передаваемой ей в качестве параметра целочисленной переменной `I`. Таким образом, процедура будет работать с локальной копией переменной `I`, а внесенные в нее изменения никак не повлияют на значение, переданное процедуре `Foo()`.

Передача параметров по ссылке

Язык Delphi позволяет также передавать параметры в процедуры и функции *по ссылке* (by reference) — такие параметры называются *переменными параметрами* (variable parameter). Передача параметра по ссылке означает, что процедура или функция сможет изменять значения полученных переменных. Для передачи параметров по ссылке используется ключевое слово `var`, помещаемое в список параметров процедуры или функции.

```
procedure ChangeMe(var x: longint);
begin
  x := 2;  { теперь X изменится в вызывающей процедуре }
end;
```

Вместо создания копии переменной `x` ключевое слово `var` потребует передачи адреса самой переменной `x`, что позволит процедуре изменять ее исходное значение.

Для передачи параметров по ссылке в языке C# используется ключевое слово `ref`, а в Visual Basic .NET — директива `ByRef`.

Выходные параметры

Подобно параметрам, объявленным с ключевым словом `var`, параметры, объявленные с ключевым словом `out`, являются средством возвращения функцией значения вызывающей процедуре в виде параметра. Однако в отличие от параметров, объявленных с ключевым словом `var`, которые должны быть инициализированы допустимым значением до вызова функции, параметры, объявленные с ключевым словом `out`, не предполагают проверки правильности передаваемого значения. Для ссылочных типов это означает, что при вызове функции наличие ссылки не имеет никакого значения.

```
procedure ReturnMe(out O: TObject);
begin
  O := SomeObject.Create;
end;
```

Существует простое правило: параметры объявляются как `var`, если они подлежат приему и возвращению, и как `out`, если они подлежат только возвращению.

Параметры-константы

Если необходимости изменять передаваемые процедуре или функции данные нет, то параметр можно объявить как константу, использовав ключевое слово `const`. Ниже приведено объявление процедуры, которой передается постоянный строковый параметр.

```
procedure Goon(const s: string);
```

Параметры в виде открытых массивов

Параметр в виде *открытого массива* (open array) позволяет передавать в процедуру или функцию различное количество аргументов. Передать можно открытый массив элементов одинакового типа или массив констант различного типа. В приведенном ниже примере объявлена функция, которой должен быть передан открытый массив целых чисел.

```
function AddEmUp(A: array of Integer): Integer;
```

В открытом массиве можно передавать переменные, константы или выражения массива любого типа (динамического, статического или входящего открытого). Ниже приведен пример вызова функции `AddEmUp()` с передачей ей нескольких различных элементов.

```
var
  I, Rez: Integer;
const
  J = 23;
begin
  I := 8;
  Rez := AddEmUp([I, I + 50, J, 89]);
```

Массив можно также передать функции непосредственно, как показано ниже.

```
var
  A: array of integer;
begin
  SetLength(A, 10);
  for i := Low(A) to High(A) do
    A[i] := i;
  Rez := AddEmUpConst(A);
```

Для работы с открытым массивом внутри процедуры или функции необходима информация о нем, получить которую можно с помощью функций `High()`, `Low()` и `Length()`. Их применение продемонстрировано ниже, в коде функции `AddEmUp()`, которая возвращает сумму всех переданных ей элементов массива `A`.

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do
    inc(Result, A[i]);
end;
```

Язык Delphi поддерживает также тип `array of const`, который позволяет передавать в одном массиве данные различных типов. Для объявления процедур и функций, использующих в качестве параметра такой массив, применяется следующий синтаксис.

```
procedure WhatHaveIGot(A: array of const);
```

Объявленная выше процедура допускает также следующий синтаксис вызова.

```
WhatHaveIGot(['Tabasco', 90, 5.6, 3.14159, True, 's']);
```

Компилятор передает каждый элемент массива как тип `System.Object`, поэтому при передаче и возвращении проблем не возникает. Пример применения типа `array of const` приведен ниже. Здесь реализована процедура `WhatHaveIGot()`, перебирающая переданный массив и отображающая номер и тип каждого из его элементов.

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
begin
  for i := Low(A) to High(A) do
    WriteLn('Index ', I, ': ', A[i].GetType.FullName);
  ReadLn;
end;
```

Область видимости

Область видимости (scope) — это определенный участок программы, на протяжении которого данная функция или переменная известна компилятору. Глобальные константы видимы в любой точке программы, в то время как локальные переменные видны только в той процедуре, где они были объявлены. Давайте рассмотрим листинг 5.2.

Листинг 5.2. Иллюстрация области видимости

```
 1:  program Foo;
 2:
 3:  {$APPTYPE CONSOLE}
 4:
 5:  const
 6:    SomeConstant = 100;
 7:
 8:  var
 9:    SomeGlobal: Integer;
10:    D: Double;
11:
```

```
12:  procedure SomeProc;
13:  var
14:    D, LocalD: Double;
15:  begin
16:    LocalD := 10.0;
17:    D := D - LocalD;
18:  end;
19:
20:  begin
21:    SomeGlobal := SomeConstant;
22:    D := 4.593;
23:    SomeProc;
24:  end.
```

Здесь переменные `SomeConstant`, `SomeGlobal` и `D` имеют глобальную область видимости, поэтому их значения известны компилятору в любой точке программы. Процедура `SomeProc()` имеет две собственные локальные переменные: `D` и `LocalD`. Любая попытка обратиться к переменной `LocalD` вне процедуры `SomeProc()` приведет к сообщению об ошибке. Обращение к переменной `D` внутри процедуры `SomeProc()` вернет значение ее локального экземпляра, тогда как обращение к переменной `D` вне процедуры вернет значение одноименной глобальной переменной.

Модули и пространства имен

Модули (unit) — это отдельные блоки исходного кода, совокупность которых составляет программу Delphi. Как правило, в модуле размещают группы процедур и функций, которые могут быть вызваны из основной программы. Чтобы считаться модулем, файл с текстом исходного кода должен состоять как минимум из трех приведенных ниже частей.

- Оператор `unit`. Каждый модуль должен начинаться со строки, объявляющей, что данный блок кода является модулем, и задающей имя этого модуля. Имя модуля всегда должно соответствовать имени его файла без расширения. Например, если файл называется `FooBar.pas`, то его первая строка должна выглядеть так.

  ```
  unit FooBar;
  ```

- Раздел интерфейса (`interface`). После оператора `unit` следующей функциональной строкой должен быть оператор `interface`. Все, что находится между этой строкой и оператором `implementation` данного модуля, доступно извне и может использоваться другими модулями и программами. Именно здесь располагают описания типов данных, констант, переменных, процедур и функций, которые должны быть доступны основной программе или другим модулям. В этом разделе допустимы *только объявления* (но не реализация!) процедур и функций. Сам оператор `interface` занимает отдельную строку и содержит единственное ключевое слово.

  ```
  interface
  ```

- Раздел реализации (`implementation`) следует за разделом интерфейса и начинается оператором `implementation`. Хотя основное содержимое этой части модуля

составляют тела описанных ранее процедур и функций, здесь также можно определять типы данных, константы и переменные, которые будут доступны только в пределах данного модуля, но не вне его. Сам оператор `implementation` занимает отдельную строку и содержит единственное ключевое слово.

```
implementation
```

В состав модуля могут входить еще два необязательных раздела.

- Раздел инициализации (`initialization`). Располагается после раздела реализации и содержит код, необходимый для инициализации данного модуля. Этот код будет выполнен только один раз перед началом выполнения основной программы.
- Раздел завершения (`finalization`). Располагается между разделом инициализации и оператором `end.` модуля. Он содержит код, выполняемый только один раз при завершении работы модуля. Раздел `finalization` впервые появился в Delphi 2.0. В Delphi 1.0 для выполнения действий, завершающих работу модуля, следовало создать особую процедуру завершения и зарегистрировать ее с помощью функции `AddExitProc()`. Таким образом, если необходимо перевести приложение Delphi 1.0, то все процедуры выхода модулей следует переместить в их разделы `finalization`.

Модуль определяет также пространство имен. Пространство имен представляет собой логической способ иерархической организации приложения или библиотеки. При создании вложенных пространств имен, позволяющих предотвратить конфликты имен, применяется точечная форма записи. Как правило, для вложенных пространств имен применяется три уровня: *`компания.продукт.пространство`*. Например: `Borland.Delphi.System` или `Borland.Vcl.Controls`.

НА ЗАМЕТКУ

При наличии разделов `initialization` и `finalization` одновременно в нескольких модулях выполнение их кода происходит в том порядке, в котором к этим модулям обращается компилятор (первый модуль в директиве `uses` основной программы, затем первый модуль в директиве `uses` данного модуля и т.д.). Однако не следует создавать код инициализации и завершения модулей, работа которого жестко полагается на этот порядок их выполнения, поскольку малейшее изменение в любом разделе `uses` может привести к ошибке, которую будет чрезвычайно трудно обнаружить!

Директива `uses`

Директива `uses` — это именно то место, где перечисляются пространства имен, включаемые в данную программу или модуль. Например, если в программе `FooProg` используются функции или типы данных из двух пространств имен (`UnitA` и `UnitB`), то ее директива `uses` будет выглядеть следующим образом.

```
program FooProg;
uses UnitA, UnitB;
```

Модули могут содержать две директивы `uses`: одну в разделе `interface`, а другую — в разделе `implementation`.

Вот пример кода типичного модуля.

```
unit FooBar;

interface

uses BarFoo;

   { Объявления открытых элементов (public) }

implementation

uses BarFly;
  { Объявления закрытых элементов (private) }
  { Определение объявленных функций находится в разделе interface }

initialization
  { Код инициализации модуля }
finalization
  { Код завершения модуля }
end.
```

НА ЗАМЕТКУ

Директива `uses` может содержать полностью определенные пространства имен, однако для совместимости с прежними версиями язык Delphi разрешает использовать в директиве `uses` лишь внутреннее имя пространства имен (например, `Controls`). Применение префиксов пространств имен можно задать в диалоговом окне Library (Библиотека), доступном при выборе в меню Tools (Инструменты) пунктов Options (Параметры), Delphi Options (Параметры Delphi), Library (Библиотека).

Взаимные ссылки модулей

Иногда может возникнуть ситуация, когда модуль `UnitA` использует модуль `UnitB`, а тот в свою очередь — модуль `UnitA`. Обычно наличие подобных *взаимных ссылок* (circular unit reference) свидетельствует о просчетах на этапе проектирования структуры приложения. Этого следует избегать. Устранить такую проблему можно, создав третий модуль, в состав которого войдут все необходимые функции и процедуры обоих модулей. Если же по каким-либо соображениям это невозможно (например, модули достаточно велики), то переместите один из разделов `uses` в раздел `implementation` модуля, а другой оставьте в разделе `interface`. Зачастую это решает проблему.

Пакеты и сборки

Пакеты (package) Delphi позволяют размещать части приложения в разных модулях, которые затем могут совместно использоваться несколькими приложениями, подобно сборкам .NET.

Более подробная информация о пакетах и сборках приведена в главе 6, “Сборки — библиотеки и пакеты”.

Объектно-ориентированное программирование

Объектно-ориентированному программированию (Object-Oriented Programming – OOP) посвящены многие тома, в дискуссиях о нем сломано немало копий, и уже начинает казаться, что это не методология программирования, а религия. Не являясь ортодоксами объектно-ориентированного программирования, авторы не будут пытаться обратить читателя в ту или иную веру. Поэтому давайте просто рассмотрим основные принципы OOP, которые положены в основу языка Delphi.

Объектно-ориентированным называется такое программирование, при котором в качестве основных элементов используются отдельные объекты, содержащие и данные, и код для их обработки. Облегчение создания и сопровождения программ не являлось основной целью разработки методологии объектно-ориентированного программирования – это, скорее, побочный эффект ее применения на практике. Кроме того, хранение данных и кода в одном объекте позволяет минимизировать воздействие одного объекта на другой, а следовательно, и облегчить поиск и исправление ошибок в программе. Традиционно объектно-ориентированные языки реализуют по меньшей мере три основные концепции.

- *Инкапсуляция* (enFirstCAPulation). Работа с данными и детали ее реализации скрыты от внешнего пользователя объекта. Достоинства инкапсуляции заключаются в модульности и изоляции кода объекта от другого кода программы.
- *Наследование* (inheritance). Возможность создания новых объектов, обладающих свойствами и поведением родительских объектов. Такая концепция позволяет создавать иерархии объектов (например, VCL), происходящих от одного общего предка и обладающих все большей специализацией и функциональностью по сравнению со своими предшественниками.
- Достоинство наследования, в первую очередь, заключается в совместном использовании общего кода многими объектами. На рис. 5.1 представлен пример наследования: один корневой объект `Fruit` (фрукт) является предком для всех остальных плодов, включая `Melon` (дыня), который, в свою очередь, является предком таких объектов, как `Watermelon` (арбуз) и `Honeydew` (зимняя дыня).

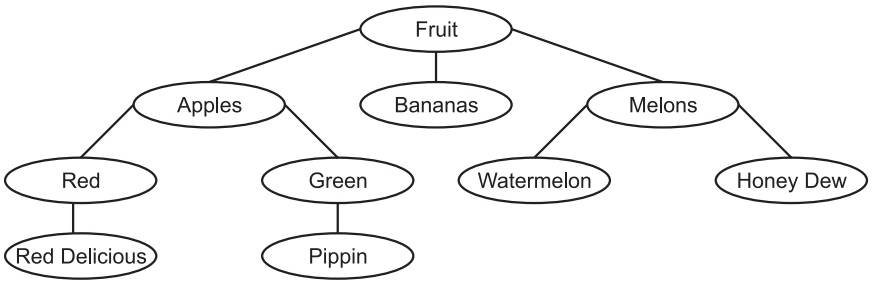

Рис. 5.1. Пример иерархии наследования

- *Полиморфизм* (polymorphism). Слово *полиморфизм* означает "много форм". В данном случае под этим подразумевается, что вызов виртуального метода объекта приведет к исполнению кода конкретного экземпляра объекта.

Обратите внимание: ни среда CLR платформы .NET, ни язык Delphi не поддерживают концепцию множественного наследования языка C++.

Чтобы рассматриваемый далее материал не вызывал затруднений, уточним значение нескольких терминов.

- *Поле* (field), называемое также *определением поля* (field definition) или *переменной экземпляра* (instance variable), представляет собой переменную с данными, содержащуюся внутри объекта. Поле в объекте сходно с полем в записи Delphi. В языке C# для них иногда используется термин *переменные-члены* (data members).
- *Метод* (method). Процедуры и функции, принадлежащие объекту. В языке C# для них используется термин *функция-член* (member function).
- *Свойство* (property) — сущность, предоставляющая доступ к данным и коду, содержащемуся в объекте. Подобно полям свойства скрывают от конечного пользователя детали реализации объекта.

НА ЗАМЕТКУ

В объектно-ориентированном подходе прямой доступ к полям объекта считается обычно плохим стилем программирования, поскольку детали реализации объекта могут со временем измениться. Предпочтительнее работать со свойствами, представляющими собой стандартный интерфейс объекта, скрывающий его конкретную реализацию. Более подробная информация о свойствах приведена в разделе "Свойства" далее в этой главе.

Использование объектов Delphi

Как уже упоминалось, классы — это элементы, способные содержать и данные, и код. *Объекты* (object) — это создаваемые экземпляры классов. Классы Delphi предоставляют разработчику все основные возможности объектно-ориентированного программирования, включая наследование, инкапсуляцию и полиморфизм.

Объявление и создание экземпляра

Безусловно, перед тем, как использовать объект, его следует объявить с помощью ключевого слова `class`. Как уже было сказано, объявления классов располагают в разделе `type` модуля или программы.

```
type
  TFooObject = class;
```

После объявления класса в разделе `var` необходимо объявить переменную или *экземпляр* (instance) класса, описанного в разделе `type`.

```
var
  FooObject: TFooObject;
```

Для создания экземпляра объекта осуществляется вызов одного из его *конструкторов* (constructor). Конструктор отвечает за создание экземпляра объекта, а также за

выделение памяти и инициализацию полей. Он не только создает объект, но и приводит его в состояние, необходимое для дальнейшего использования. Каждый объект содержит по крайней мере один конструктор `Create()`, который может иметь различное количество параметров разного типа. Хоть и не обязательно, но объект может иметь несколько конструкторов. В этой главе рассматривается только простейший конструктор `Create()` (без параметров).

Конструкторы объектов в языке Delphi не вызываются автоматически, разработчик должен организовать вызов конструктора объекта самостоятельно. Синтаксис вызова конструктора имеет следующий вид.

```
FooObject := TFooObject.Create;
```

Обратите внимание на уникальную особенность вызова конструктора. Фактически происходит обращение к конструктору `Create()` класса, а не экземпляра, как у других методов. На первый взгляд это может показаться нелепостью, однако в этом есть глубокий смысл — ведь экземпляр объекта `FooObject` в момент вызова конструктора еще не создан. Но код конструктора класса `TFooObject` уже находится в памяти, а следовательно, обращение к конструктору `Create()` полностью допустимо.

Вызов конструктора для создания экземпляра объекта зачастую называют *созданием экземпляра* (instantiation).

НА ЗАМЕТКУ

После создания конструктором экземпляра объекта среда CLR инициализирует все его поля нулевыми значениями. Все числовые поля инициализируются нулями, объекты — значением `nil`, логические поля — значением `False`, а строки будут пусты.

Уничтожение

Все классы платформы .NET наследуют метод `Finalize()`, переопределив который, можно выполнить любые действия, необходимые для завершения работы экземпляра класса. Сборщик мусора платформы .NET автоматически вызывает метод `Finalize()` для каждого экземпляра. Однако гарантии, что метод `Finalize()` будет фактически вызван и выполнен полностью при любых обстоятельствах, нет. Именно поэтому не рекомендуется освобождать в методе `Finalize()` такие критически важные или объемные ресурсы, как располагающиеся в памяти большие буферы, соединения с базами данных или обработчики операционной системы. Вместо этого, чтобы освободить ценные ресурсы, разработчики Delphi должны использовать *схему расположения* (dispose pattern), переопределяя *деструктор* (destructor) `Destroy()`. Более подробная информация по этой теме приведена в главе 9, “Управление памятью и сбор мусора”.

Адам всех объектов

У начинающих разработчиков может возникнуть вопрос: откуда все эти методы у небольшого объекта, ведь явно их никто не объявлял? Так и есть. Фактически обсуждаемые здесь методы принадлежат базовому классу платформы .NET `System.Object`. Класс `TObject` языка Delphi является псевдонимом класса `System.Object` платформы .NET.

Таким образом, все объекты являются потомками класса `TObject`, независимо от того, объявлен он или нет.

```
type
  TFoo = class
  end;
```

Таким образом, это объявление эквивалентно приведенному ниже.

```
type
  TFoo = class(TObject)
  end;
```

Поля

Добавление *полей* (field) в класс осуществляется с помощью синтаксиса, очень похожего на объявление переменных в блоке `var`. Например, код, приведенный ниже, добавляет в класс `TFoo` поля типа `Integer`, `String` и `Double`.

```
type
  TFoo = class(TObject)
    I: Integer;
    S: string;
    D: Double;
  end;
```

Язык Delphi поддерживает также *статические поля* (static field), т.е. поля, данные которых совместно используются всеми экземплярами этого класса. В объявление класса нужно добавить один или несколько блоков `class var`. Следующий код иллюстрирует добавление в класс `TFoo` трех статических полей.

```
type
  TFoo = class(TObject)
    I: Integer;
    S: string;
    D: Double;
    class var
      I_Static: Integer;
      S_Static: string;
      D_Static: Double;
  end;
```

Учтите: блок `var` вполне можно (хотя синтаксически и не нужно) поместить в определении класса там, где определяются обычные поля.

По своему назначению блок `class var` идентичен ключевому слову `static` языка C#. Обратите внимание: блоки `class var` или `var` завершаются любым из следующих элементов.

- Следующий блок `class var` или `var`.
- Объявление свойства.
- Объявление метода.
- *Спецификатор видимости* (visibility specifier).

Методы

Методы (method) представляют собой процедуры и функции, принадлежащие объекту. Можно сказать, что методы определяют поведение объекта. Выше были рассмотрены два важнейших метода объектов: конструктор и деструктор. Для решения конкретных задач можно самостоятельно создать любое количество других методов.

Создание метода осуществляется в два этапа. Сначала метод объявляется в объявлении типа класса, а затем в коде создается его реализация. Процесс объявления и определения метода продемонстрирован ниже.

```
type
  TBoogieNights = class
    Dance: Boolean;
    procedure DoTheHustle;
  end;

procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

Отметим, что при определении тела метода необходимо использовать его полное имя с указанием имен класса и метода. Вторая важная деталь: метод способен непосредственно обратиться к *любому* полю объекта. Таким образом, к полю объекта `Dance` можно обратиться изнутри метода непосредственно.

Типы методов

В классе могут быть объявлены методы как нормального типа, так и методы класса (`class static`, `class virtual`), статические (`static`), виртуальные (`virtual`), динамические (`dinamic`) или методы обработки сообщения (`message`). Давайте рассмотрим следующий пример объявления класса.

```
TFoo = class
  procedure IAmNormal;
  class procedure IAmAClassMethod;
  class procedure IAmAStatic; static;
  procedure IAmAVirtual; virtual;
  class procedure IAmAVirtualClassMethod; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message WM_SOMEMESSAGE;
end;
```

Обычные методы

В приведенном выше примере метод `IAmNormal()` — это *обычный метод* (regular method). Это стандартный тип метода, который работает аналогично обычной процедуре или функции. Адрес такого метода известен уже на стадии компиляции, и компилятор в коде программы оформляет все вызовы данного метода как статические. Такие методы работают быстрее других, однако они не допускают переопределения, обеспечивающего полиморфизм.

Методы класса

В приведенном выше примере метод `IAmAClassMethod()` является специфическим для Delphi типом статического *метода класса* (class method). Методы класса можно вызывать без экземпляра, а их реализация является общей для всех экземпляров данного класса. Но методы класса обладают специальным, неявным и скрытым параметром `Self`, который передается компилятором, обеспечивая вызов полиморфных (виртуальных) методов класса. Методы класса могут быть *простыми* (plain) или *виртуальными* (virtual). Из метода класса нельзя обращаться к *нестатическим* (nonstatic) элементам и элементам, *не принадлежащим классу* (nonclass).

Статические методы

В приведенном выше примере метод `IAMStatic()` — это настоящий *статический метод* (static method), совместимый с платформой .NET. Подобно статическим полям, реализация статических методов является общей для всех экземпляров данного класса. Из статического метода также нельзя обращаться к нестатическим элементам. Поскольку параметром `Self` статические методы не обладают, из них нельзя вызывать нестатические методы.

Виртуальные методы

В приведенном выше примере метод `IAmAVirtual()` является *виртуальным методом* (virtual method). Вызов виртуальных методов осуществляется подобно статическим, однако в связи с возможностью переопределения во время компиляции, адрес конкретного вызываемого метода не известен. Для решения этой задачи JIT-компилятор платформы .NET создает *таблицу виртуальных методов* (Virtual Method Table — VMT), которая позволяет найти адрес функции во время выполнения. Обращения ко всем виртуальным методам передаются через таблицу VMT. Таблица VMT объекта содержит информацию о виртуальных методах, объявленных как в его собственном, так и в базовом классе.

Динамические методы

В приведенном выше примере метод `IAmADynamic()` является *динамическим методом* (dynamic method). Компилятор платформы .NET соотносит динамические методы с виртуальными, в отличие от компилятора Win32, который для их обработки обладает специальным механизмом.

Методы обработки сообщений

В приведенном выше примере метод `IAmAMessage()` является *методом обработки сообщений* (message-handling method). Ключевое слово `message` позволяет создать метод, отвечающий за динамическую передачу сообщений. Расположенное после ключевого слова `message` значение указывает, какое именно сообщение обрабатывает данный метод. В библиотеке VCL методы сообщений используются для автоматической реакции на сообщения Windows. В приложении методы обработки сообщений никогда не вызываются непосредственно.

Переопределение методов

Переопределение (overriding) метода реализует в языке Delphi концепцию полиморфизма объектно-ориентированного программирования. Оно позволяет изменять поведение метода от потомка к потомку. Переопределены могут быть только те методы Delphi, которые ранее были объявлены виртуальными (`virtual`), динамическими (`dynamic`) или обработки сообщения (`message`). Для переопределения виртуального или динамического метода в разделе `type` производного класса используется директива `override` на месте ключевого слова `virtual` или `dynamic` базового класса. Для переопределения метода обработки сообщения следует повторить директиву `message` с тем же самым идентификатором сообщения, что и в базовом классе. Например, чтобы переопределить методы `IAmAVirtual()`, `IAmADynamic()` и `IAmAMessage()`, можно применить следующий код.

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message WM_SOMEMESSAGE;
end;
```

Директива `override` приводит к замещению строки описания исходного метода в таблице VMT строкой описания нового метода. При повторном объявлении методов `IAmAVirtual` и `IAmADynamic` с ключевым словом `virtual` или `dynamic` вместо ключевого слова `override` можно создать *новые* методы, а не переопределять методы базового класса. Как правило, это приводит к передаче компилятором предупреждения. Чтобы подавить передачу предупреждения, достаточно добавить к объявлению метода директиву `reintroduce`. Кроме того, при переопределении в производном классе обычного метода он полностью скроет от пользователей класса прежний метод.

Перегрузка методов

Подобно обычным процедурам и функциям методы могут быть *перегружены* (overloaded) таким образом, чтобы класс содержал несколько методов с одним именем, но с различными списками параметров. Перегруженные методы объявляются с директивой `overload`, хотя использовать эту директиву для первого экземпляра перегруженного метода в иерархии класса необязательно. Пример кода класса, содержащего три перегруженных метода, приведен ниже.

```
type
  TSomeClass = class
    procedure AMethod(I: Integer); overload;
    procedure AMethod(S: string); overload;
    procedure AMethod(D: Double); overload;
  end;
```

Дублирование имен методов

Иногда может понадобиться добавить в класс метод, заменяющий виртуальный метод с тем же именем базового класса. В данном случае требуется не переопределить метод базового класса, а полностью его заменить. Если просто добавить такой метод в новый класс, то компилятор выдаст предупреждение о том, что новый метод скрывает ме-

тод базового класса с тем же именем. Для подавления этого предупреждения в новом методе следует использовать директиву `reintroduce`. Пример применения директивы `reintroduce` приведен ниже.

```
type
  TSomeBase = class
    procedure Cooper; virtual;
  end;

  TSomeClass = class
    procedure Cooper; reintroduce;
  end;
```

Переменная `Self`

Неявная переменная `Self` доступна во всех методах объектов. Фактически `Self` — это ссылка на экземпляр класса, используемого при вызове метода. Переменная `Self` передается методу компилятором в качестве скрытого параметра. Она аналогична переменным `this` языка C# и `Me` языка Visual Basic .NET.

Ссылки на класс

Обычные переменные типа класса содержат ссылку на сам объект, а *ссылки на класс* (class reference) — ссылку на его тип. Используя ссылку на класс, можно обратиться к классу, к статическим методам класса, а также создать экземпляр класса. Приведенный ниже код иллюстрирует синтаксис объявления нового класса по имени `SomeClass`, а также применение ссылки на класс `SomeClass`.

```
type
  SomeClass = class
    constructor Create; virtual;
    class procedure Foo;
  end;

  SomeClassRef = class of SomeClass;
```

Теперь, используя ссылку на класс `SomeClassRef`, метод класса `SomeClass.Foo()` можно вызвать следующим образом.

```
var
  SCRef: SomeClassRef;
begin
  SCRef := SomeClass;
  SCRef.Foo;
```

Экземпляр класса `SomeClass` также может быть создан с помощью ссылки на класс.

```
var
  SCRef: SomeClassRef;
  SC: SomeClass;
begin
  SCRef := SomeClass;
  SC := SCRef.Create;
```

Обратите внимание: создание класса с помощью ссылки на класс требует, чтобы класс имел по крайней мере один виртуальный конструктор. *Виртуальный конструктор* (virtual constructor) — это уникальная возможность языка Delphi, позволяющая создавать экземпляры классов по ссылкам на класс, тип которого не известен во время компиляции.

Свойства

Свойство (property) — это специализированное средство доступа класса, позволяющее изменять данные его полей и выполнять код его методов. По отношению к компонентам свойства являются теми элементами, сведения о которых отображаются в окне инспектора объектов. Ниже приведен простой пример, иллюстрирующий определение свойств класса.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Класс `TMyObject` представляет собой объект, содержащий одно поле — целое по имени `SomeValue`, один метод — процедуру `SetSomeValue()`, а также одно свойство по имени `Value`. Назначение процедуры `SetSomeValue()` заключается в присвоении полю `SomeValue` некоторого значения. Свойство `Value` не содержит данных — оно используется в качестве средства доступа к полю `SomeValue`. При попытке присвоить свойству `Value` некоторое значение вызывается процедура `SetSomeValue()`, предназначенная для изменения значения поля `SomeValue`. Эта технология обладает несколькими преимуществами. Во-первых, она позволяет разработчику избежать побочных эффектов, возникающих при непосредственном возвращении и установке значений свойств (например, повторного вычисления уравнений или перерисовки элементов управления). Во-вторых, она создает для пользователя класса некий интерфейс, полностью скрывающий от него реализацию класса и обеспечивающий контроль за доступом к его полям. И наконец, она позволяет переопределять методы доступа в производных классах, что обеспечивает полиморфизм поведения.

Подобно статическим полям и методам язык Delphi поддерживает также статические свойства, использующие ключевое слово `class`. Приведенный ниже код демонстрирует класс со статическим свойством, обращающимся к статическому полю.

```
TMyClass = class(TObject)
  class var FValue: Integer;
  class procedure SetValue(Value: Integer); static;
  class property Value: Integer read FValue write SetValue;
end;
```

Обратите внимание на то, что в качестве *средств установки и получения значений* (getter и setter) статические свойства способны использовать только статические поля и методы.

События

Язык Delphi поддерживает два разных вида *событий* (event): *одиночные* (singleton) и *многоадресатные* (multicast). Одиночные события были в языке Delphi и ранее. Они объявлялись как свойства процедурного типа с указанием метода доступа `read` и `write`. Одиночные события могут иметь один *обработчик события* (event listener) или не иметь его. Для назначения событию обработчика используется оператор присвоения, а для удаления обработчика ему достаточно присвоить значение `nil`. Пример объявления и применения одиночного события приведен в листинге 5.3.

Листинг 5.3. Пример одиночного события

```
 1:  program singleevent;
 2:
 3:  {$APPTYPE CONSOLE}
 4:
 5:  type
 6:    TMyEvent = procedure (Sender: TObject; Msg: string) of object;
 7:
 8:    TClassWithEvent = class
 9:    private
10:      FAnEvent: TMyEvent;
11:    public
12:      procedure FireEvent;
13:      property AnEvent: TMyEvent read FAnEvent write FAnEvent;
14:    end;
15:
16:    TListener = class
17:      procedure EventHandler(Sender: TObject; Msg: string);
18:    end;
19:
20:  { TClassWithEvent }
21:
22:  procedure TClassWithEvent.FireEvent;
23:  begin
24:    if Assigned(FAnEvent) then
25:      FAnEvent(Self, '*singleton event*');
26:  end;
27:
28:  { TListener }
29:
30:  procedure TListener.EventHandler(Sender: TObject; Msg: string);
31:  begin
32:    WriteLn('Event was fired. Message is: ', Msg);
33:  end;
34:
```

```
35:  var
36:    L: TListener;
37:    CWE: TClassWithEvent;
38:  begin
39:    L := TListener.Create;          // создать объекты
40:    CWE := TClassWithEvent.Create;
41:    CWE.AnEvent := L.EventHandler;  // присвоить обработчик события
42:    CWE.FireEvent;                  // вызвать событие
43:    CWE.AnEvent := nil;             // отключить обработчик события
44:    ReadLn;
45:  end.
```

Ниже представлен результат выполнения программы листинга 5.3.

```
Event was fired. Message is: *singleton event*
```

Многоадресатные события были введены в язык Delphi для того, чтобы обеспечить присущую платформе .NET возможность существования нескольких обработчиков одного события. Многоадресатные события объявляются как свойства процедурного типа с указанием обоих методов доступа, `read` и `write`. Многоадресатные события могут иметь любое количество обработчиков. Для добавления и удаления обработчиков многоадресатного события используются процедуры `Include()` и `Exclude()`.

Пример объявления и применения многоадресатного события приведен в листинге 5.4.

Листинг 5.4. Пример многоадресатного события

```
 1:  program multievent;
 2:
 3:  {$APPTYPE CONSOLE}
 4:
 5:  uses
 6:    SysUtils;
 7:
 8:  type
 9:    TMyEvent = procedure (Sender: TObject; Msg: string) of object;
10:
11:    TClassWithEvent = class
12:    private
13:      FAnEvent: TMyEvent;
14:    public
15:      procedure FireEvent;
16:      property AnEvent: TMyEvent add FAnEvent remove FAnEvent;
17:    end;
18:
19:    TListener = class
20:      procedure EventHandler(Sender: TObject; Msg: string);
21:    end;
22:
23:  { TClassWithEvent }
24:
```

```
25:  procedure TClassWithEvent.FireEvent;
26:  begin
27:    if Assigned(FAnEvent) then
28:      FAnEvent(Self, '*multicast event*');
29:  end;
30:
31:  { TListener }
32:
33:  procedure TListener.EventHandler(Sender: TObject; Msg: string);
34:  begin
35:    WriteLn('Event was fired. Message is: ', Msg);
36:  end;
37:
38:  var
39:    L1, L2: TListener;
40:    CWE: TClassWithEvent;
41:  begin
42:    L1 := TListener.Create;                 // создать объекты
43:    L2 := TListener.Create;
44:    CWE := TClassWithEvent.Create;
45:    Include(CWE.AnEvent, L1.EventHandler); //присвоить обработчик события
46:    Include(CWE.AnEvent, L2.EventHandler); //присвоить обработчик события
47:    CWE.FireEvent;                         //вызвать событие
48:    Exclude(CWE.AnEvent, L1.EventHandler); //отключить обработчик события
49:    Exclude(CWE.AnEvent, L2.EventHandler); //отключить обработчик события
50:    ReadLn;
51:  end
```

Ниже представлен результат выполнения программы листинга 5.4.

```
Event was fired. Message is: *multicast event*
Event was fired. Message is: *multicast event*
```

Обратите внимание: повторное применение процедуры `Include()` для добавления того же самого метода в список обработчиков приведет к многократному вызову этого метода.

Для обеспечения совместимости с другими языками .NET CLR компилятор Delphi реализует многоадресатную семантику даже для одиночных событий, позволяя создавать, добавлять и удалять методы для одиночного события. В данной реализации применение метода `add()` приведет к перезаписи существующего значения.

Спецификаторы видимости

Язык Delphi обеспечивает полный контроль над поведением объектов, позволяя объявить их поля и методы с помощью таких директив, как `private`, `strict private`, `protected`, `strict protected`, `public` и `published`. Синтаксис применения этих директив имеет следующий вид.

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
```

```
strict private
  procedure AStrictPrivateMethod;
protected
  procedure AProtectedProcedure;
  function ProtectMe: Byte;
strict protected
  procedure AStrictProtectedMethod;
public
  constructor APublicContructor;
  destructor Destroy; override;  // открытый деструктор
published
  property AProperty: Integer
  read APrivateVariable write APrivateVariable;
end;
```

За каждой из директив может следовать любое необходимое количество объявлений полей или методов. Требования хорошего тона предполагают наличие перед спецификатором отступа, аналогичного применяемому для имен классов. Эти директивы имеют следующее назначение.

- `Private` (закрытый). Объявленные здесь переменные и методы доступны только для того кода, который находится в модуле, реализующем объект. Директива `private` скрывает особенности реализации объекта от пользователей и защищает его от непосредственного доступа и изменения извне.
- `Strict private` (строго закрытый). Объявленные здесь члены доступны внутри только того класса, в котором они объявлены, а не внутри того же самого модуля. Используется для более строгой инкапсуляции данных, чем позволяет директива `private`.
- `Protected` (защищенный). Защищенные члены доступны в классах, производных от данного. Это позволяет скрыть внутреннее устройство класса от пользователя и в то же время обеспечить необходимую гибкость, а также эффективность доступа к полям и методам класса для производных от него.
- `Strict protected` (строго защищенный). Объявленные здесь члены доступны внутри только того класса, в котором они объявлены, и производных от него, но не внутри модуля, в котором объявлен класс. Используется для более строгой инкапсуляции данных, чем позволяет директива `protected`.
- `Public` (открытый). Объявленные здесь поля и методы доступны в любом месте программы. Конструкторы и деструкторы всегда должны быть объявлены как `public`.
- `Published` (публикуемый). С точки зрения видимости аналогичен `public`. Однако обладает дополнительным преимуществом, возможностью добавления к содержащимся внутри свойствам атрибута `[Browsable(true)]`, который позволяет отображать их в инспекторе объектов конструктора форм. Более подробная информация об атрибутах приведена далее в этой главе.

НА ЗАМЕТКУ

Смысл публикуемых данных довольно существенно изменился со времен реализации языка Delphi для Win32. В версии Win32 для публикуемых свойств создавалась *информация о типах времени выполнения* (Runtime Type Information — RTTI). В среде .NET эквивалентом RTTI является *рефлексия* (reflection), однако рефлексия возможна для всех членов класса, независимо от их спецификатора видимости.

Ниже показано объявление приведенного ранее класса `TMyObject`, которое дополнено директивами, повышающими его целостность.

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
      SomeValue := AValue;
end;
```

Теперь пользователи этого класса не смогут изменить значение поля `SomeValue` непосредственно и вынуждены будут использовать только интерфейс, предоставляемый свойством `Value`.

"Дружественные" классы

В языке C++ реализована концепция *дружественных классов* (friend class), т.е. классов, которым разрешен доступ к закрытым данным и функциям другого класса. В языке C++ этот механизм реализуется при помощи ключевого слова `friend`. Такие языки .NET, как Delphi и C#, обладают подобной концепцией, хотя реализована она иначе. Члены класса, объявленные не как строго закрытые и защищенные, доступны для других классов и кода, объявленного внутри того же самого пространства имен модуля.

Вспомогательные классы

Вспомогательные классы (class helper) являются средством, позволяющим дополнить класс, фактически не изменяя его. Вместо этого создается новый вспомогательный класс (`helper`), а его методы переходят к исходному классу. Это позволяет пользователям исходного класса вызывать методы вспомогательного класса так, как будто они являются методами исходного.

В приведенном ниже примере кода создается простой и вспомогательный классы, а затем демонстрируется вызов метода вспомогательного класса.

```
program Helpers;

{$APPTYPE CONSOLE}

type
  TFoo = class
    procedure AProc;
  end;

  TFooHelper = class helper for TFoo
    procedure AHelperProc;
  end;

{ TFoo }

procedure TFoo.AProc;
begin
  WriteLn('TFoo.AProc');
end;

{ TFooHelper }

procedure TFooHelper.AHelperProc;
begin
  WriteLn('TFooHelper.AHelperProc');
  AProc;
end;

var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  Foo.AHelperProc;
end.
```

ВНИМАНИЕ!

Применение вспомогательных классов — это весьма интересная возможность, однако вовсе не та, которой следует пользоваться в хорошем проекте приложения. Корпорация *Borland* добавила эту возможность для того, чтобы сгладить различия между классами стандарта .NET и аналогичными классами Delphi Win32. Случай, когда в хорошо проработанном проекте может возникнуть реальная необходимость применения вспомогательных классов, практически исключен.

Вложенные типы данных

Язык Delphi допускает присутствие директивы `type` в объявлении класса, позволяя вложить объявление типа внутрь класса. Для обращения к таким *вложенным типам* (nesting type) данных используется синтаксис *ВнешнийТип*.*ВложенныйТип*, как показано в следующем примере кода.

```
type
  OutClass = class
    procedure SomeProc;
    type
      InClass = class
        procedure SomeOtherProc;
      end;
  end;
var
  IC: OutClass.InClass;
```

Перегрузка операторов

Язык Delphi обеспечивает *перегрузку* (overloading) отдельных операторов для классов и записей. Синтаксис перегрузки оператора столь же прост, как и объявление метода класса с собственным именем и сигнатурой. Полный список перегружаемых операторов приведен в интерактивной справочной системе Delphi в разделе "*Operator Overloads*", однако следующий пример кода демонстрирует перегрузку операторов класса + и -.

```
OverloadsOps = class
  private
    FField: Integer;
  public
    class operator Add(a, b: OverloadsOps): OverloadsOps;
    class operator Subtract(a, b: OverloadsOps): OverloadsOps;
  end;

class operator OverloadsOps.Add(a, b: OverloadsOps): OverloadsOps;
begin
  Result := OverloadsOps.Create;
  Result.FField := a.FField + b.FField;
end;

class operator OverloadsOps.Subtract(a, b: OverloadsOps): OverloadsOps;
begin
  Result := OverloadsOps.Create;
  Result.FField := a.FField - b.FField;
end;
```

Обратите внимание: перегруженные операторы объявлены как `class operator`, а сам класс им передается в качестве параметра. Поскольку операторы + и - являются *бинарными* (binary), возвращают они также объявленный класс.

Перегруженные операторы применяются подобно обычным, как показано ниже.

```
var
  O1, O2, O3: OverloadsOps;
begin
  O1 := OverloadsOps.Create;
  O2 := OverloadsOps.Create;
  O3 := O1 + O2;
end;
```

Атрибуты

Одной из наиболее интересных возможностей платформы .NET является реализация разработки, ориентированной на *атрибуты* (attribute), что позволяет унифицировать несколько языков программирования, существенно отличавшихся в прошлом. Атрибуты предоставляют средства, позволяющие увязать метаданные с такими элементами языка, как классы, свойства, методы, переменные и тому подобное, которые предоставляют дополнительную информацию об этих элементах их потребителям.

Атрибуты объявляют в квадратных скобках перед аргументируемым элементом. Приведенный ниже код демонстрирует применение атрибута `DllImport`, уведомляющего среду .NET о том, что метод должен быть импортирован из указанной библиотеки DLL.

```
[DllImport('user32.dll')]
function MessageBeep(uType : LongWord) : Boolean; external;
```

В среде .NET атрибуты используются достаточно часто. Например, примененный к свойству атрибут `Browsable` обеспечивает его отображение в инспекторе объектов.

```
[Browsable(True)]
property Foo: string read FFoo write FFoo;
```

Поскольку атрибуты реализованы как классы, система атрибутов среды .NET допускает расширение. Фактически это позволяет расширять систему атрибутов как угодно, поскольку собственные атрибуты можно создавать как с самого начала, так и наследуя существующие классы атрибутов, а эти новые атрибуты можно использовать в других классах.

Интерфейсы

Язык Delphi обладает встроенной поддержкой *интерфейсов* (interface), которая существенно упрощает определение набора процедур и функций, применяющихся при взаимодействии приложения и объекта. Определение конкретного интерфейса известно как разработчику, так и его пользователю. Оно воспринимается как соглашение о правилах объявления и использования интерфейса. В классе может быть реализовано несколько интерфейсов. В результате объект становится "многоликим", являя клиенту каждого интерфейса особое "лицо".

Как и следует из его названия, интерфейс — это лишь общее определение того, как объект и его клиент взаимодействуют друг с другом, а реализация всех процедур и функций интерфейса — это уже задача класса, поддерживающего данный интерфейс.

НА ЗАМЕТКУ

В отличие от Delphi Win32, интерфейсы .NET не происходят неявно от интерфейсов `IInterface` или `IUnknown`. Кроме того, они больше не реализуют методы `QueryInterface()`, `_AddRef` и `_Release()`. Приведение типов теперь касается лишь идентификаторов типа, а подсчет ссылок встроен в платформу .NET.

Определение интерфейсов

Синтаксис определения интерфейса очень похож на синтаксис определения класса. Основное различие между ними заключается в том, что интерфейс может быть (но необязательно) связан с *глобальным уникальным идентификатором* (Globally Unique Identifier — GUID), который является уникальным для каждого интерфейса. Приведенный ниже код определяет новый интерфейс по имени `IFoo`, в котором объявлен метод по имени `F1()`.

```
type
  IFoo = interface
    function F1: Integer;
  end;
```

Обратите внимание: GUID необязателен для определений интерфейса .NET, в отличие от интерфейса Win32. Следовательно, применение GUID рекомендуется только тогда, когда необходима многоплатформенная поддержка кода, или предполагается использование технологии .NET COM Interop для обеспечения совместимости между .NET и COM.

СОВЕТ

Интегрированная среда разработки Delphi создает новый GUID при нажатии комбинации клавиш <Ctrl+Shift+G>.

Следующий код определяет новый интерфейс `IBar`, производный от `IFoo`.

```
type
  IBar = interface(IFoo)
    function F2: Integer;
  end;
```

Реализация интерфейса

Приведенный ниже фрагмент кода демонстрирует реализацию интерфейсов `IFoo` и `IBar` в классе `TFooBar`.

```
type
  TFooBar = class(TObject, IFoo, IBar)
    function F1: Integer;
    function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
  Result := 0;
end;
```

Обратите внимание: в первой строке объявления класса после указания базового класса может быть перечислено несколько интерфейсов. Связывание функции интерфейса с определенной функцией класса осуществляется компилятором, когда он обнаруживает совпадение сигнатуры метода класса с сигнатурой метода интерфейса. Класс должен содержать собственные объявления и реализации всех методов интерфейса, в противном случае компилятор выдаст соответствующее сообщение об ошибке, и программа не будет откомпилирована.

Упрощенное создание методов интерфейса

Давайте сделаем небольшое отступление. Интерфейсы — это очень хорошо, но вводить вручную весь код, необходимый для объявления и реализации их методов в классе, крайне затруднительно! В IDE существует прием, позволяющий реализовать все методы интерфейса при помощи нескольких щелчков мыши и нажатий нескольких клавиш.

1. Добавьте подлежащие реализации интерфейсы в объявление класса.
2. Поместите курсор в область класса и нажмите комбинацию клавиш <Ctrl+пробел>, чтобы запустить автоматическое завершение кода. Теперь в окне автоматического завершения кода нереализованные методы окрасятся красным.
3. Удерживая клавишу <Shift>, выделите в списке при помощи мыши или клавиш курсора все методы, окрашенные красным.
4. Нажмите клавишу <Enter>, и методы интерфейса будут автоматически добавлены в определение класса.
5. Нажмите комбинацию клавиш <Ctrl+Shift+C>, чтобы частично завершить реализацию новых методов.
6. Теперь осталось только набрать основную часть реализации каждого метода!

Если класс реализует несколько интерфейсов, которые имеют методы с одинаковыми сигнатурами, то избежать неоднозначности позволит назначение методам *псевдонимов* (alias), как показано в следующем примере.

```
type
  IFoo = interface
    function F1: Integer;
  end;

  IBar = interface
    function F1: Integer;
  end;

  TFooBar = class(TObject, IFoo, IBar)
    // Назначение методам псевдонимов
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
    // Методы интерфейса
    function FooF1: Integer;
    function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;
```

```
function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```

НА ЗАМЕТКУ

Директива `implements` языка Delphi Win32 в текущей версии компилятора Delphi .NET не доступна.

Использование интерфейсов

При использовании переменных типа интерфейса (экземпляров) в приложениях следует помнить о некоторых правилах. Подобно другим типам .NET экземпляры интерфейсов являются объектами с управляемой продолжительностью существования. Сборщик мусора освобождает занимаемую объектом память сразу, как только будут освобождены все ссылки на него, а следовательно, реализованные в нем интерфейсы будут удалены или выйдут из области видимости. Перед использованием тип интерфейса всегда инициализируется значением `nil`. Присвоение интерфейсу значения `nil` вручную приведет к освобождению ссылки на реализующий его объект.

Еще одно правило, уникальное для переменных типа интерфейса: с точки зрения присвоения интерфейс совместим с объектом, реализующим данный интерфейс. Но это односторонняя совместимость: ссылку на объект можно присвоить ссылке на интерфейс, но не наоборот. В приведенном ниже примере используется класс `TFooBar`, определенный ранее.

```
procedure Test(FB: TFooBar)
var
  F: IFoo;
begin
  F := FB;  // Корректно, поскольку FB совместим с IFoo
  .
  .
  .
```

Если бы объект `FB` не был совместим с интерфейсом `IFoo`, то код все равно был бы откомпилирован, но ссылке на интерфейс было бы присвоено значение `nil`. Любая последующая попытка использования этой ссылки во время выполнения привела бы к передаче исключения `NullReferencedException`.

И наконец, для обращения к переменной интерфейса из другого интерфейса того же самого объекта применяются оператор преобразования типов `as`, как показано в следующем примере.

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create;
```

```
F := FB;  // орректно, поскольку FB совместим с IFoo
B := F as IBar;  // приведение для IBar
.
.
.
```

Если запрошенный интерфейс не поддерживается, то выражение возвратит значение `nil`.

Структурная обработка исключений

Структурная обработка исключений (Structured Exception Handling – SEH) представляет собой метод централизованной и стандартизованной обработки ошибок, позволяющий корректно реагировать в коде на несущественные ошибки, а также элегантно обрабатывать ошибки, возникшие в результате практически любых причин. SEH языка Delphi приведена в соответствие с аналогом среды CLR .NET.

Исключения (exception) – это очень простые классы, содержащие информацию о месте происхождения и характере определенной ошибки. Это делает исключения столь же простыми в реализации и использовании, как и любой другой класс.

Платформа .NET обладает обширным набором предопределенных исключений, предназначенных для обработки множества стандартных ошибок в программе, таких как нехватка памяти, деление на нуль, переполнение или недополнение числа, а также ошибок чтения и записи в файл. Корпорация *Borland* предоставила также большое количество классов исключений внутри библиотек Delphi RTL и VCL. И, кроме того, ничто не мешает разработчику определять свои собственные классы исключений и применять их в создаваемых приложениях.

В листинге 5.5 приведен пример обработки исключений, передаваемых при ошибках файловых операций чтения и записи.

Листинг 5.5. Обработка исключений файловых операций

```
 1:  program FileIO;
 2:
 3:  {$APPTYPE CONSOLE}
 4:
 5:  uses System.IO;
 6:
 7:  var
 8:    F: TextFile;
 9:    S: string;
10:  begin
11:    AssignFile(F, 'FOO.TXT');
12:    try
13:      Reset(F);
14:      try
15:        ReadLn(F, S);
16:        WriteLn(S);
17:      finally
18:        CloseFile(F);
19:      end;
```

```
20:    except
21:      on System.IO.IOException do
22:        WriteLn('Error Accessing File!');
23:    end;
24:    ReadLn;
25:  end
```

В листинге 5.5 внутренний блок `try..finally` используется для гарантии того, что файл будет закрыт в любом случае, т.е. независимо от того, допущена ошибка или нет. Иначе говоря, это звучит так: "Программа, попытайся выполнить код между операторами `try` и `finally`. Независимо от того, возникла проблема или нет, необходимо выполнить код между операторами `finally` и `end`. Но, если проблема все же возникла, необходимо обратиться к блоку обработки исключений". Таким образом, файл будет закрыт гарантированно, а ошибка обработана правильно, независимо от типа.

НА ЗАМЕТКУ

Операторы после ключевого слова `finally` в блоке `try..finally` выполняются независимо от того, было передано исключение в процессе выполнения этого блока или нет. Создавая подобный код, убедитесь, что он не зависит от того, передавалось исключение или нет. Кроме того, в связи с тем, что оператор `finally` не прекращает продвижение исключения, выполнение программы возобновится в расположенном далее по тексту программы блоке обработки исключения.

Внешний блок `try..except` используется для обработки исключения, которое может быть передано. После закрытия файла во внутреннем блоке `finally` блок `except` выводит на консоль сообщение об ошибке чтения и записи в файл (если она произошла).

Одно из главных преимуществ системы обработки исключений по сравнению с традиционными методами обработки ошибок заключается в отделении кода обнаружения ошибки от кода ее обработки. Это позволяет сделать код более удобным для чтения и понимания, а также сосредоточиться на отдельном аспекте работы программы.

Безусловно, блок `try..finally` не сможет перехватить абсолютно все исключения. Помещая в программу блок `try..finally`, программист заботится не только об обнаружении некоторого конкретного исключения. Основная цель заключается в том, чтобы выполнить все необходимые действия для корректного выхода из *любой* нештатной ситуации, которая может произойти при выполнении этого фрагмента программы. Блок `finally` является идеальным местом для освобождения любых распределенных ресурсов (например, файлов или ресурсов Windows), поскольку он выполняется даже при возникновении ошибки. Тем не менее во многих ситуациях обработка ошибок может быть связана с выполнением определенных действий, зависящих от типа возникшей ошибки. Для этого и предназначен блок обработки исключений `try..except`, пример применения которого приведен в листинге 5.6.

Листинг 5.6. Блок обработки исключений `try..except`

```
1:  program HandleIt;
2:
3:  {$APPTYPE CONSOLE}
4:
```

```
 5:  var
 6:    D1, D2, D3: Double;
 7:  begin
 8:    try
 9:      Write('Enter a decimal number: ');
10:      ReadLn(D1);
11:      Write('Enter another decimal number: ');
12:      ReadLn(D2);
13:      Writeln('I will now divide the first number by the second...');
14:      D3 := D1 / D2;
15:      Writeln('The answer is: ', D3:5:2);
16:    except
17:      on System.OverflowException do
18:        Writeln('Overflow in performing division!');
19:      on System.DivideByZeroException do
20:        Writeln('You cannot divide by zero!');
21:      on Borland.Delphi.System.EInvalidInput do
22:        Writeln('That is not a valid number!');
23:    end;
24:  end
```

Помимо перехвата конкретных исключений, в блоке `try..except`, используя конструкцию `else`, можно организовать обработку всех остальных исключений. Синтаксис конструкции `try..except..else` приведен ниже.

```
try
  операторы
except
  On ESomeException выполняемые действия; // Обработчик конкретного
                                          // исключения
else
  { Стандартный обработчик для всех остальных исключений }
end;
```

ВНИМАНИЕ!

При использовании конструкции `try..except..else` в блоке `else` следует гарантировать обработку *всех* исключений, в том числе и неожиданных, например исключений, причиной передачи которых являются ошибки выделения памяти или другие ошибки времени выполнения. Поэтому при использовании ключевого слова `else` проявляйте осторожность, ведь в случае некорректной обработки исключения возникнет еще одно исключение. Более подробная информация по этой теме приведена далее, в разделе “Повторная передача исключения”.

Перехват всех типов исключений можно осуществить и при помощи конструкции `try..except`, но без указания класса исключения. Например, так.

```
try
  операторы
except
  ОбработкаИсключения     // Аналог оператора else
end;
```

Классы исключений

Исключения (exception) представляют собой экземпляры специальных объектов. Они создаются при передаче исключения и уничтожаются после их обработки. Базовым классом для всех исключений является класс .NET `System.Exception`.

Важнейшим элементом объекта `Exception` является свойство `Message`, имеющее строковый тип. Оно содержит дополнительную информацию или разъяснение причин передачи исключения. Содержащаяся в свойстве `Message` информация зависит от типа переданного исключения.

ВНИМАНИЕ!

Определяя собственный класс исключения, обязательно объявляйте его как производный от уже существующего класса исключения `Exception` или любого производного от него. Это позволит гарантировать перехват нового объекта исключения стандартным обработчиком.

При обработке конкретного типа исключения в блоке `except` его обработчик будет также перехватывать и все остальные производные исключения. Например, если исключение `System.ArithmeticException` является базовым для нескольких типов исключений, в том числе для `DivideByZeroException`, `NotFiniteNumberException` и `OverflowException`, то все они будут перехвачены приведенным ниже обработчиком для базового исключения `ArithmeticException`.

```
try
  операторы
except
  on ArithmeticException do // Перехват исключения класса
                            // ArithmeticException и всех
                            // производных от него
    ОбработкаИсключения
end;
```

Любое исключение, которые не будет перехвачено и обработано в приложении явно, продолжит освобождать программный стек до тех пор, пока в конечном счете не будет перехвачено *стандартным обработчиком исключений* (default exception handler). В приложениях форм Windows и Web-форм среды .NET, стандартный обработчик исключений предпримет действия, позволяющие отобразить на экране сообщение об ошибке. В приложениях VCL стандартный обработчик отображает диалоговое окно, содержащее сообщение о переданном исключении.

При обработке исключения может потребоваться доступ к самому экземпляру объекта исключения — например, чтобы получить дополнительную информацию о случившемся, содержащуюся в свойстве `Message`. Для этого есть два пути: рекомендованный подход заключается в использовании необязательного идентификатора в конструкции `on ESomeException`. Можно также воспользоваться функцией `ExceptObject()`, но это не рекомендуется.

Добавив необязательный идентификатор в конструкцию `on ESomeException` блока `except`, можно получить идентификатор текущего объекта исключения. Синтаксис применения необязательного идентификатора заключается в указании самого идентификатора и двоеточия непосредственно *перед* типом исключения:

```
try
  операторы
except
  on E:ESomeException do
    ShowMessage(E.Message);
end;
```

Идентификатор (в данном случае `E`) получает ссылку на переданный объект исключения. Этот идентификатор всегда имеет тот же тип, что и породившее его исключение.

Синтаксис передачи исключения подобен синтаксису создания экземпляра объекта. Например, чтобы передать пользовательское исключение по имени `EBadStuff`, можно воспользоваться следующим синтаксисом.

```
raise EBadStuff.Create('Some bad stuff happened.');
```

Процесс обработки исключений

После создания и передачи объекта исключения нормальный ход выполнения программы прерывается, и управление начинает передаваться от одного обработчика исключений к другому до тех пор, пока исключение не будет обработано, а экземпляр объекта исключения уничтожен. Этот процесс построен на обработке стека вызовов и, следовательно, имеет глобальный характер в пределах всей программы, а не только в рамках текущей процедуры или модуля. В листинге 5.7 приведен модуль VCL, иллюстрирующий процесс выполнения программы при передаче исключения. Здесь представлен главный модуль приложения Delphi, содержащего одну форму с единственной кнопкой. Если щелкнуть на кнопке, то метод `Button1Click()` (обработки этого события) вызовет процедуру `Proc1()`, которая, в свою очередь, вызовет процедуру `Proc2()`, вызывающую процедуру `Proc3()`. Исключение передается именно в этой, последней, наиболее глубоко вложенной процедуре `Proc3()`, что позволяет проследить весь процесс прохождения исключения через каждый из блоков `try..finally` до тех пор, пока оно не будет обработано внутри метода `Button1Click()`.

СОВЕТ

Если запустить эту программу из интегрированной среды разработки Delphi, то проследить процесс обработки исключения будет проще, если предварительно отключить встроенный обработчик исключений отладчика. Для этого следует сбросить флажок Tools⇨Options⇨Debugger Options⇨Borland .NET Debugger⇨Language Exceptions⇨Stop on Language Exceptions.

Листинг 5.7. Демонстрация передачи и обработки исключений

```
 1:  unit Main;
 2:
 3:  interface
 4:
 5:  uses
 6:    Windows, Messages, SysUtils, Variants, Classes, Graphics,
 7:    Controls, Forms, Dialogs;
 8:
```

```
 9:  type
10:    TForm1 = class(TForm)
11:      Button1: TButton;
12:      procedure Button1Click(Sender: TObject);
13:    end;
14:
15:  var
16:    Form1: TForm1;
17:
18:  implementation
19:
20:  {$R *.nfm}
21:
22:  type
23:    EBadStuff = class(Exception);
24:
25:  procedure Proc3;
26:  begin
27:    try
28:      raise EBadStuff.Create('Up the stack we go!');
29:    finally
30:      ShowMessage('Exception raised. Proc3 sees the exception');
31:    end;
32:  end;
33:
34:  procedure Proc2;
35:  begin
36:    try
37:      Proc3;
38:    finally
39:      ShowMessage('Proc2 sees the exception');
40:    end;
41:  end;
42:
43:  procedure Proc1;
44:  begin
45:    try
46:      Proc2;
47:    finally
48:      ShowMessage('Proc1 sees the exception');
49:    end;
50:  end;
51:
52:  procedure TForm1.Button1Click(Sender: TObject);
53:  const
54:    ExceptMsg = 'Exception handled in calling procedure. The
↳message is "%s"';
55:  begin
56:    ShowMessage('This method calls Proc1 which calls Proc2 which
↳calls Proc3');
```

```
57:    try
58:      Proc1;
59:    except
60:      on E:EBadStuff do
61:       ShowMessage(Format(ExceptMsg, [E.Message]));
62:    end;
63:  end;
64:
65:  end.
```

Повторная передача исключения

Если во внутреннем блоке `try..except` создается пользовательский обработчик исключения, выполняющий специальные действия, но не прекращающий дальнейшую передачу исключения вплоть до стандартного обработчика, можно воспользоваться технологией *повторной передачи исключения* (reraising the exception). Пример повторной передачи исключения приведен в листинге 5.8.

Листинг 5.8. Повторная передача исключения

```
 1:  try              // внешний блок
 2:    { операторы }
 3:    { операторы }
 4:    { операторы }
 5:    try            // специальный внутренний блок
 6:      { действия, требующие специальной обработки исключения }
 7:    except
 8:      on ESomeException do
 9:      begin
10:        { специальный внутренний обработчик исключения }
11:        raise;  // повторная передача исключения во внешний блок
12:      end;
13:    end;
14:  except
15:    // внешний блок со стандартным обработчиком
16:    on ESomeException do Something;
17:  end;
```