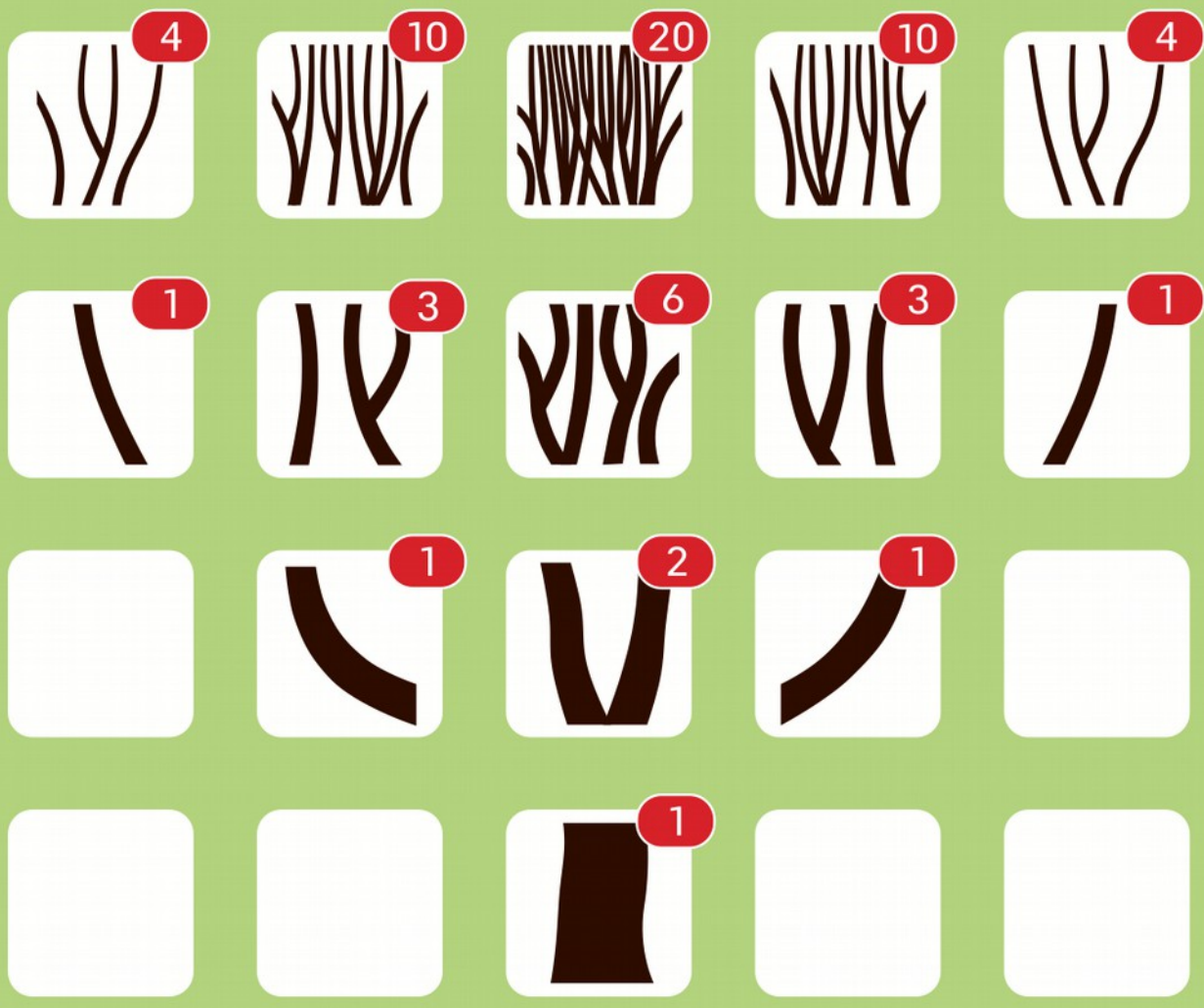


MARCO CANTÙ

OBJECT PASCAL HANDBOOK

Marco Cantù

Object Pascal Handbook
October 2014 Draft

A first draft of the "*Complete Guide to the Object Pascal programming language*",
based on the compilers produced by Embarcadero Technologies

Piacenza (Italy), October 2014

Author: Marco Cantù

Publisher: Marco Cantù

Editor: Peter W A Wood

Cover Designer: Fabrizio Schiavi (www.fsd.it)

The Object Pascal Handbook, October 2014 Draft.

ISBN-10: *

ISBN-13: *

More information on ***http://www.marcocantu.com/objectpascal***

begin

Power and simplicity, expressiveness and readability, great for learning and for professional development alike, these are some of the traits of today's Object Pascal, a language with a long history, a lively present, and a brilliant future ahead.

Object Pascal is a multi-facet language. It combines the power of object-oriented programming, advanced support for generic programming and dynamic constructs like attributes, but without removing support for more traditional style of procedural programming. A tool for all trades, with compilers and development tools embracing the mobile era. A language ready for the future, but with solid roots in the past.

What is the Object Pascal language for? From writing desktop apps to client-server applications, from massive web server modules to middleware, from office automation to apps for the latest phones and tablets, from industrial automated systems to Internet virtual phone networks... this is not what the language could be used for, but what it is *currently* used for today, in the real world.

The core of the Object Pascal language as we use today comes from its definition in 1995, a terrific year for programming languages, given that this was the year Java and JavaScript were also invented. While the root of the language dates back to its Pascal ancestor, its evolution didn't stop in 1995, with core enhancements continuing as of today, with the desktop and mobile compilers build by Embarcadero Technologies and found in Appmethod, Delphi, and RAD Studio.

A Book on Todays' Language

Given the changing role of the language, its extension over the years, and the fact it is now attracting new developers, I felt it important to write a book that offers complete coverage of the Object Pascal language as it is today. The goal is to offer a language manual for new developers, for developers coming from other similar languages, but also for old timers of different Pascal dialects that want to learn more about recent language changes.

Newcomers certainly need some of the foundations, but given changes have been pervasive even old-timers will find something new in the initial chapters.

Beside a short Appendix covering the short history of the Object Pascal language, this book was written to cover the language as it is today. A large part of the core features of the language hasn't changed significantly since the early versions of the Delphi, the first implementation of modern Object Pascal in 1995.

As I'll hint throughout the book, the language has been far from stagnant during all of these years, it has been evolving at quite a fast pace. In other books I wrote in the past, I followed a more *chronological* approach, covering classic Pascal first, and following extensions more or less as they appeared over time. In this book, however, the idea is to use a more *logical* approach, progressing through the topics and covering how the language works today, and how to best use it, rather than how it evolved over time. As an example, native data types dating back to the original Pascal language have method-like capabilities (thanks to intrinsic type helpers) introduced recently. So in Chapter 2 I'll introduce how to use this feature, although it won't be until much later than you'll figure out how to make such custom type extensions.

In other words, this book covers the Object Pascal language how it is today, teaching it from the ground up, with only a very limited historical perspective. Even if you have used the language in the past, you might want to skim thought the entire text looking for newer features, and not focus only on the final chapters.

Learn by Doing

The idea of the book is to explain core concepts and immediately present short demos that readers are encouraged to try to execute, experiment with, and extend to understand the concepts and assimilate them better. The book is not a reference manual, explaining what the language should do in theory and listing all possible corner cases. While trying to be precise, the focus is more on teaching the language offering a practical step-by-step guide. Examples are generally very simple, because the goal is to have them focused on one feature at a time.

The entire source code is available in a subversion repository, rather than a download file, so that you can easily update your code in case I publish any changes or additional demos. You can use any subversion client (my personal preference on Windows is for TortoiseSVN) and point it to the following HTTP URL to get all of the demos of the book (alternatively you can also check out individual chapters):

```
http://code.marcocantu.com/svn/marcocantu_objectpascalhandbook/
```

The repository source code can also be browsed online by selecting the “Browse” link in the code repository wiki page at:

```
http://code.marcocantu.com/trac/marcocantu_objectpascalhandbook
```

To compile and test it the demo code, you'll need one of the available versions of the Object Pascal compiler, and possibly a recent one to compile them all. There are trial versions available that you can use, generally allowing you a 30-days free use of the compiler. Appendix C explains how to get started with the available IDEs.

A Companion Web Site

The book has a companion web site with further information, links, updates and more. There are both a static, traditional site with information at:

```
http://www.marcocantu.com/objectpascalhandbook
```

and online pages on Google+ (using a `bit.ly` link) and Facebook at

```
http://bit.ly/objectpascalgplus
https://www.facebook.com/objectpascalhandbook
```

Acknowledgments

As any book, this volumes owes a lot to many people, too many to list one by one. The person who shared most of the work on the book was my editor, Peter Wood, who kept bearing with my ever changing schedule and was able to smoothen my technical English very significantly as usual, helping to make this book (like my previous handbooks) what it is.

Given my current work position as product manager at Embarcadero Technologies, I owe a lot to all my my coworkers and the members of the R&D team, as during my time at the company my understanding of the product and its technology has further increased thanks to the insight I got in countless conversations, meetings, and email threads.

Other people outside Embarcadero continued being important for this, from the current group at Wintech Italia, to the countless customers, Embarcadero sales and technical partners, Delphi community members, MVPs and even developers using other languages and tools I keep meeting so often.

And finally big thank you goes to my family for bearing with my travel schedule, nights of meetings, plus some extra book writing on weekends. Thanks Lella, Benedetta, and Jacopo.

About Myself, the Author

My name is Marco and I've spent most part of the past 20 years writing, teaching, and consulting on software development with the Object Pascal language. I wrote the Mastering Delphi best-selling series and later self-published several Handbooks on the development tool (about the different versions from Delphi 2007 to Delphi XE). I have spoken at a large number of programming conferences in most continents, and taught to hundreds of developers.

Having worked as an independent consultant and trainer for many years, in 2013 my career took a sudden change: I accepted a position as Delphi and now RAD Studio product manager at Embarcadero Technologies, the company that builds and sells these great development tools, along with the new Appmethod product.

To avoid annoying you any further, I'll only add that I currently live in Italy, commute to California, have a lovely wife and two wonderful kids, and enjoy getting back to programming as much as I can.

I hope you enjoy reading the book, as much as I enjoyed writing it (my 19th work in print). For any further information, use any of the following contact details:

```
http://www.marcocantu.com
http://blog.marcocantu.com
http://twitter.com/marcocantu
https://www.google.com/+MarcoCantu
http://www.facebook.com/marcocantu
```

table of contents

part i: foundations

Object Pascal is an extremely powerful language based upon core foundations such as a good program structure and extensible data types.

In this first part, you'll lean about the language syntax, the structure of programs, the use of variables and data types, the fundamental language statements (like conditions and loops), the use of procedures and functions, and core type constructors such as arrays, records, and strings.

These are the foundations of the more advanced features, from classes to generic types, that we'll explore in later parts of the book. Learning a language is like building a house, and you need to start on solid ground and good foundations, or everything else up and above would be shining... but shaky.

Summary of Part I

01: coding in pascal

This chapter starts with some of the building blocks of an Object Pascal application, covering standard ways of writing code and related comments, introducing keywords, and the structure of a program. I'll start writing some simple applications, trying to explain what they do and thus introducing some other key concepts covered in more details in the coming chapters.

Let's Start with Code

This chapter covers the foundation of the language, but it will take me a few chapters to guide you through the details of a complete working application. So for now let's have a look at two first programs (different in their structure), without really getting into too many details. Here I just want to show you the structure of programs that I'll use to build demos explaining specific language constructs before I'll be able to

cover all of the various elements. Given that I want you to be able to start putting in practice the information in the book as soon as possible, looking at demo examples from the beginning would be a good idea.

note If are new to programming in Object Pascal and need a step-by-step guide to get started with using the book examples and writing your own, you can refer to Appendix C.

Object Pascal has been designed to work hand-in-glove with its Integrated Development Environment. It is through this powerful combination that Object Pascal can match the ease of development speed of programmer-friendly languages and at the same time match the running speed of machine-friendly languages.

The IDE lets you design user interfaces, help you write code, run your programs and much, much more. I'll be using the IDE throughout this book as I introduce Object Pascal to you.

A First Console Application

As a starting point, I'm going to show you the code of a simple *Hello, World* console application showing some of the structural elements of a Pascal program. A console application is a program with no graphical user interface, displaying text and accepting keyboard input and generally executed from an operating system console or command prompt. Console apps generally make little sense on mobile platforms, and are seldom used on PCs these days.

I won't explain what the elements on the code below mean just yet, as that is the purpose of the first few chapters of the book. Here is the code, from the `HelloConsole` project:

```
program HelloConsole;

{$APPTYPE CONSOLE}

var
  strMessage: string;

begin
  strMessage := 'Hello, World';
  writeln (strMessage);
  // wait until the Enter key is pressed
  readln;
end.
```

note As explained in the introduction, the complete source code of all of the demos covered in the book is available in a subversion repository. Refer to the book introduction for more details on how to get those demos. In the text I refer to the project name (in this case `HelloConsole`), which is also the name of the folder containing the various files of the demo. The project folders are grouped by chapter, so you'll find this first demo under `01/HelloConsole`.

You can see the program name in the first line after a specific declaration, a compiler directive (prefixed by the `$` symbol and enclosed in curly braces), a variable declaration (a string with a given name), and three lines of code plus a comment within the main `begin-end` block. Those three lines of code copy a value into the string, call a system function to write that line of text to the console, and call another system function to read a line of user input (or in this case to wait until the user pressed the Enter key). As we'll see, you can define your own functions, but Object Pascal comes with hundreds of pre-defined ones.

Again, we'll learn about all of these elements soon, as this initial section serves only to give you an idea of what a small but complete Pascal program looks like. Of course you can open and run this application, which will produce output like the following (the actual Windows version is displayed in Figure 1.1).

```
Hello, World
```

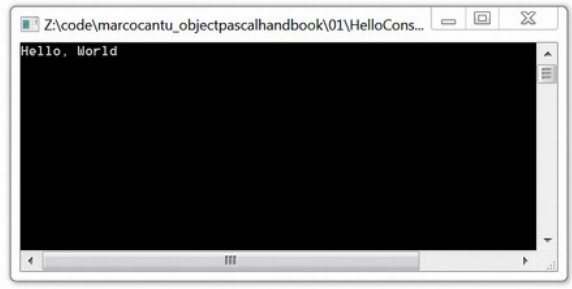


Figure 1.1: The output of the HelloConsole example, running on Windows

A First Visual Application

A modern application, though, rarely looks like this old fashioned console program, but is generally made of visual elements (referred to as controls) displayed in windows (referred to as forms). In most cases in this book I'll build visual demos (even if in most cases they'll boil down to displaying simple text) using the FM platform library.

note In case of Delphi, the visual controls come in two *flavors*: the VCL (Visual Component Library for Windows) and the FM platform (a multi-device library for all supported platforms, desktop and mobile). On the other hand Appmethod includes only the FM platform for multi-device development. In any case, it should be rather simple to adapt the demos to the Windows-specific VCL library.

To understand the exact structure of a visual application, you'll have to read a good part of this book, as a form is an object of a given class and has methods, event handlers, and properties... all features that will take a while to go through. But to be able to create these applications, you don't need to be an expert, as all you have to do is use a menu command to create a new desktop or mobile application. What I'll do in the initial part of the book is to base the demos on the FM platform (supported by both IDEs) and simply use the context of forms and button click operations. To get started, you can create a form of any type (desktop or mobile, I'd generally pick a mobile "blank" application, as it will also run on Windows), and place a button control on it, after a multi line text control (or Memo) to display the output. Figure 1.2 shows how your form will look for a mobile application in the IDE, given the default settings. You can refer to Appendix C for step by step instructions on how to build this demo.

What you have to do to create a similar application is to add a button to an empty mobile form. Now to add the actual code, which is the only thing we are interested in for now, double click on the button, you'll be presented with the following code skeleton (or something very similar):

```
procedure TForm1.Button1Click (Sender: Tobject)
begin

end;
```

Even if you don't know what a method of a class is (which is what `Button1Click` is), you can type something in that code fragment (that means within the `begin` and `end` keywords) and that code will execute when you press the button.

Our first "visual" program has code matching that of the first console application, only in a different context and calling a different library function, namely `ShowMessage`. This is the code you can find in the HelloVisual demo and you can try rebuilding it from scratch quite easily:

```
procedure TForm1.Button1Click (Sender: Tobject)
var
  strMessage: string;
begin
  strMessage := 'Hello, World';
  ShowMessage (strMessage);
end;
```

Figure 1.2:
A simple mobile application with a single button, used by the HelloVisual demo

Notice how you need to place the declaration of the `strMessage` variable before the begin statement and the actual code after it. Again, don't worry if things are not clear, everything will get explained in due time and in great detail.

note You can find the source code of this demo in a folder under the 01 container for the chapter. In this case, however, there is a project file name like the demo but also a secondary unit file with the word "Form" added after the project name. That's the standard I'm going to follow in the book. The structure of a project is covered at the end of this chapter.

In Figure 1.3 you can see the output of this simple program, running on Windows (but you can run this demo on Android and iOS as well).

Now that we have a way to write and test a demo program, let's get back to square one, by covering all of the details of the first few building blocks of an application, as I promised at the beginning of this chapter. The first thing you need to know is how

to *read* a program, how the various elements are written, and what is the structure of the application we just build (which has both a PAS file and DPR file).

Figure 1.3:
A simple mobile application with a single button, used by the HelloVisual demo

Syntax and Coding Style

Before we move on to the subject of writing actual Object Pascal language statements, it is important to highlight some elements of Object Pascal coding style. The question I'm addressing here is this: Besides the syntax rules (which we still haven't looked into), how should you write code? There isn't a single answer to this question, since personal taste can dictate different styles. However, there are some principles you need to know regarding comments, uppercase, spaces, and what many years ago was called *pretty-printing* (pretty for us human beings, not the computer), a term now considered obsolete.

In general, the goal of any coding style is clarity. The style and formatting decisions you make are a form of shorthand, indicating the purpose of a given piece of code. An essential tool for clarity is consistency—whatever style you choose, be sure to follow it throughout a project and across projects.

note The IDE (Integrated Development Environment) has support for automatic code formatting (at the unit or project level): You can ask the editor to re-format your code with the Ctrl+D keys, following a set of rules you can change by tweaking about 40 different formatting elements (found among the IDE Options), and even share these settings with other developers on your team to make formatting consistent.

Comments

Although code is often self-explanatory, it is relevant to add a significant amount of comments in the source code of a program, to further explain to others (and to the yourself when you look at your code a long time in the future) why the code was written in a given way and what were the assumptions.

In traditional Wirth Pascal comments were enclosed in either braces or parentheses followed by a star. Modern versions of the language also accept the C++ style comments, double slash, which span to the end of the line and require no symbol to indicate the end the comment:

```
{ this is a comment }
(* this is another comment *)
// this is a comment up to the end of the line
```

The first form is shorter and more commonly used. The second form was often preferred in Europe because many European keyboards lacked the brace symbol. The third form of comment has been borrowed from C/C++, which also use the `/* comment */` syntax for multiline comments, along with C#, Objective-C, Java, and JavaScript.

Comments up to the end of the line are very helpful for short comments and for commenting out a single line of code. They are by far the most common form of comments in the Object Pascal language.

note In the IDE editor, you can comment or uncomment the current line (or a group of selected lines) with a direct keystroke. This is Ctrl+/ on the US keyboard and a different combination (with the physical / key) on other keyboards: The actual key is listed in the popup menu of the editor.

Having three different forms of comments can be helpful for marking nested comments. If you want to comment out several lines of source code to disable them, and

these lines contain some real comments, you cannot use the same comment identifier:

```
{
  code...
  {comment, creating problems}
  code...
}
```

The code above results in a compiler error, as the first closed brace indicates the end of the entire commented section. With a second comment identifier, you can write the following code, which is correct:

```
{
  code...
  // this comment is OK
  code...
}
```

An alternative is to comment out a group of lines as explained above, given it will add a second // comment to the commented line, you can easily remove by uncommenting the same block (preserving the original comment).

---

note If the open brace or parenthesis-star is followed by the dollar sign($), it is not a comment any more, but becomes a compiler directive, as we have seen in the first demo in the line `{$APPTYPE CONSOLE}`. Compiler directives instruct the compiler to do something special, and are briefly explained towards the end of this chapter.

Actually, compiler directives are still comments. For example, *{$X+ This is a comment}* is legal. It's both a valid directive and a comment, although most *sane* programmers will probably tend to separate directives and comments.

Symbolic Identifiers

A program is made of many different symbols you can introduce to name the various elements (data types, variables, functions, objects, classes, and so on). Although you can use almost any identifier you want, there are a few rules you have to follow:

- Identifiers cannot include spaces (as spaces do separate identifiers from other language elements)
- Identifiers can use letters and numbers, including the letters in the entire Unicode alphabet; so you can name symbols in your own language if you want
- Out of the traditional ASCII symbols, identifiers can use only the underscore symbol (_); all other ASCII symbols beside letters and numbers are not allowed. Illegal symbols in identifiers include match symbols (+, -, *, /, =),

all parenthesis and braces, punctuation, special characters (including @, #, $, %, ^, &, \, |). What you can use, though, are Unicode symbols, like ☼ or ∞.

- Identifiers must start with a letter or the underscore, starting with a number is not allowed (in other words, you can use numbers, but not as the first symbol). Here with numbers we refer to the ASCII numbers, 0 to 9, while other Unicode representations of numbers are allowed.

The following are examples of classic identifiers, listed in the `IdentifiersTest` demo:

```
MyValue
Value1
My_Value
_Value
Val123
_123
```

These are example of legal Unicode identifiers (where the last is a bit extreme):

```
Cantù (Latin accented letter)
结 (Cash Balance in Simplified Chinese)
画像 (picture in Japanese)
✲ (Sun Unicode symbol)
```

These are a few examples of *invalid* identifiers:

```
123
1Value
My Value
My-Value
My%Value
```

tip In case you want to check for a valid identifier at runtime (something rarely needed, unless you are writing a tool to help other developers), there is a function in the runtime library that you can use, called `IsValidIdent`.

Use of Uppercase

Unlike many other languages, including all those based on the C syntax (like C++, Java, C#, and JavaScript), the Object Pascal compiler ignores the case, or capitalization, of the identifiers. Therefore, the identifiers `Myname`, `MyName`, `myname`, `myName`, and `MYNAME` are all exactly the same. In my opinion, case-insensitivity is definitely a positive feature, as syntax errors and other subtle mistakes can be caused by incorrect capitalization in case-sensitive languages.

If you consider the fact that you can use Unicode for identifiers, however, things get a bit more complicated, as the uppercase version of a letter is treated like the same

element, while an accented version of the same letter is treated like a separate symbol. In other words:

```
cantu: Integer;
Cantu: Integer; // error: duplicate identifier
cantù: Integer; // correct: different identifier
```

note There is only one exception to the case-insensitivity rule of the language: the *Register* procedure of a components' package must start with the uppercase *R*, because of a C++ compatibility issue. Of course, when you refer to identifiers exported by other languages (like a native operating system function) you might have to use the proper capitalization.

There are a couple of subtle drawbacks, however. First, you must be aware that these identifiers really are the same, so you must avoid using them as different elements. Second, you should try to be consistent in the use of uppercase letters, to improve the readability of the code.

A consistent use of case isn't enforced by the compiler, but it is a good habit to get into. A common approach is to capitalize only the first letter of each identifier. When an identifier is made up of several consecutive words (you cannot insert a space in an identifier), every first letter of a word should be capitalized:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

This is often called “Pascal-casing”, to contrast it with the so-called “Camel-casing” of Java and other languages based on the C syntax, which capitalizes internal words with an initial lowercase letter, as in

```
myLongIdentifier
```

Actually, it is more and more common to see Object Pascal code in which local variables use camel-casing (lowercase initial), while class elements, parameters and other more global elements use the Pascal-casing. In any case, in the book source code snippets I've tried to use Pascal-casing consistently for all symbols.

White Space

Other elements completely ignored by the compiler are the spaces, new lines, and tabs you add to the source code. All these elements are collectively known as *white space*. White space is used only to improve code readability; it does not affect the compilation in any way.

Unlike traditional BASIC, Object Pascal allows you to write a statement over several lines of code, splitting a long instruction over two or more lines. The drawback of allowing statements over more than one line is that you have to remember to add a

semicolon to indicate the end of a statement, or more precisely, to separate one statement from the next. The only restriction in splitting programming statements on different lines is that a string literal may not span several lines.

Although odd, the following blocks all represent the same statement:

```
a := b + 10;

a :=
  b
  +
  10;

a
:=
// this is a mid-statement comment
b + 10;
```

Again, there are no fixed rules on the use of spaces and multiple-line statements, just some rules of thumb:

- The editor has a vertical line you can place after 80 or so characters. If you use this line and try to avoid surpassing this limit, your source code will look better and you won't have to scroll horizontally to read it on a computer with a smaller screen. The original idea behind the 80 characters was to make the code look nicer when printed, something not so common these days (but still valuable).
- When a function or procedure has several complex parameters, it is common practice to place the parameters on different lines, a habit that mostly comes from the C language.
- You can leave a line completely white (blank) before a comment or to divide a long piece of code in smaller portions. Even this simple idea can improve the readability of the code.
- Use spaces to separate the parameters of a function call, and maybe even a space before the initial open parenthesis. Also I like keeping operands of an expression separated, although this is a matter of preference..

Indentation

The last suggestion on the use of white spaces relates to the typical Pascal language-formatting style, originally known as pretty-printing but now generally referred as indentation.

This rule is simple: Each time you need to write a compound statement, indent it two spaces (not a tab, like a C programmer would generally do) to the right of the

current statement. A compound statement inside another compound statement is indented four spaces, and so on:

```
if ... then
  statement;

if ... then
begin
  statement1;
  statement2;
end;

if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

Again, programmers have different interpretations of this general rule. Some programmers indent the `begin` and `end` statements to the level of the inner code, other programmers put the `begin` at the end of the line of previous statement (in a C-like fashion). This is mostly a matter of personal taste.

A similar indentation format is often used for lists of variables or data types:

```
type
  Letters = ('A', 'B', 'C');
  AnotherType = ...

var
  Name: string;
  I: Integer;
```

In the past it was also common to use an indentation of the type when declaring custom types and variables, but this is now less frequent. In such a case, the code above will look like:

```
type
  Letters     = ('A', 'B', 'C');
  AnotherType = ...

var
  Name : string;
  I    : Integer;
```

Indentation is also used commonly for statements that continue from the previous line or for the parameters of a function (if you don't put each parameter on a separate line):

```
MessageDlg ('This is a message',
  mtInformation, [mbOk], 0);
```

Syntax Highlighting

To make it easier to read and write Object Pascal code, the IDE editor has a feature called syntax highlighting. Depending on the meaning in the language of the words you type, they are displayed using different colors and font styles. By default, keywords are in bold, strings and comments are in color (and often in italic), and so on.

Reserved words, comments, and strings are probably the three elements that benefit most from this feature. You can see at a glance a misspelt keyword, a string not properly terminated, and the length of a multiple-line comment.

You can easily customize the syntax highlight settings using the Editor Colors page of the Options dialog box. If you are the only person using your computer to look to Object Pascal source code, choose the colors you like. If you work closely with other programmers, you should all agree on a standard color scheme. I often found that working on a computer with a different syntax coloring than the one I normally use was really confusing.

Error Insight and Code Insights

The IDE editor has many more features to help you write correct code. The most obvious is Error Insight, that places a red squiggle under source code elements it doesn't understand, in the same fashion that a word processor marks spelling mistakes.

---

note At times you need to compile your program a first time to avoid having Error Insight indications for perfectly legitimate code. Also saving a file such as a form might force the inclusion of the proper units required for the current components, solving incorrect Error Insight indications.

Other features, like Code Completion, help you write code by providing a list of legitimate symbols in the place where you are writing. When a function or method has parameters, you'll see those listed as you type. And you can also hover over a symbol to see its definition. However, these are editor specific features that I don't want to delve into in detail, as I want to remain focused on the language and not discuss the IDE editor in detail (even if it is by far the most common tools used for writing Object Pascal code).

Language Keywords

Keywords are all the identifiers reserved by the language. These are symbols that have a predefined meaning and role and you cannot use them in a different context. Formally there is a distinction between reserved words and directives: Reserved words cannot be used as identifiers, while directives have a special meaning but could be used also in a different context (although you are recommended not to do so). In practice, you should not use any keyword as an identifier.

If you write some code like the following (where `property` is indeed a keyword):

```
var
  property: string
```

you'll see an error message like:

```
E2029 Identifier expected but 'PROPERTY' found
```

In general when you misuse a keyword, you'll get different error messages depending on the situation, as the compiler recognizes the keyword, but gets confused by its position in the code or by the following elements.

Here I don't want to show you a complete list of keywords, as some of them are rather obscure and rarely used, but only list a few, grouping them by their role. It will take me several chapters to explore all of these keywords and others I'm skipping in this list.

---

note Notice that some keywords can be used in different contexts, and here I'm generally referring only to the most common context (although a couple of keywords are listed twice). One of the reasons is that over the years the compiler team wanted to avoid introducing new keywords, as this might break existing applications, so they *recycled* some of the existing ones.

So let's start our exploration of keywords with some you've already seen in the initial demo source code and that are used to define the **structure of an application project**:

`program`	Indicates the name of an application project
`library`	Indicates the name of a library project
`package`	Indicates the name of a package library project
`unit`	Indicates the name of a unit, a source code file
`uses`	Refers to other units the code relies upon
`interface`	The part of a unit with declarations
`implementation`	The part of a unit with the actual code
`initialization`	Code executed when a program starts

`finalization`	Code executed on program termination
`begin`	The start of a block of code
`end`	The end of a block of code

Another set of keywords relates to the declaration of different basic **data types and variables** of such data types:

`type`	Introduces a block of type declarations
`var`	Introduces a block of variable declarations
`const`	Introduces a block of constant declarations
`set`	Defines a *power set* data type
`string`	Defines a string variable or custom string type
`array`	Defines an array type
`record`	Defines a record type
`integer`	Defines an integer variable
`real`	Defines a floating point variable
`file`	Defines a file
`record`	Defines record type

---

note There are many other data types defined in Object Pascal that I will cover later.

A third group includes keywords is used for the **basic language statements**, such a condition and loops, including also functions and procedures:

`if`	Introduces a conditional statement
`then`	Separates the condition from the code to execute
`else`	Indicates possible alternative code
`case`	Introduces a conditional statement with multiple options
`of`	Separates the condition from the options
`for`	Introduces a fixes repetitive cycle
`to`	Indicates the final upper value of the `for` cycle
`downto`	Indicates the final lower value of the `for` cycle
`in`	Indicates the collection to iterate over in a cycle
`while`	Introduces a conditional repetitive cycle
`do`	Separates the cycle condition from the code
`repeat`	Introduces a repetitive cycle with a final condition
`until`	Indicates the final condition of the cycle
`with`	Indicates a data structure to work with
`function`	A sub-routine or group of statements returning a result

`procedure`	A sub-routine or group of statements which doesn't return a result
`inline`	Requests the compiler to optimize a function or procedure
`overload`	Allows the reuse of the name of a function or procedure

Many other keywords relate with **classes and objects**:

`class`	Indicates a class type
`object`	Used to indicate an older class type (now deprecated)
`abstract`	A class that is not fully defined
`sealed`	A class from which other classes cannot inherit
`interface`	Indicates an interface type (listed also in the first group)
`constructor`	An object or class initialization method
`destructor`	An object or class cleanup method
`virtual`	A virtual method
`override`	The modified version of a virtual method
`inherited`	Refers to a method of the base class
`private`	Portion of a class not accessible from the outside
`protected`	Portion of a class with limited access from the outside
`public`	Portion of a class fully accessible from the outside
`published`	Portion of a class made specifically available to users
`strict`	A stronger limitation for private and protected sections
`property`	A symbol mapped to a value or method
`read`	The mapper for getting the value of a property
`write`	The mapper for setting the value of a property
`nil`	The value of a *zero* object (used also for other entities)

A smaller group of keywords is used for **exceptions handling** (see Chapter 11):

`try`	The start of an exception handling block
`finally`	Introduces code to be executed regardless of an exception
`except`	Introduces code to be executed in case of an exception
`raise`	Used to trigger an exception

Another group of keywords is used for **operators** and is covered in the section "Expressions and Operators" later in this chapter, (beside some advanced operators covered only in later chapters):

```
as              and             div
is              in              mod
not             or              shl
shr             xor
```

Finally, here is partial list of other **rarely used keywords**, including some old ones you should really avoid using. Look them up in the help or in the index of this book, if you are interested in more information about these:

```
default                dynamic                export
exports                external               file
forward                goto                   index
label                  message                name
nodefault              on                     out
packed                 reintroduce            requires
```

Notice that the list of Object Pascal language keywords has seen very few additions over recent years, as any additional keyword implies potentially introducing compilation errors into some existing programs preventing that had happened to use one of the new keyword as a symbol. Most of the recent additions to the language required no new keyword, like generics and anonymous methods.

The Structure of a Program

You hardly ever write all of the code in a single file, although this was the case with the first simple console application I showed earlier in this chapter. As soon as you create a visual application, you get at least one secondary source code file beside the project file. This secondary file is called *unit* and it's indicated by the PAS extension (for Pascal source unit), while the main project file uses the DPR extension (for Delphi Project file). Both files contain Object Pascal source code.

Object Pascal make extensive use of units, or program modules. Units, in fact, can be used to provide modularity and encapsulation even without using objects. An Object Pascal application is generally made up of several units, including units hosting forms and data modules. In fact, when you add a new form to a project, the IDE actually adds a new unit, which defines the code of the new form.

Units do not need to define forms; they can simply define and make available a collection of routines, or one of more data types (including classes). If you add a new blank unit to a project, it will only contain the keywords used to delimit the sections a unit is divided into:

```
unit Unit1;

interface

implementation

end.
```

The structure of a unit is rather simple, as shown above:

- First, a unit has a unique name corresponding to its filename (that is, the sample unit above must be stored in the file *Unit1.pas*).
- Second, the unit has an `interface` section declaring what is visible to other units.
- Third, the unit has an `implementation` section with implementation details, the actual code, and possibly other local declarations, not visible outside of the unit.

Unit and Program Names

As I mentioned a unit name must correspond to the name of the file of that unit. The same is true for a program. To rename a unit, you perform a Save As operation in the IDE, and the two will be kept in synch. Of course, you can also change the file name at the file system level, but if you don't also change the declaration at the beginning of the unit, you'll see an error when the unit is compiled (or even when it is loaded in the IDE). This is a sample error message you'll get if you change the declaration of a unit without updating also the file name:

```
[DCC Error] E1038 Unit identifier 'Unit3' does not match file name
```

The implication of this rule is that a unit or program name must be a valid Pascal identifier, but also a valid file name in the file system. For example, it cannot contain a space, not special characters beside the underscore (_), as covered earlier in this chapter in the section on identifiers. Given units and programs must be named using an Object Pascal identifier, they automatically result in valid file names, so you should not worry about that. The exception, of course, would be using Unicode symbols that are not valid file names at the file system level.

Dotted Unit Names

There is an extension to the basic rules for unit identifiers: a unit name can use a dotted notation. So all of the following are all valid unit names:

```
unit1
myproject.unit1
mycompany.myproject.unit1
```

The reason for this extension is that unit names must be unique, and with more and more units being provided by Embarcadero and by third party vendors, this became more and more complex. All of the RTL units and the various other units that ship as part of the product libraries now follow the dotted unit name rule, with specific prefixes denoting the area, such as:

- `System` for core RTL

- `Data` for database access and the like
- `FMX` for the FM platform (the multi-device components)
- `VCL` for the Visual Component Library for Windows

---

note You'd generally refer to a dotted unit names, including the library units, with the complete name. It is also possible to use only the last portion of the name in a reference (allowing backward com - patibility with older code) by setting up a corresponding rule in the project options. This setting is called “Unit scope names” and it is a semicolon separated list. Notice, however, that using this fea - ture tends to slow down the compilation compared to using fully qualified unit names.

More on the Structure of a Unit

Beside the interface and implementation sections, a unit can have an optional `initialization` section with some startup code, to be executed when the program is first loaded into memory. If there is an `initialization` section you can add also have a `finalization` section, to be executed on program termination.

---

note You can also add initialization code in a class constructor, a recent language feature covered in Chapter 12. Using class constructors helps the linker remove unneeded code, which is why it is recommended to use class constructors and destructors, rather than the old initialization and finalization sections. As a historical note, the compiler still supports using the `begin` keyword in place of the `initialization` keyword. A similar use of `begin` is still standard in the project source code.

In other word, the general structure of a unit, with all its possible sections and some sample elements, is like the following:

```
unit unitName;

interface

// other units we refer to in the interface section
uses
  unitA, unitB, unitC;

// exported type definitions
type
  newType = TypeDefinition;

// exported constants
const
  Zero = 0;

// global variables
var
  Total: Integer;
```

```
// list of exported functions and procedures
procedure MyProc;

implementation

// other units we refer to in the implementation
uses
  unitD, unitE;

// hidden global variable
var
  PartialTotal: Integer;

// all the exported functions must be coded
procedure MyProc;
begin
  // ... code of procedure MyProc
end;

initialization
  // optional initialization code

finalization
  // optional clean-up code

end.
```

The purpose of the interface part of a unit is to make details of what the unit contains and can do to programs and other units that will make use of the unit. Whereas the implementation part contains the nuts and bolts of the unit which are hidden from outside viewers. This is how Object Pascal can provide this so called encapsulation even without using classes and objects.

As you can see, the `interface` of a unit can declare a number of different elements, including procedures, functions, global variables, and data types. Data types are generally used the most. The IDE automatically places a new class data type in a unit each time you create a visual form. However, containing form definitions is certainly not the only use for units in Object Pascal. You can have code only units, with functions and procedures (in a traditional way) and with classes that do not refer to forms or other visual elements.

The Uses Clause

The `uses` clause at the beginning of the interface section indicates which other units we need to access in the interface portion of the unit. This includes the units that define the data types we refer to in the definition of data types of this unit, such as the components used within a form we are defining.

The second `uses` clause, at the beginning of the implementation section, indicates additional units we need to access only in the implementation code. When you need to refer to other units from the code of the routines and methods, you should add elements in this second uses clause instead of the first one. All the units you refer to must be present in the project directory or in a directory of the search path.

tip You can set the Search Path for a project in the Project Options. The system also considers units in the Library path, which is a global setting of the IDE.

C++ programmers should be aware that the `uses` statement does not correspond to an include directive. The effect of a uses statement is to import just the pre-compiled interface portion of the units listed. The implementation portion of the unit is considered only when that unit is compiled. The units you refer to can be both in source code format (PAS) or compiled format (DCU).

Although seldom used, Object Pascal had also an `$INCLUDE` compiler directive that works similarly to C/C++ includes. These special include files are used by some libraries for sharing compiler directives or other settings among multiple units, and generally have the INC file extension. This directive is covered shortly at the end of this chapter.

note Notice that compiled units in Object Pascal are compatible only if they are build with the same version of the compiler and system libraries. A unit compiled in an older version of the product is generally not compatible with a later version of the compiler.

Units and Scope

In Object Pascal units are the key to encapsulation and visibility and, in that sense, they are probably even more important than the `private` and `public` keywords of a class. The scope of an identifier (such as a variable, procedure, function, or a data type) is the portion of the code in which the identifier is accessible or *visible*. The basic rule is that an identifier is meaningful only within its scope—that is, only within the unit, function, or procedure in which it is declared. You cannot use an identifier outside its scope.

note Unlike C or C++, Object Pascal doesn't have the concept of a generic code block that can include a declaration. While you can use begin and end to create a compound statement, this isn't like a C/C++ block with curly braces that has its own scope for internally declared variables.

In general, an identifier is visible only after it is defined. There are techniques in the language that allow declaring an identifier before its complete definition, but the general rule still applies if we consider both definitions and declarations.

Given that it makes little sense to write an entire program in a single file, though, how does the rule above change when you use multiple units? In short, by referring to another unit with a uses statement, the identifiers in the interface section of that unit becomes visible to the new unit.

Reversing the perspective, if you declare an identifier (type, function, class, variable, and so on) in the interface portion of a unit, it becomes visible to any other module referring to that unit. If you declare an identifier in the implementation portion of a unit, instead, it can only be used in that unit (and it is generally referred to as a *local identifier*).

Using Units Like Namespaces

We have seen that the uses statement is the standard technique to access identifiers declared in the scope of another unit. At that point you can access the definitions of the unit. But it might happen that two units you refer to declare the same identifier; that is, you might have two classes or two routines with the same name.

In this case you can simply use the unit name to prefix the name of the type or routine defined in the unit. For example, you can refer to the `ComputeTotal` procedure defined in the given `Calc` unit as `Calc.ComputeTotal`. This isn't required often, as you are strongly advised against using the same identifier for two different elements of the same program, if you can avoid it.

However, if you look into the system or third party libraries, you'll find functions and classes that have the same name. A good example are the visual controls of different user interface frameworks. When you see a reference to `TForm` or `TControl`, it could mean different classes depending on the actual units you refer to.

If the same identifier is exposes by two units in your uses statement, the one in the last unit being used overrides the symbol, and will be the one that the compiler uses. In other words, the symbols defined in the last unit in the list wins. If you simply cannot avoid such a scenario, it is recommended to prefix the symbol with the unit name, to avoid having you code depend on the order in which the units are listed.

The Program File

As we have seen, a Delphi application consists of two kinds of source code files: one or more units and one, and only one, program file. The units can be considered sec-

ondary files, which are referenced by the main part of the application, the program. In theory, this is true. In practice, the program file is usually an automatically generated file with a limited role. It simply needs to start up the program, generally creating and running the main form, in case of a visual application. The code of the program file can be edited manually, but it is also modified automatically by using some of the Project Options of the IDE (like those related to the application object and the forms).

The structure of the program file is usually much simpler than the structure of the units. Here is the source code of a sample program file (with some optional standard units omitted) that is automatically created by the IDE for you:

```
program Project1;

uses
  FMX.Forms,
  Unit1 in 'Unit1.PAS' {Form1};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

As you can see, there is simply a uses section and the main code of the application, enclosed by the begin and end keywords. The program's uses statement is particularly important, because it is used to manage the compilation and linking of the application.

---

note The list of units in the program file corresponds to the list of the units that are part of the project in the IDE Project Manager. When you add a unit to a project in the IDE, the unit is automatically added to the list in the program file source. The opposite happens if you remove it from the project. In any case, if you edit the source code of the program file, the list of units in the Project Manager is updated accordingly.

Compiler Directives

Another special element of the program structure (other than its actual code) are compiler directives, as mentioned earlier. These are special instructions for the compiler, written with the format:

```
{$X+}
```

Some compiler directives have a single character, as above, with a plus or minus symbol indicating if the directive is activated or unactivated. Most of the directives also have a longer and readable version, and use `ON` and `OFF` to mark if they are active. Some directives have only the longer, descriptive format.

Compiler directives don't generate compiled code directly, but affect how the compiler generates code after the directive is encountered. In many cases, using a compiler directive is an alternative to changing one of the compiler settings in the IDE Project Options, although there are scenarios in which you want to apply a specific compiler setting only to a unit or to a fragment of code.

I'll cover specific compiler directives when relevant in the discussion of a language feature they can affect. In this section I only want to mention a couple of directives that relate to the program code flow: conditional defines and includes.

Conditional Defines

Conditional defines like `$IFDEF` let you indicate to the compiler to include a portion of source code or ignore it. They can be based on defined symbols or on constant values. The defined symbols can be predefined by the system (like the platform symbols), can be defined in a specific project option, or a can be introduced with another compiler directive, `$DEFINE`:

```
{$DEFINE TEST}
...
{$IFDEF TEST}
  // this is going to be compiled
{$ENDIF}

{$IFNDEF TEST}
  // this is not going to be compiled
{$ENDIF}
```

You can also have two alternatives, using an `$ELSE` directive to separate them. A more flexible alternative is the use of the `$IF` directive, closed by the `$IFEND` directive and based on expressions like comparison functions (which can refer to any constant value in the code). So you can just define a constant and use an expression against it. An example is shown below related to compiler versions, one of the commonly used system defines.

Compiler Versions

Each version of the Delphi compiler has a specific define you can use to check if you are compiling against a specific version of the product. This might be required if you are using a feature introduced later but want to make sure the code still compiles for older versions.

If you need to have specific code for some of the recent versions of Delphi, you can base your `$IFDEF` statements on the following defines:

Delphi 2007	`VER180`
Delphi XE	`VER220`
Delphi XE2	`VER230`
Delphi XE4	`VER250`
Delphi XE5	`VER260`
Appmethod Spring 2014	`VER260`
Delphi XE6	`VER270`
Appmethod June 2014	`VER270`
Delphi XE7	`VER280`
Appmethod September 2014	`VER280`

The decimal digits of these version numbers indicate the actual compiler version (for example 26 in Delphi XE5). The numeric sequence is not specific to Appmethod or Delphi, but dates back to the first Pascal compiler published by Borland.

You can also use the internal versioning constant in `$IF` statements, with the advantage of being able to use a comparison operator (`>=`) rather than a match for a specific version. The versioning constant is called `CompilerVersion` and in Delphi XE5 it's assigned to the floating-point value `26.0`. So for example:

```
{$IF CompilerVersion >= 26)}
  // code to compile in Delphi XE5 or later
{$IFEND}
```

Similarly, you can use system defines for different platforms you can compile for, in case you need some code to be platform-specific (generally an exception in Object Pascal, not common practice):

Windows (both 32 and 64 bit)	`MSWINDOWS`
Mac OS X	`MACOS`
iOS	`IOS`
Android	`ANDROID`

Below is a code snippet with the tests based on the platforms define above, part of the `HelloPlatform` project:

```
{$IFDEF IOS}
  ShowMessage ('Running on iOS');
{$ENDIF}

{$IFDEF ANDROID}
  ShowMessage ('Running on Android');
{$ENDIF}
```

Include Files

The other directive I want to cover here is the `$INCLUDE` directive, already mentioned when discussing the uses statement. This directive lets you refer to and include a fragment of source code in a given position of a source code file. At times this is used to be able to include the same fragment in different units, in cases where the code fragment defines compiler directives and other elements used directly by the compiler. When you use a unit, it is compiled only one. When you include a file, that code is compiled within each of the units it is added to (which is why you should generally avoid having any new identifier declared in an include file).

In other words, you should generally not add any language elements and definitions in include files (unlike the C language), as this is what units are for. So how do you use an include file? A good example is a set of compiler directives you want to enable in most of your units, or some extra special defines.

Large libraries often use include files for that purpose, an example would be the FireDAC library, a database library which is now part of the system libraries. Another example, showcased by the system RTL units, is the use of individual includes for each platform, with an IFDEF used for conditionally including only one of them.

02: variables and data types

Object Pascal is what is known as a *strongly-typed* language. Variables in Object Pascal are declared to be of a *data type (or user defined data type*). The type of a variable determines the values a variable can hold, and the operations that can be performed on it. This allows the compiler both to identify errors in your code and generate faster programs for you.

This is why the concept of type is stronger in Pascal than in languages like C or C++. Later languages based on the same syntax but that break compatibility with C, like C# and Java, divert from C and embrace Pascal's strong notion of data type. In C, for example, arithmetic data types are almost interchangeable. By contract the original versions of BASIC, had no similar concept, and in many of today's scripting languages (JavaScript being an obvious example) the notion of data type is very different.

note In fact, there are some tricks to bypass type safety, like using variant record types. The use of these tricks is strongly discouraged and are little used today.

As I mentioned, all of the dynamic languages, from JavaScript onwards, don't have the same notion of data type, or (at least) have a very loose idea of types. In some of these languages the type is inferred by the value you assign to a variable, and the variable type can change over time. What is important to point out is that data types are a key element for enforcing correctness of a large application at compile-time, rather than relying on run-time checks. Data types require more order and structure, and some planning of the code you are going to write... which clearly has advantages and disadvantages.

note Needless to say I prefer strongly typed languages, but in any case my goal in this book is to explain how the language works, more than to advocate why I think it is such a great programming language. Though I'm sure you'll get that impression while you read the book.

Variables and Assignments

Like other strongly-typed languages, Object Pascal requires all variables to be declared before they are used. Every time you declare a variable, you must specify a data type. Here are some variable declarations:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

The `var` keyword can be used in several places in a program, such as at the beginning of a function or procedure, to declare variables local to that portion of the code, or inside a unit to declare global variables.

note Differently from C and other curly-brace languages, in Object Pascal you cannot mix variable declarations with programming statements, but you need to group them in specific sections (like at the beginning of a method). Since this is not always handy, the IDE code editor let's you actually *type* the *var* keyword followed by the actual declaration within your method or function code, but it will immediately move it up to the correct position. This is one of the predefined Live Templates, a very nice coding helper in the IDE that you can customize and extend.

After the `var` keyword comes a list of variable names, followed by a colon and the name of the data type. You can write more than one variable name on a single line, as `A` and `B` in the last statement of the previous code snippet (a coding style that is less common today, compared to splitting the declaration on two separate lines).

Once you have defined a variable of a given type, you can only perform the operations supported by its data type on it. For example, you can use the Boolean value in a test and the integer value in a numerical expression. You cannot mix Booleans and Integers, for example, or any incompatible data type (even if the internal representation might is physically compatible, as it is the case for Booleans and Integers).

The simplest assignment is that of an actual value, let's say you want the `Value` variable to hold the value `10`. But how do you express literal values? While this concept might be obvious, it is worth looking into it with some detail.

Literal Values

A literal value is a value you type directly in the program source code. If you need a number with the value of twenty, you can simply write:

```
20
```

There is also an alternative representation of the same numeric value, based on the hexadecimal value, like:

```
$14
```

These will be literal values for an integer number (or one of the various integer number types that are available). If you want the same numeric value, but for a floating point literal value, you generally add an empty decimal after it:

```
2.0
```

Literal values are not limited to numbers. You can also have characters and strings. Both use single quotes (were many other programming languages will use double quotes for both, or single quotes for characters and double quotes for strings):

```
// literal characters
'K'
#55

// literal string
'Marco'
```

As you can see above, you can also indicate characters by their corresponding numeric value (originally the ASCII number, now the Unicode code point value), prefixing the number with the # symbol, as in (#32, for a space). This is useful mostly for control character without a textual representation in the source code, like a backspace or a tab.

In case you need to have a quote within a string, you'll have to double it. So if I want to have my first and last name (spelled with a final quote rather than an accent) I can write:

```
'Marco Cantu'''
```

The two quotes stand for a quote within the string, while the third consecutive quote marks the end of the string. Also note that a string literal must be written on a single line, but you can concatenate multiple string literals using the + sign. If you want to have the new line or line break within the string, don't write it on two lines, but concatenate the two elements with the `sLineBreak` system constant (which is platform specific), as in:

```
'Marco' + sLineBreak + 'Cantu'''
```

Assignment Statements

Assignments in Object Pascal use the colon-equal operator (`:=`), an odd notation for programmers who are used to other languages. The = operator, which is used for assignments in many other languages, in Object Pascal is used to test for equality.

note The := operators comes from a Pascal predecessor, Algol, a language few of todays developers have heard of (let even used). Most of today's languages avoid the := notation and favor the = assignment notation.

By using different symbols for an assignment and an equality test, the Pascal compiler (like the C compiler) can translate source code faster, because it doesn't need to examine the context in which the operator is used to determine its meaning. The use of different operators also makes the code easier for people to read. Truly Pascal picked two different operators than C (and syntactic derivatives like Java, C#, JavaScript), which uses = for assignment and == for equality test.

note For the sake of completeness I should mention JavaScript has also a === operator, but that's something that even most JavaScript programmers get confused about.

The two elements of an assignment are often called *lvalue* and *rvalue*, for left value (the variable or memory location you are assigning to) and right value, the value of the expressions being assigned. While the rvalue can be an expression, the lvalue must refer (directly or indirectly) to a memory location you can modify. There are some data types that have specific assignment behaviors which I'll cover in due time.

The other rule is that the type of the lvalue and of the rvalue must match, or there must be an automatic conversion between the two, as explained in the next section.

Assignments and Conversion

Using simple assignments, we can write the following code (that you can find among many other snippets in this section in the `VariablesTest` project):

```
Value := 10;
Value := Value + 10;
IsCorrect := True;
```

Given the previous variable declarations, these two assignments are correct. The next statement, instead, is not correct, as the two variables have different data types:

```
Value := IsCorrect; // error
```

If you try to compile this code, the compiler issues an error with a description like this:

```
[dcc32 Error]: E2010 Incompatible types: 'Integer' and 'Boolean'
```

The compiler informs you that there is something wrong in your code, namely two incompatible data types. Of course, it is often possible to convert the value of a variable from one type to another type. In some cases, the conversion is automatic, for example if you assign an integer value to a floating point variable (but not the opposite, of course). Usually you need to call a specific system function that changes the internal representation of the data.

Initializing Global Variable

For global variables, you can assign an initial value as you declare the variable, using the constant assignment notation covered below (=) instead of the assignment operator (:=). For example, you can write:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

This initialization technique works only for global variables, which are initialized to their default values anyway (like 0 for a number).

Variables declared at the beginning of a procedure or function, instead, are not initialized to a default value and don't have an assignment syntax. For those variables, it is often worth adding explicit initialization code at the beginning of the code:

```
var
  Value: Integer;
begin
  Value := 0; // initialize
```

Again, if don't initialize a local variable but use it as it is, the variable will have a totally random value (depending on the bytes that were present at that memory location). In several scenarios, the compiler will warn you of the potential error.

In other words, if you write:

```
var
  Value: Integer;
begin
  ShowMessage (Value.ToString); // X is indefined
```

The output will be a totally random value, whatever bytes happened to be at the memory location of the `Value` variable considered as an Integer.

Constants

Object Pascal also allows the declaration of constants. This let's you to give meaningful names to values that do not change during program execution (and possibly reducing the size by not duplicating constant values in your compiled code).

To declare a constant you don't need to specify a data type, but only assign an initial value. The compiler will look at the value and automatically infer the proper data type. Here are some sample declarations (also from the `VariablesTest` example):

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantu';
```

The compiler determines the constant data type based on its value. In the example above, the `Thousand` constant is assumed to be of type `SmallInt`, the smallest integral type that can hold it. If you want to tell the compiler to use a specific type you can simply add the type name to the declaration, as in:

```
const
  Thousand: Integer = 1000;
```

When you declare a constant, the compiler can choose whether to assign a memory location to the constant and save its value there, or to duplicate the actual value each time the constant is used. This second approach makes sense particularly for simple constants.

Once you have declared a constant you can use it almost like any other variable, but you cannot assign a new value to it. If you try, you'll get a compiler error.

note Oddly enough, Object Pascal does allow you to change the value of a typed constant at run-time, as if it was a variable but only if you enable the `$J` compiler directive, or use the corresponding *Assignable typed constants* compiler option. This optional behavior is included for backward compatibility of code which was written with an old compiler. This is clearly not a suggested cod - ing style, and I've covered it in this note most as a historical anecdote about such programming techniques.

Resource String Constants

Although this is a slightly more advanced topic, when you define a string constant, instead of writing a standard constant declaration you can use a specific directive, `resourcestring`, that indicates to the compiler and linker to treat the string like a Windows resource (or an equivalent data structure on non-Windows platforms Object Pascal supports):

```
const
  sAuthorName = 'Marco';

resourcestring
  strAuthorName = 'Marco';

begin
  ShowMessage (strAuthorname);
```

In both cases you are defining a constant; that is, a value you don't change during program execution. The difference is only in the internal implementation. A string constant defined with the `resourcestring` directive is stored in the resources of the program, in a string table.

In short, the advantages of using resources are more efficient memory handling performed by Windows, a corresponding implementation for other platforms, and a better way of localizing a program (translating the strings to a different language) without having to modify its source code. As a rule of thumb, you should use `resourcestring` for any text that is shown to users and might need translating, and internal constants for every other internal program string, like a fixed configuration file name.

tip The IDE editor has an automatic *refactoring* you can use to replace a string constant in your code with a corresponding `resourcestring` declaration. Place the edit cursor within a string literal and press Ctrl+Shift+L to activate this refactoring.

Lifetime and Visibility of Variables

Depending on how you define a variable, it will use different memory locations and remain available for a different amount of time (something generally called the variable *lifetime*) and will be available in different portions of your code (a feature referred to by the term *visibility*).

Now, we cannot have a complete description of all of the options so early in the book, but we can certainly consider the most relevant cases:

- **Global variables**: If you declare a variable (or any other identifier) in the interface portion of the unit, its scope extends to any other unit that uses the one declaring it. The memory for this variable is allocated as soon as the program starts and exists until it terminates. You can assign a default value to it or use the initialization section of the unit in case the initial value is computed in a more complex way.
- **Global hidden variables**: If you declare a variable in the implementation portion of a unit, you cannot use it outside that unit, but you can use it in any block of code and procedure defined within the unit, from the position of the declaration onwards. Such a variable uses global memory and has the same lifetime as the first group; the only difference is in its visibility. The initialization is the same as that of global variable.
- **Local variables**: If you declare a variable within the block defining a function, procedure, or method, you cannot use this variable outside that block of code. The scope of the identifier spans the whole function or method, including nested routines (unless an identifier with the same name in the nested routine hides the outer definition). The memory for this variable is allocated on the stack when the program executes the routine defining it. As soon as the routine terminates, the memory on the stack is automatically released.

Any declarations in the interface portion of a unit are accessible from any part of the program that includes the unit in its uses clause. Variables of form classes are declared in the same way, so that you can refer to a form (and its public fields, methods, properties, and components) from the code of any other form. Of course, it's poor programming practice to declare everything as global. Besides the obvious memory consumption problems, using global variables makes a program harder to maintain and update. In short, you should use the smallest possible number of global variables.

Data Types

In Pascal there are several predefined data types, which can be divided into three groups: *ordinal types*, *real types*, and *strings*. We'll discuss ordinal and real types in the following sections, while strings will be specifically covered in Chapter 6.

Delphi also includes a *non-typed* data type, called *variant*, and other "flexible" types, such as `TValue` (part of the enhanced RTTI support). Some of these more advanced data types will be discussed later in Chapter 5.

Ordinal and Numeric Types

Ordinal types are based on the concept of order or sequence. Not only can you compare two values to see which is higher, but you can also ask for the next or previous values of any value and compute the lowest and highest possible values the data type can represent.

The three most important predefined ordinal types are `Integer`, `Boolean`, and `Char` (character). However, there are other related types that have the same meaning but a different internal representation and support a different range of values. The following table lists the ordinal data types used for representing numbers:

Size	Signed	Unsigned
8 bits	`ShortInt`: -128 to 127	`Byte`: 0 to 255
16 bits	`SmallInt`: -32768 to 32767 (-32K to 32K)	`Word`: 0 to 65,535 (0 to 64K)
32 bits	`Integer`: -2,147,483,648 to 2,147,483,647 (-2GB to +2GB)	`Cardinal`: 0 to 4,294,967,295 (0 to 4 GB)
64 bits	`Int64`: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`UInt64`: 0 to 18,446,744,073,709,551,615 (*if you can read it!*)

As you can see, these types correspond to different representations of numbers, depending on the number of bits used to express the value, and the presence or absence of a sign bit. Signed values can be positive or negative, but have a smaller range of values (half of the corresponding unsigned value), because one less bit is available for storing the value itself.

The Int64 type represents integer numbers with up to 18 digits. This type is fully supported by some of the ordinal type routines (such as `High` and `Low`), numeric routines (such as `Inc` and `Dec`), and string-conversion routines (such as `IntToStr`) of the run time library.

Aliased Integral Types

If you have a hard time remembering the difference between a `ShortInt` and a `SmalInt` (including which one is effectively smaller), rather than the actual type you can use one of the predefined aliases declared in the `System` unit:

```
type
  Int8   = ShortInt;
  Int16  = SmallInt;
  Int32  = Integer;
  UInt8  = Byte;
  UInt16 = Word;
  UInt32 = Cardinal;
```

Again, these types don't add anything new, but are probably easier to use, as it is simple to remember the actual implementation of an Int16 rather than that of a SmallInt. These type aliases are also easier to use for developers coming from C and other languages that use similar type names.

Integer Type, 64bit, and NativeInt

In 64-bit versions of Object Pascal you may be surprised to learn that the Integer type is still 32 bit. It is so because this is the most efficient type for numeric processing.

The Pointer type (more about pointers later on) and other related reference types that are 64 bit. If you need a numeric type that adapts to the pointer size and the native CPU platform, you can use the two special `NativeInt` and `NativeUInt` aliased types. These are 32 bit on 32-bit platform and 64 bit on 64-bit platforms.

Integer Types Helpers

While the Integer types are treated separately from objects in the Object Pascal language, it is possible to operate on variables (and constant values) of these types with operations that you apply using “dot notation”. This is the notation generally used to apply methods to objects.

note Technically these operations on native data types are defined using “intrinsic record helpers”. Class and record helpers are covered in Chapter 12. In short, you can customize the operations applicable to core data types. Expert developers can notice that type operations are defined as class static methods in the matching intrinsic record helper.

You can see a couple of examples in the following code extracted from the `IntegersTest` demo:

```
var
  N: Integer;
begin
  N := 10;
  Show (N.ToString);

  // display a constant
  Show (33.ToString);

  // type operation, show the bytes required to store the type
  Show (Integer.Size.ToString);
```

note The `Show` function used in this code snippet is a simple procedure used to display some string out - put in a memo control, to avoid having to close multiple ShowMessage dialogs. A side advantage is this approach makes easier to copy the output and paste in the text (as I've done below). You'll see this approach used through most of the demos of this book.

The output of the program is the following

```
10
33
4
```

Given these operations are very important (more than others that are part of the run time library) it is worth listing them here:

`ToString`	Convert to the number to a string, using a decimal format
`ToBoolean`	Conversion to Boolean type
`ToHexString`	Convert to a string, using a hexadecimal format
`ToSingle`	Conversion to single floating point data type
`ToDouble`	Conversion to double floating point data type
`ToExtended`	Conversion to extended floating point data type

The first and third operations convert to the number to a string, using a decimal or hexadecimal operation. The second is a conversion to Boolean, while the last three are conversions to floating point types described later.

There are other operations you can apply to the Integer type (and most other numerical types), such as:

`Size`	The number of bytes required to store a variable of this type

`Parse`	Convert a string to the numeric value it represents
`TryParse`	Try to convert the string a a number

Standard Ordinal Types Routines

Beside the operations defined by Integer type helpers and listed above, there are several standard and "classic" functions you can apply to any ordinal type (not just the numeric ones). A classic example is asking for information about the type itself, using the functions `SizeOf`, `High`, and `Low`. The result of the `SizeOf` system function (that you can apply to any data type of the language) is an integer indicating the number of bytes required to represent values of the given type (just like the Size helper function shown above)

The system routines that work on ordinal types are shown in the following table:

`Dec`	Decrements the variable passed as parameter, by one or by the value of the optional second parameter
`Inc`	Increments the variable passed as parameter, by one or by the specified value
`Odd`	Returns True if the argument is an odd number. For testing for even numbers, you should use a `not` expression (`not Odd`)
`Pred`	Returns the value before the argument in the order determined by the data type, the predecessor
`Succ`	Returns the value after the argument, the successor
`Ord`	Returns a number indicating the order of the argument within the set of values of the data type (used for non-numerical ordinal types)
`Low`	Returns the lowest value in the range of the ordinal type passed as parameter
`High`	Returns the highest value in the range of the ordinal data type

note C and C++ programmers should notice that the two versions of the `Inc` procedure, with one or two parameters, correspond to the ++ and += operators (the same holds for the `Dec` procedure which corresponds to the -- and -= operators). The Object Pascal compiler optimizes these increment and decrement operations, similarly to the way C and C++ compilers do.

Notice that some of these routines are automatically evaluated by the compiler and replaced with their value. For example, if you call `High(X)` where `X` is defined as an `Integer`, the compiler replaces the expression with the highest possible value of the `Integer` data type.

In the IntegersTest I've added an event with a few of these ordinal type functions:

```
var
  n: UInt16;
begin
```

```
n := Low (UInt16);
Inc (n);
Show (IntToStr (n));
Inc (n, 10);
Show (IntToStr (n));
if Odd (n) then
  Show (IntToStr (n) + ' is odd');
```

This is the output you should see:

```
1
11
11 is odd
```

You can change the data type form Uint16 to Integer or other ordinal types to see how the output changes.

Out-Of-Range Operations

A variable like n above has only a limited range of valid values. If the value you assign to it is negative or too big, this results in an error. There are actually three different types of errors you can encounter with out-of-range operations.

The first type of error is a compiler error, which happens if you assign a constant value (or a constant expression) that is out of range. For example, if you add to the code above:

```
n := 100 + High (n);
```

the compiler will issue the error:

```
[dcc32 Error] E1012 Constant expression violates subrange bounds
```

The second scenario takes place when the compiler cannot anticipate the error condition, because it depends on the program flow. Suppose we write (in the same piece of code):

```
Inc (n, High (n));
Show (IntToStr (n));
```

The compiler won't trigger an error because there is a function call, and the compiler doesn't know its effect in advance (and the error would also depend on the initial value of `n`). In this case there are two possibilities. By default, if you compile and run this application, you'll end up with a completely illogical value in the variable (in this case the operation will result in subtracting 1!). This is the worst possible scenario, as you get no error, but your program is not correct.

What you can do (and it is highly suggested to do) is to turn on a compiler option called “Overflow checking”, which will guard against a similar *overflow* operation and raise an error, in this specific case “Integer overflow”. I've enabled this check in the IntegersTest demo, so you'll see an error message when you run it.

Boolean

Logical `True` and `False` values are represented using the Boolean type. This is also the type of the condition in conditional statements, as we'll see in the next chapter. The `Boolean` type can only have one of the two possible values `True` and `False`.

note For compatibility with Microsoft's COM and OLE automation, the data types `ByteBool`, `WordBool`, and `LongBool` represent the value True with -1, while the value False is still 0. Again, you should generally ignore these types and avoid all low-level Boolean manipulation and numeric mapping unless absolutely necessary.

Unlike in the C language and some of its derived languages, `Boolean` is an enumerated type in Object Pascal, there is no direct conversion to the value representing the `Boolean`, and you should not abuse direct type casts by trying to convert a `Boolean` to a numeric value. It is true, however, that `Boolean` type helpers include the functions `ToInteger` and `ToString`. I cover enumerated types later in this chapter.

Notice that using `ToString` returns the string with the numeric value of the Boolean variable. As an alternative you can use the `BoolToStr` global function, setting the second parameter to True, to indicate the use of Boolean strings (*'True'* and *'False'*) for the output. (See the section “Char Type Operations” below for an example.)

Characters

Character variable are defined using the Char type. Unlike older versions, the language today uses the Char type to represent double-byte Unicode characters.

note The Windows and Mac version of the compiler still offer the distinction between `AnsiChar` for one byte ANSI characters and `WideChar` for Unicode ones, with the `Char` type defined as an alias of the latter. The recommendation is to focus on `WideChar`, and use the `Byte` data type for single byte elements.

For an introduction to characters in Unicode, including the definition of a code point and that of surrogate pairs (among other advanced topics) you can read Chapter 6. In this section I'll just focus on the core concepts of the Char type.

As I mentioned earlier while covering literal values, constant characters can be represented with their symbolic notation, as in `'k'`, or with a numeric notation, as in `#78`. The latter can also be expressed using the `Chr` system function, as in `Chr (78)`. The opposite conversion can be done with the `Ord` function. It is generally better to use the symbolic notation when indicating letters, digits, or symbols.

When referring to special characters, like the control characters below `#32`, you'll generally use the numeric notation. The following list includes some of the most commonly used special characters:

`#8`	backspace
`#9`	tab
`#10`	newline
`#13`	carriage return
`#27`	escape

Char Type Operations

As other ordinal types, the Char type has several predefined operation you can apply to variables of this type using the dot notation. These operations are defined with an intrinsic record helper, again.

However, the usage scenario is quite different. First, to use this feature you need to *enable* it by referring to the `Character` unit in a uses statement. Second, rather than a few conversion functions, the helper for the Char type includes a couple of dozen Unicode-specific operations, including tests like `IsLetter`, `IsNumber`, and `IsPunctuation`, and conversions like `ToUpper` and `ToLower`. Here is an example taken from the `CharsTest` demo:

```
uses
  Character;
...
var
  ch: Char;
begin
  ch := 'a';
  Show (BoolToStr(ch.IsLetter, True));
  Show (ch.ToUpper);
```

The output of this code is:

```
True
A
```

note The `ToUpper` operation of the Char type helper is fully Unicode enabled. This means that if you pass an accented letter like ù the result will be Ù. Some of the traditional RTL functions are not so smart and work only for plain ASCII characters.

Char as an Ordinal Type

The Char data type is quite large, but it is still an ordinal type, so you can use `Inc` and `Dec` functions on it (to get to the next or previous character or move ahead by a given number of elements, as we have seen in the section “Standard Ordinal Types

Routines") and write `for` loops with a Char counter (more on `for` loops in the next chapter).

Here is a simple fragment used to display a few characters, obtained by increasing the value from a starting point:

```
var
  ch: Char;
  str1: string;
begin
  ch := 'a';
  Show (ch);
  Inc (ch, 100);
  Show (ch);

  str1 := '';
  for ch := #32 to #1024 do
    str1 := str1 + ch;
  Show (str1)
```

The `for` loop of the `CharsTest` application adds a lot of text to the string, making the output is quite long. It starts with the following lines of text:

```
a
A
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
defghijklmnopqrstuvwxyz{|}~
// few more lines omitted...
```

Converting with Chr

We have seen that there is an `Ord` function that returns the numeric value (or Unicode code point) of a character. There is also an opposite function you can to get the character corresponding to a code point, that is the `Chr` special function.

32-bit Characters

Although the default Char type is now mapped to WideChar, it is worth noticing that Delphi also defines a 4-byte character type, `UCS4Char`, defined in the `System` unit as:

```
type
  UCS4Char = type LongWord;
```

This type definition and the corresponding one for `UCS4String` (defined as an array of `UCS4Char`) are little used, but they are part of the language runtime and used in some of the functions of the `Character` unit.

Floating Point Types

While integer numbers of various kinds can be represented with an ordinal set of values, floating point numbers are not ordinal (they have the concept of order, but not the concept of a sequence of elements) and represented by some approximate value, with some error in their representation.

Floating-point numbers comes in various formats, depending on the number of bytes used to represent them and the quality of the approximation. Here is a list of floating-point data types in Object Pascal:

`Single` The smallest storage size is given by `Single` numbers, which are implemented with a 4-byte value. The name indicates a single precision floating point value and the same type is indicated with the name `float` in other languages.

`Double` These are floating-point numbers implemented with 8 bytes. The name indicates a double precision floating point value and is shared by many languages. The `Double` precision is the most commonly used floating point data type and is also an alias of an older Pascal type called `Real`.

`Extended` These are numbers implemented with 10 bytes, but this type is not available on all platforms (on some, like Win64, it reverts back to `Double`). Other languages call this data type `long double`.

These are all floating-point data types with different precision, which correspond to the IEEE standard floating-point representations, and are directly supported by the CPU (or, to be precise, by the FPU, the floating point unit), for maximum speed.

There are also two peculiar non-ordinal numeric data types you can used to represent numbers with a precise, not an approximate representation:

`Comp` Describes very big integers using 8 bytes (which can hold numbers with 18 decimal digits). The idea is to represent large numbers with no loss of precision, unlike the corresponding floating point values.

`Currency` Indicates a fixed-point decimal value with four decimal digits, and the same 64-bit representation as the `Comp` type. As the name implies, the `Currency` data type has been added to handle very precise monetary values, with four decimal places (again with no loss of precision in calculations).

All of these non-ordinal data types don't have the concepts of the `High`, `Low`, or `Ord` function. Real types represent (in theory) an infinite set of numbers; ordinal types represent a fixed set of values.

Why Floating Point Values are Different

Let me explain further. When you have the integer 23 you can determine which is the following value. Integers are finite (they have a determined range and they have an order). Floating point numbers are infinite even within a small range, and have no order: in fact, how many values are there between 23 and 24? And which number follows 23.46? Is it 23.47, 23.461, or 23.4601? That's really impossible to know!

For this reason, whilst it makes sense to ask for the ordinal position of the character `'w'` in the range of the `Char` data type, it makes no sense at all to ask the same question about 7143.1562 in the range of a floating-point data type. Although you can indeed know whether one real number has a higher value than another, it makes no sense to ask how many real numbers exist before a given number (this is the meaning of the `Ord` function).

Another key concept behind floating point values is that their implementation cannot represent all numbers precisely. It is often the case that the result of a calculation you'd expect to be a specific number (at times an integer one), could in fact be an approximate value of it. Consider this code, taken from the `FloatTest` example:

```
var
  s1: Single;
begin
  s1 := 0.5 * 0.2;
  Show (s1.ToString);
```

You would expect the result to be `0.1`, while in fact you'd get something like `0.100000001490116`. This is close to the expected value, but not exactly it. Needless to say, if you round the result, you'll get the expected value. If you use a `Double` variable, instead, the output will be `0.1`, as the `FloatTest` example also shows.

---

note Now I don't have time for an in-depth discussion of floating point math on computers, so I'm cutting this discussion rather short, but if you are interested in this topic from the Object Pascal language perspective, I can recommend you an excellent article from Rudy Velthuis at http://rvelthuis.de/articles/articles-floats.html.

Floating Helpers and the Math Unit

As you can see from the code snippet above, the floating point data types also have class helpers allowing you to apply operations directly to the variables, as if they were objects. In fact, the list of operations for floating point numbers is actually quite long.

This is the list of operations on instances for the `Single` type (with most operations quite obvious from their names, so I've omitted a description):

```
Exponent            Fraction            Mantissa
Sign                Exp                 Frac
SpecialType         BuildUp             ToString
IsNan               IsInfinity          IsNegativeInfinity
IsPositiveInfinity  Bytes               Words
```

The run time library has also a `Math` unit that defines advanced mathematical routines, covering trigonometric functions (such as the `ArcCosh` function), finance (such as the `InterestPayment` function), and statistics (such as the `MeanAndStdDev` procedure). There are a number of these routines, some of which sound quite strange to me, such as the `MomentSkewKurtosis` function (I'll let you find out what this is).

The `Math` unit is very rich in capabilities, but you'll also find many external collections of mathematical functions for Object Pascal.

Simple User-Defined Data Types

Along with the notion of type, one of the great ideas introduced by Wirth in the Pascal language was the ability to define new data types in a program. You can define your own data types by means of *type definitions*, such as subrange types, array types, record types, enumerated types, pointer types, and set types. The most important user-defined data type is the class, which is part of the object-oriented capabilities of the language, covered in the second part of this book.

If you think that type constructors are common in many programming languages, you are right, but Pascal was the first language to introduce the idea in a formal and very precise way. Object Pascal still has some rather unique capabilities, like the definition of subrange, enumerations, and sets, covered in the following sections. More complex data type constructors (like arrays and records) are covered in Chapter 5.

Named vs. Unnamed Types

User-defined data types can be given a name for later use or applied to a variable directly. The convention in Object Pascal is to use a letter `T` prefix to denote any data type, including classes but not limited to them. I strongly suggest you to stick to this rule, even if might not feel natural at first if you are coming from a Java or C# background.

When you give a name to a type, you must do so in a "type" section of your program (you can add as many types as you want in each unit). Below is a simple example of a few type declarations:

```
type
  // subrange definition
  TUppercase = 'A'..'Z';

  // enumerated type definition
  TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  TColorPalette = set of TMyColor;
```

With these types, you can now define some variables:

```
var
  UpSet: TUpperLetters;
  Color1: TMyColor;
```

In the scenario above I'm using a *named* type. As an alternative, the type definition can be used directly to define a variable without an explicit type name, as in the following code:

```
var
  Palette: set of TMyColor;
```

In general, you should avoid using *unnamed* types as in the code above, because you cannot pass them as parameters to routines or declare other variables of the same type. Given the language ultimately resorts to *type name equivalence* rather than structural type equivalence, having a single definition for each type is indeed important. Also remember that type definitions in the `interface` portion of a unit can be seen in the code of any other units by means of a `uses` statement.

What do the type definitions above mean? I'll provide some descriptions for those who are not familiar with traditional Pascal type constructs. I'll also try to underline the differences from the same constructs in other programming languages, so you might be interested in reading the following sections in any case.

Subrange Types

A subrange type defines a range of values within the range of another type (hence the name *subrange*). For example, you can define a subrange of the `Integer` type, from 1 to 10 or from 100 to 1000, or you can define a subrange of the `Char` type with English uppercase characters only, as in:

```
type
  TTen = 1..10;
  TOverHundred = 100..1000;
```

```
  TUppercase = 'A'..'Z';
```

In the definition of a subrange, you don't need to specify the name of the base type. You just need to supply two constants of that type. The original type must be an ordinal type, and the resulting type will be another ordinal type. When you have defined a variable as a subrange, you can then assign it any value within that range. This code is valid:

```
var
  UppLetter: TUpperCase;

begin
  UppLetter := 'F';
```

But this is not:

```
var
  UppLetter: TUpperCase;

begin
  UppLetter := 'e'; // compile-time error
```

Writing the code above results in a compile-time error, *"Constant expression violates subrange bounds."* If you write the following code instead:

```
var
  UppLetter: TUppercase;
  Letter: Char;

begin
  Letter :='e';
  UppLetter := Letter;
```

the compiler will accept it. At run-time, if you have enabled the Range Checking compiler option (in the Compiler page of the Project Options dialog box), you'll get a *Range check error* message, as expected. This is similar to the integer type overflow errors which I described earlier.

I suggest that you turn on this compiler option while you are developing a program, so it'll be more robust and easier to debug, as in case of errors you'll get an explicit message and not an undetermined behavior. You can eventually disable this option for the final build of the program, so that it will run a little faster. However, the increase in speed is almost negligible so I suggest to leave all of these run-time checks turned on, even in a shipping program.

Enumerated Types

Enumerated types (usually referred to as "enums") constitute another user-defined ordinal type. Instead of indicating a range of an existing type, in an enumeration you

list all of the possible values for the type. In other words, an enumeration is a list of (constant) values. Here are some examples:

```
type
  TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
  TSuit = (Club, Diamond, Heart, Spade);
```

Each value in the list has an associated *ordinality*, starting with zero. When you apply the `Ord` function to a value of an enumerated type, you get this "zero-based" value. For example, `Ord (Diamond)` returns 1.

Enumerated types can have different internal representations. By default, Delphi uses an 8-bit representation, unless there are more than 256 different values, in which case it uses the 16-bit representation. There is also a 32-bit representation, which at times is useful for compatibility with C or C++ libraries.

note You can change the default representation of enumerated types, asking for a larger one, by using the `$Z` compiler directive.

Scoped Enumerators

The specific constant values of an enumerated type can be considered to all effects as global constants, and there have been cases of names conflicts among different enumerated values. This is why the language supports scoped enumerations, a feature you can activate using a specific compiler directive, `$SCOPEDENUMS`, and which requires you to refer to the enumerated value using the type name as a prefix:

```
// classic enumerated value
s1 := Club;

// "scoped" enumerated value
s1 := TSuit.Club;
```

note This is exactly how C# invariably works, but in that language enumerations have a slightly different behavior, can have holes in the sequence and have specific values assigned to the various constants.

When this feature was introduced, the default remained the traditional behavior, to avoid breaking existing code. Scoped enumerators, in fact, changes the behavior of enumerations making it compulsory to refer to them with a type prefix.

Having an *absolute* name to refer to enumerated values removes the risk of a conflict, could let you avoid using the initial prefix of the enumerated values as a way to differentiate with other enumerations, and makes the code more readable, even if much longer to write.

As an example, the `IOUtils` unit defines this type:

```
{$SCOPEDENUMS ON}
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

This means you cannot refer to the second value as `soAllDirectories`, but you have to refer to the it with its complete name:

```
TSearchOption.soAllDirectories
```

The FM Platform library uses quite a number of scoped enumerators, as well, requiring the type as a prefix to the actual values.

---

note Enumerated values in Object Pascal libraries often use two or three initials of the type at the beginning of the value, like “so” for Search Options in the example above. When using the type as a prefix, this might seem a bit redundant, but given the commonality of the approach, I don't see it going away any time soon.

Set Types

Set types indicate a group of values, where the list of available values is indicated by the ordinal type the set is based onto. These ordinal types are usually limited, and quite often represented by an enumeration or a subrange. If we take the subrange 1..3, the possible values of the set based on it include only 1, only 2, only 3, both 1 and 2, both 1 and 3, both 2 and 3, all the three values, or none of them.

A variable usually holds one of the possible values of the range of its type. A set-type variable, instead, can contain none, one, two, three, or more values of the range. It can even include all of the values. Here is an example of a set:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
  TSuits = set of TSuit;
```

Now I can define a variable of this type and assign to it some values of the original type. To indicate some values in a set, you write a comma-separated list, enclosed within square brackets. The following code shows the assignment to a variable of several values, a single value, and an empty value:

```
var
  Cards1, Cards2, Cards3: TSuits;

begin
  Cards1 := [Club, Diamond, Heart];
  Cards2 := [Diamond];
  Cards3 := [];
```

In Object Pascal, a set is generally used to indicate several nonexclusive flags. For example a value based on a set type is the style of a font. Possible values indicate a

bold, italic, underline, and strike-through font. Of course the same font can be both italic and bold, have no attributes, or have them all. For this reason it is declared as a set. You can assign values to this set in the code of a program as follows:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles active
```

Set Operators

We have seen that sets are a very Pascal-specific user defined data type. That's why the set operators are worth a specific coverage. They include union (`+`), difference (`-`), intersection (`*`), membership test (`in`), plus some relational operators. To add an element to a set, you can make the union of the set with another one that has only the elements you need. Here's an example related to font styles:

```
// add bold
Style := Style + [fsBold];

// add bold and italic, but remove underline if present
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

As an alternative, you can use the standard `Include` and `Exclude` procedures, which are much more efficient (but cannot be used with component properties of the set type):

```
Include (Style, fsBold);
Exclude (Style, fsItalic);
```

Expressions and Operators

We have seen that you can assign to a variable a type-compatible literal value, a constant value, or the value of another variable. In many cases, what you assign to a variable is the result of an expression, involving one or more values and one or more operators. Expressions are another core element of the language.

Using Operators

There isn't a general rule for building expressions, since they mainly depend on the operators being used, and Object Pascal has a number of operators. There are logical, arithmetic, Boolean, relational, and set operators, plus some other special ones:

```
// sample expressions
```

```
20 * 5       // multiplication
30 + n       // addition
a < b        // less than comparison
- 4          // negative value
c = 10       // test for equality (like == in C syntax)
```

Expressions are common to most programming languages, and most operators are the same. An expression is any valid combination of constants, variables, literal values, operators, and function results. Expressions can be used to determine the value to assign to a variable, to compute the parameter of a function or procedure, or to test for a condition. Every time you are performing an operation on the value of an identifier, rather than using an identifier by itself, you are using an expression.

note The result of an expression is generally stored in a temporary variable of the proper data type automatically generated by the compiler on your behalf. You might want to use an explicit variable when you need to compute the same expression more than once in the same code fragment. Notice that complex expressions might require multiple temporary variables to store intermediate results, again something the compiler takes care of for you and you can generally ignore.

Showing the Result of an Expression

If you want to make a few experiments with expressions, there is nothing better than writing a simple program. As for most of the initial demos of this book, create a simple program based on a form, and use the custom `Show` function to display something to the user. In case the information you want to show is a not a string message but number or a boolean logical value, you need to convert it, for example calling the `IntToStr` or `BoolToStr` function.

note In Object Pascal parameters passed to a function or procedures are enclosed in parenthesis. Some other languages (notably Rebol and, to some extent, Ruby) let you pass parameters simply by writing them after the function or procedure name. Getting back to Object Pascal, nested functions calls use nested parenthesis, like in the code below.

Here is a sample code snippet from the demo program `ExpressionsTest`:

```
Show (IntToStr (20 * 5));
Show (IntToStr (30 + 222));
Show (BoolToStr (3 < 30, True));
Show (BoolToStr (12 = 10, True));
```

The output is this code snippet is quite trivial:

```
100
252
True
False
```

I've provided this demo as a skeleton for you to try out different types of expressions and operators, and see the corresponding output.

note Expressions you write in Object Pascal are parsed by the compiler and generate assembly code. If you want to change one of these expressions, you need to change the source code and recompile the application. The system libraries, however, have support for dynamic expressions calculated at runtime, a features tied to reflection and covered in Chapter 16.

Operators and Precedence

Expressions are made of operators applied to values. As I mentioned, most operators are shared among the various programming languages are are quite intuitive, such as the basic match and comparison operators. In this section I'll highlight only specific elements of Object Pascal operators.

You can see a list of the operators of the language below, grouped by precedence and compared to operators in C#, Java, and Objective-C (and most languages based on the C language syntax, anyway).

Relational and Comparison Operators (Lowest Precedence)

`=`	Test whether equal (in C this is `==`)
`<>`	Test whether not equal (in C this is `!=`)
`<`	Test whether less than
`>`	Test whether greater than
`<=`	Test whether less than or equal to, or a subset of a set
`>=`	Test whether greater than or equal to, or a superset of a set
`in`	Test whether the item is a member of the set
`is`	Test whether an object is compatible with a given type (covered in Chapter 8) or implements a given interface (covered in Chapter 11)

Additive Operators

`+`	Arithmetic addition, set union, string concatenation, pointer offset addition
`-`	Arithmetic subtraction, set difference, pointer offset subtraction
`or`	Boolean or bitwise or (in C this is either `||` or `|`)
`xor`	Boolean or bitwise exclusive or (in C bitwise exclusive or is `^`)

Multiplicative and Bitwise Operators

`*`	Arithmetic multiplication or set intersection
`/`	Floating-point division
`div`	Integer division (in C this also uses `/`)

`mod`	Modulo (the remainder an of integer division) (in C this is `%`)
`as`	Allows a type-checked conversion at runtime (covered in Chapter 8)
`and`	Boolean or bitwise and (in C this is either `&&` or `&`)
`shl`	Bitwise left shift (in C this is `<<`)
`shr`	Bitwise right shift (in C this is `>>`)

Unary Operators (Highest Precedence)

`@`	Memory address of a variable or function (returns a pointer, in C this is `&`)
`not`	Boolean or bitwise not (in C this is `!`)

Different from many other programming languages, the `and` and `or` operators have higher precedence than comparison ones. So if you write:

```
a < b and c < d
```

the compiler will do the `and` operation first, generally resulting in a compiler error. If you want to test both comparisons, you should enclose each of the < expressions in parentheses:

```
(a < b) and (c < d)
```

For math operations, instead, the common rules apply, with multiplication and division taking precedence over addition and subtraction. The first two expressions below are equivalent, while the third is different:

```
10 + 2 * 5      // result is 20
10 + (2 * 5)    // result is 20
(10 + 2) * 5    // result is 60
```

Some of the operators have different meanings when used with different data types. For example, the + operator can be used to add two numbers, concatenate two strings, make the union of two sets, and even add an offset to a pointer (if the specific pointer type has *pointer math* enabled):

```
10 + 2 + 11
10.3 + 3.4
'Hello' + ' ' + world'
```

However, you cannot add two characters, as is possible in C.

An unusual operator is `div`. In Object Pascal, you can divide any two numbers (real or integers) with the `/` operator, and you'll invariably get a real-number result. If you need to divide two integers and want an integer result, use the `div` operator instead. Here are two sample assignments (this code will become clearer as we cover data types in the next chapter):

```
realValue := 123 / 12;
integerValue := 123 div 12;
```

To make sure the integral division has no remainder, you can use the `mod` operator and check if the result is zero, like in the following Boolean expression:

```
(x mod 12) = 0
```

Date and Time

While there was no native type for date and time in the early versions of the Pascal language, Object Pascal has a native type for date and time. It uses a floating-point representation to handle date and time information. To be more precise the `System` unit defines a specific `TDateTime` data type for that purpose.

This is a floating-point type, because it must be wide enough to store years, months, days, hours, minutes, and seconds, down to millisecond resolution in a single variable:

- Dates are stored as the number of days since 1899-12-30 (with negative values indicating dates before 1899) in the integer part of the `TDateTime` value
- Times are stored as fractions of a day in the decimal part of the value

note In case you are wondering were that strange date comes from, there is a rather long story behind it tied to Excel and dates representations in Windows applications. The idea was to consider day number 1 as the first of January 1900, so that New Year's eve of 1899 would have been day number 0. However, the original developer of that date representation duly forgot that year 1900 wasn't a leap year, and so calculations were later adjusted by 1 day, turning the first of January 1900 into day number 2.

As mentioned, `TDateTime` is not a predefined type the compiler understands, but it is defined in the `System` unit as:

```
type
  TDateTime = type Double;
```

note The `System` unit could be somehow considered almost as part of the core language, given it is always automatically included in each compilation, without a uses statement (actually adding the `System` unit to a `uses` section will cause a compilation error). Technically, though, this unit is considered as the core part of the run-time library (RTL), and it will be covered in Chapter 18.

There are also two related types to handle the time and date portions of a `TDateTime` structure, defined as `TDate` and `TTime`. These specific types are aliases of the full `TDateTime`, but they are treated by system functions trimming the unused part of the data.

Using date and time data types is quite easy, because Delphi includes a number of functions that operate on this type. There are several core functions in the `SysUtils` unit, and many specific functions in the `DateUtils` unit (which despite the name includes also functions for manipulating time).

Here you can find a short list of commonly used date/time manipulation functions:

`Now`	Returns the current date and time into a date/time value.
`Date`	Returns only the current date.
`Time`	Returns only the current time.
`DateTimeToStr`	Converts a date and time value into a string, using default formatting; to have more control on the conversion use the `FormatDateTime` function instead.
`DateToStr`	Converts the date portion of a date/time value into a string.
`TimeToStr`	Converts the time portion of a date/time value into a string.
`FormatDateTime`	Formats a date and time using the specified format; you can specify which values you want to see and which format to use by providing a complex format string.
`StrToDateTime`	Converts a string with date and time information to a date/time value, raising an exception in case of an error in the format of the string. Its companion function, `StrToDateTimeDef` returns the default value in case of an error rather than raising an exception.
`DayOfWeek`	Returns the number corresponding to the day of the week of the date/time value passed as parameter.
`DecodeDate`	Retrieves the year, month, and day values from a date value.
`DecodeTime`	Retrieves the hours, minutes, seconds, and milliseconds from a date value.
`EncodeDate`	Turns year, month, and day values into a date/time value.
`EncodeTime`	Turns hour, minute, second, and millisecond values into a date/time value.

To show you how to use this data type and some of its related routines, I've built a simple example, named `TimeNow`. When the program starts it automatically computes and displays the current time and date.

```
var
  StartTime: TDateTime;
begin
  StartTime := Now;
  Show ('Time is ' + TimeToStr (StartTime));
  Show ('Date is ' + DateToStr (StartTime));
```

The first statement is a call to the `Now` function, which returns the current date and time. This value is stored in the `StartTime` variable.

note When an Object Pascal function is called with no parameters there is no need to type the empty parentheses unlike with the C style languages.

The next two statements display the time portion of the `TDateTime` value, converted into a string, and the date portion of the same value. This is the output of the program:

```
Time is 6:33:14 PM
Date is 10/7/2014
```

To compile this program you need to refer to functions that are part of the `SysUtils` unit (a short name for "system utilities"). Besides calling `TimeToStr` and `DateToStr` you can use the more powerful `FormatDateTime` function.

Notice that time and date values are transformed into strings depending on the system's international settings. The date and time formatting information is read from the system, depending on the operating system and the locale, populating a `TFormatSettings` data structure. If you need customized formatting, you can create a custom structure of that type and pass it as parameter to most date time formatting functions.

note The `TimeNow` project has also a second button you can use to enable a timer. This is component that executes an event handler automatically over time (you specify the interval). In the demo, if you enable the timer you'll see the current time added to the list every second. A more useful user interface would be to update a label with the current time every second, basically building a clock.

Typecasting and Type Conversions

As we have seen, you cannot assign a variable of one data type to one of a different type. The reason is, depending on the actual representation of the data, you might end up with something meaningless.

Now, this is not true for each and every data type. Numerical types, for example, can always be promoted safely. "Promoted" here means you can always safely assign a value to a type with a larger representation. So you can assign a word to an integer, and an integer to an Int64 value. The opposite operation, called "demotion", is allowed by the compiler but it will issue a warning, because you might end up with partial data. Other automatic conversions are one way only: For example, you can assign an integer to a floating point number, but the opposite operation is illegal.

There are scenarios you want to change the type of a value and the operation makes sense. When you need to do this, there are two choices. One is to perform a direct type cast, which will copy the physical data and might result in a proper conversion or a not depending on the types. When you perform a typecast, you are telling the compiler “I know what I'm doing, let me go for it”. So, better if you really know what you are doing, as you are losing the compiler safety net.

Type casting uses a simple functional notation, with the name of the destination data type used as a function:

```
var
  N: Integer;
  C: Char;
  B: Boolean;

begin
  N := Integer ('X');
  C := Char (N);
  B := Boolean (N);
```

You can safely typecast between data types having the same size (that is the same number of bytes to represent the data – unlike in the code snippet above!). It is usually safe to typecast between ordinal types, but you can also typecast between pointer types (and also objects) as long as you know what you are doing.

Direct type casting is a dangerous programming practice, because it allows you to access a value as if it represented something else. Since the internal representations of data types generally do not match (and might even change depending on the target platform), you risk accidentally creating hard-to-track errors. For this reason, *you should generally avoid type casting*.

The second choice to assign a variable to one of a different type is to use a type conversion function. A list of functions allowing you to convert between various basic types is summarized below (and I've already used some of these functions in the demos of this chapter):

`Chr`	Converts an ordinal number into a character.
`Ord`	Converts an ordinal-type value into the number indicating its order.
`Round`	Converts a real-type value into an Integer-type value, rounding its value (also see the following note).
`Trunc`	Converts a real-type value into an Integer-type value, truncating its value.
`Int`	Returns the Integer part of the floating-point value argument.
`FloatToDecimal`	Converts a floating-point value to record including its decimal representation (exponent, digits, sign).
`FloatToStr`	Converts a floating-point value to its string representation using default formatting.

`StrToFloat`	Converts a string to a floating-point value.

note The implementation of the Round function is based on the native implementation offered by the CPU. Moderns processors generally adopts the so-called "Banker's Rounding", which rounds middle values (such as 5.5 or 6.5) up and down depending whether they follow an odd or an even number. There are other rounding functions, such as `RoundTo`, that offer you more control on the actual operation.

As mentioned earlier in this chapter, some of these conversion functions are available also as direct operations on the data type (thanks to the type helper mechanism). While there are classic conversions like `IntToStr`, you can apply the `ToString` operation to most numeric types to convert them to a string representation. There are many conversions you can apply directly to variables using type helpers, and that should be your preferred coding style.

Some of these routines work on the data types that we'll discuss in the following sections. Notice that the table doesn't include routines for special types (such as `TDateTime` or variant) or routines specifically intended for formatting more than conversion, like the powerful `Format` and `FormatFloat` routines.

03: language statements

If the concept of data type was one of the breakthrough of the Pascal programming language when it was first invented, the other side is represented by the code or programming statements. At that time, this idea was clarified by Nicklaus Wirth's outstanding book “Algorithms + Data Structures = Programs”, published by Prentice Hall in February 1976 (a classic book, still reprinted and available). While this book predates object-oriented programming by many years, it can be considered one of the foundations of modern programming, based on a strong notion of data type, and in this way a foundation of the concepts that lead to object-oriented programming languages.

Statements of the programming language are based on keywords (covered in Chapter 1) and other elements which allow you to indicate to a compiler a sequence of operations to perform. Statements are often enclosed in procedures or functions, as we'll start to see in more detail in the next chapter. For now, we'll just focus on the basic types of instructions you can write to create a program.

As we saw in Chapter 1 (in the section covering white space and code formatting), the actual program code can be written quite freely. I also covered comments and

some other special elements, but never fully introduced some core concepts, like a programming statement.

Simple and Compound Statements

Programming instructions are generally called *statements*. A program block can be made of a *several* statements. There are two types of statements, simple and compound.

A statement is called *simple* when it doesn't contain any other sub-statements. Examples of simple statements are assignment statements and procedure calls. In Object Pascal simple statements are *separated* by a semicolon:

```
X := Y + Z;  // assignment
Randomize;   // procedure call
...
```

To define a *compound* statement, you can include one of more statements within the keywords `begin` and `end`, which act as brackets. A compound statement can appear anywhere a simple Object Pascal statement can appear. Here is an example:

```
begin
  A := B;
  C := A * 2;
end;
```

The semicolon after the last statement of the compound statement (that is, before the `end`) isn't required, as in the following:

```
begin
  A := B;
  C := A * 2
end;
```

Both versions are correct. The first version has a useless (but harmless) final semicolon. This semicolon is, in fact, a null statement or an empty statement; that is, a statement with no code. This is significantly different from many other programming languages (like those based on the C syntax), in which the semicolon is a statement *terminator* (not a separator) and is always required at the end of a statement.

Notice that, at times, a null statement can be specifically used inside loops or in other particular cases in place of an actual statement, as in:

```
while condition_with_side_effect do
  ; // null or empty statement
```

Although these final semicolons serve no purpose, most developers tend to use them and I suggest you to do the same. Sometimes after you've written a couple of lines you might want to add one more statement. If the last semicolon is missing you have to remember to add it, so it is usually better to add it in the first place. As we'll see right away, there is an exception to this rule of adding extra semicolons, and that is when the next element is an `else` statement inside a condition.

The If Statement

A conditional statement is used to execute either one of the statements it contains or none of them, depending on a specific test (or condition). There are two basic flavors of conditional statements: `if` statements and `case` statements.

The `if` statement can be used to execute a statement only if a certain condition is met (`if-then`) or to choose between two different alternatives (`if-then-else`). The condition is defined with a Boolean expression.

A simple Object Pascal example, called `IfTest`, will demonstrate how to write conditional statements. In this program we'll use a checkbox to get user input, by reading its `IsChecked` property (and storing it to a temporary variable, although this isn't strictly required, as you could directly check the property value in the conditional expression):

```
var
  isChecked: Boolean;
begin
  isChecked := CheckBox1.IsChecked;
  if isChecked then
    Show ('Checkbox is checked');
```

If the checkbox is checked, the program will show a simple message. Otherwise nothing happens. By comparison, the same statement using the C language syntax will look like the following (where the conditional expression *must* be enclosed within parentheses):

```
if (isChecked)
 Show ("Checkbox is checked");
```

Some other languages have the notion of an `endif` element to allow you to write multiple statements, where in Object Pascal syntax the conditional statement is a single statement by default. You use a `begin-end` block to execute more than one statement as part of the same condition.

If you want to do different operations depending on the condition, you can use an `if-then-else` statement (and in this case I used a direct expression to read the checkbox status):

```
// if-then-else statement
if CheckBox1.IsChecked then
  Show ('Checkbox is checked')
else
  Show ('Checkbox is not checked');
```

Notice that you cannot have a semicolon after the first statement and before the `else` keyword or the compiler will issue a syntax error. The reason is that the `if-then-else` statement is a single statement, so you cannot place a semicolon in the middle of it.

An `if` statement can be quite complex. The condition can be turned into a series of conditions (using the `and`, `or`, and `not` Boolean operators), or the `if` statement can nest a second `if` statement. Beside nesting `if` statements, when there are multiple distinct conditions, it is common to have consecutive statements `if-then-else-if-then`. You can keep chaining as many of these `else-if` conditions as you want.

The third button of the `IfTest` example demonstrates these scenarios, using the first character of an edit box (which might be missing, hence the external test) as input:

```
var
  aChar: Char;
begin
  // multiple nested if statements
  if Edit1.Text.Length > 0 then
  begin
    aChar := Edit1.Text.Chars[0];

    // checks for a lowercase char (two conditions)
    if (aChar >= 'a') and (aChar <= 'z') then
      Show ('char is lowercase');

    // follow up conditions
    if aChar <= Char(47) then
      Show ('char is lower symbol')
    else if (aChar >= '0') and (aChar <= '9') then
      Show ('char is a number')
    else
      Show ('char is not a number or lower symbol');
  end;
```

Look at the code carefully and run the program to see if you understand it (and play with similar programs you can write to learn more). You can consider more options and Boolean expressions and increase the complexity of this small example, making any test you like.

Case Statements

If your `if` statements become very complex, at times you can replace them with `case` statements. A `case` statement consists of an expression used to select a value and a list of possible values, or a range of values. These values are constants, and they must be unique and of an ordinal type. Eventually, there can be an `else` statement that is executed if none of the values you specified correspond to the value of the selector. While there isn't a specific `endcase` statement, a `case` is always terminated by an `end` (which in this case isn't a block terminator, as there isn't a matching `begin`).

note Creating a case statement requires an enumerated value. A case statement based on a string value is currently not allowed. In that case you need to use nested if statements or a different data structure, like a dictionary (as I show later in the book in Chapter 14).

Here is an example (part of the `CaseTest` project), which uses as input the integral part of the number entered in a NumberBox control, a numeric input control:

```
var
  number: Integer;
  aText: string;
begin
  number := Trunc(NumberBox1.Value);
  case number of
    1: aText := 'One';
    2: aText := 'Two';
    3: aText := 'Three';
  end;
  if aText <> '' then
    Show(aText);
```

Another example is the extension of the previous complex if statement, turned into a number of different conditions of a `case` test:

```
case aChar of
  '+' : aText := 'Plus sign';
  '-' : aText := 'Minus sign';
  '*', '/': aText := 'Multiplication or division';
  '0'..'9': aText := 'Number';
  'a'..'z': aText := 'Lowercase character';
  'A'..'Z': aText := 'Uppercase character';
  #12032..#12255: aText := 'Kangxi Radical';
else
  aText := 'Other character: ' + aChar;
end;
```

note As you can see in the previous code snippet, a range of values is defined with the same syntax of a subrange data type. Multiple values for a single branch, instead, are separated by a comma. For the Kangxi Radical section I've used the numerical value rather than the actual characters, because most of the fixed-size fonts used by the IDE editor won't display properly symbols like ⼀ (the first element of the group).

It is considered good practice to include the `else` part to signal an undefined or unexpected condition. A `case` statement in Object Pascal selects one execution path, it doesn't position itself at an entry point. In other word, it will execute the statement or block after the colon of the selected value and it will skip to the next statement after the case.

This is very different from the C language (and some of its derived languages) which treat branches of a `switch` statement as entry points and will execute all following statements unless you specifically use a `break` request (although this is a specific scenario in which Java and C# actually differ in their implementation). The C language syntax is like the following:

```
switch (aChar) {
  case '+': Text = "plus sign"; break;
  case '-': Text = "minus sign"; break;
  ...
  default: Text = "unknown"; break;
}
```

The For Loop

The Object Pascal language has the typical repetitive or looping statements of most programming languages, including `for`, `while`, and `repeat` statements, plus the more modern `for-in` (or *for-each*) cycle. Most of these loops will be familiar if you've used other programming languages, so I'll only cover them briefly (indicating the key differences from other languages).

The `for` loop in Object Pascal is strictly based on a counter, which can be either increased or decreased each time the loop is executed. Here is a simple example of a `for` loop used to add the first ten numbers (part of the `ForTest` demo).

```
var
  Total, I: Integer;
begin
  Total := 0;
  for I := 1 to 10 do
    Total := Total + I;
```

```
  Show(Total.ToString);
```

For those curious, the output is `55`. The `for` loop in Pascal is less flexible than in other languages (it is not possible to specify an increment different than one), but it is simple and easy to understand. As a comparison, this is the same `for` loop written in the C language syntax:

```
int total = 0;
for (int i = 1; i <= 10; i++) {
  total = total + i;
}
```

In these languages, the increment is an expression you can use to specify any kind of sequence, which can lead to some really unreadable code as the following:

```
int total = 0;
for (int i = 10; i > 0; total += i--) {
  ..
}

            Speed before comprehension ;-)
]
```

In Object Pascal, instead, you can only use a single step increment. If you want to test for a more complex condition, or if you want to provide a customized counter, you'll need to use a `while` or `repeat` statement, instead of a `for` loop.

The only alternative to single increment is single decrement, or a reverse `for` loop:

```
var
  Total, I: Integer;
begin
  Total := 0;
  for I := 10 downto 1 do
    Total := Total + I;
```

note Reverse counting is useful, for example, when you are affecting a list-based data structure you are looping through. When deleting some elements, you often go backwards, as with a forward loop you might affect the sequence you are operating onto (that is, if you delete the third element of a list, the fourth element becomes the third: now you are on the third, move to the next one (the fourth) but you are actually operating on what was the fifth element, skipping one).

In Object Pascal the counter of a `for` loop doesn't need to be a number. It can be a value of any ordinal type, such as a character or an enumerated type. This helps you to write more readable code. Here is an example with a `for` loop based on the `Char` type:

```
var
  aChar: Char;
begin
  for aChar := 'a' to 'z' do
    Show (aChar);
```

This code (part of the `ForTest` program) shows all of the letters of the English alphabet, one in a separate line of the output Memo control.

note I've already shown a similar demo, but based on an integer counter, as part of the `CharsTest` example of Chapter 2. In that case, though, the chars were concatenated in a single output string.

Here is another code snippet that shows a `for` loop based on a custom enumeration:

```
type
  TSuit = (Club, Diamond, Heart, Spade);

var
  ASuit: TSuit;
begin
  for ASuit := Club to Spade do
    ...
```

This last loop that cycles on all of the elements of the data type, could also be written to explicitly operate on each element of the type rather than specifically indicating the first and the last one, by writing:

```
  for ASuit := Low (TSuit) to High (TSuit) do
```

In a similar way, it is quite common to write for loop on all elements of a data structure, such as a string. In this case you can use this code (also part of the ForTest project):

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low (S) to High (S) do
    Show(S[I]);
```

This code can be error prone, as you need to remember how to query for the first and last element of the structure. This is why in a similar scenario, it is better to use a `for-in` loop, a special-purpose `for` loop discussed in the next section.

note How the compiler treats direct access to the string using the `[]` operators and determines the lower and upper bounds of a string is a rather complex topic in Object Pascal. While this will be covered in Chapter 6, the code above (and all other snippets based on strings) work in all possible scenarios.

The for-in Loop

Microsoft's Visual Basic has always had a specific loop construct for cycling over all of the elements of a list or collection, called *for each*. The same idea was later introduced in C#, where the `foreach` mechanism is quite open and based on the use of the `IEnumerator` interface and a standard coding pattern, while Java uses the `for` keyword to express both types of for loops.

Recent versions of Object Pascal have a similar loop called `for-in`. In this `for` loop the cycle operates on each element of an array, a list, a string, or some other type of container. Object Pascal doesn't require the `IEnumerator` interface, but the internal implementation is somewhat similar.

---

note You can find the technical details of how to support the `for-in` loop in a class, adding custom enumeration support, in Chapter 10.

Let's start with a very simple container, a string, which can be seen as a collection of characters. We have seen at the end of the previous section how to use a for loop to operate on all elements of a string. The same exact effect can be obtained with the following `for-in` loop based on a string, where the `Ch` variable receives as value each of the string elements in turn:

```
var
  S: string;
  Ch: Char;
begin
  S := 'Hello world';
  for Ch in S do
    Show(Ch);
```

This snippet is also part of the ForTest example. The advantage over using a traditional `for` loop is that you don't need to remember which is the first element of the string and how to extract the position of the last one. This loop is easier to write and maintain and has a similar efficiency.

The `for-in` loop can be used to access to the elements of the several different data structures:

- Characters in a string (see the previous code snippet)
- Active values in a set
- Items in a static or dynamic array, including two-dimensional arrays (covered in Chapter 5)
- Objects referenced by classes with `GetEnumerator` support, including many predefined ones like strings in a string list, elements of the various container

classes, the components owned by a form, and many others. How to implement this will be discussed in Chapter 10.

Now it is a little difficult at this point in the book to cover these advanced usage patterns, so I'll get back to examples of this loop later in the book.

note The `for-in` loop in some languages (for example JavaScript) has a bad reputation for being very slow to run. This is not the case in Object Pascal, where is takes about the same time of a corresponding standard `for` loop. To prove this, I've added to the LoopsTest example some timing code, which first creates a string of 30 million elements and later scans it with both types of loops (doing a very simple operation at each iteration. The difference in speed is about 10% in favor of the classic for loop (62 milliseconds vs. 68 milliseconds on my Windows machine).

While and Repeat Statements

The idea behind the `while-do` and the `repeat-until` loops is repeating the execution of a code block over and over until a given condition is met. The difference between these two loops is that condition is checked at the beginning or at the end of the loop. In other words, the code block of the `repeat` statement is always executed at least once.

note Most other programming languages have only one type of open looping statement, generally called and behaving like a while loop. The C language syntax has the same two option as the Pascal syntax, with the `while` and `do-while` cycles. Notice, thought, that they use the same logical condition, differently from the `repeat-until` loop that has a reverse condition.

You can easily understand why the `repeat` loop is always executed at least once, by looking at a simple code example:

```
while (I <= 100) and (J <= 100) do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;

repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

note You will have noticed that in both the `while` and `repeat` conditions I have enclosed the "sub-conditions" in parentheses. It is necessary in this case, as the compiler will execute `or` before performing the comparisons (as I covered in the section about operators of Chapter 2).

If the initial value of `I` or `J` is greater than 100, the `while` loop is completely skipped, while statements inside the `repeat` loop are executed once anyway.

The other key difference between these two loops is that the `repeat-until` loop has a *reversed* condition. This loop is executed *as long as the condition is not met*. When the condition is met, the loop terminates. This is the opposite of a `while-do` loop, which is executed while the condition is true. For this reason I had to reverse the condition in the code above to obtain a similar effect.

note The "reverse condition" is formally known as the "De Morgan's" laws (described, for example, on Wikipedia at http://en.wikipedia.org/wiki/De_Morgan%27s_laws).

Examples of Loops

To explore some more details of loops, let's look at a small practical example. The LoopsTest program highlights the difference between a loop with a fixed counter and a loop with an open counter. The first fixed counter loop, a `for` loop, displays numbers in sequence:

```
var
  I: Integer;
begin
  for I := 1 to 20 do
    Show ('Number ' + IntToStr (I));
end;
```

The same could have been obtained also with a `while` loop, with an internal increment of one (notice you increment the value after using the current one). With a `while` loop, however, you are free to set a custom increment, for example by 2:

```
var
  I: Integer;
begin
  I := 1;
  while I <= 20 do
  begin
    Show ('Number ' + IntToStr (I));
    Inc (I, 2)
  end;
end;
```

This code shows all of the odd numbers from one to 19. These loops with fixed increments are logically equivalent and execute a predefined number of times. This is not always the case. There are loops that are more undetermined in their execution, depending for example on external conditions.

note When writing a `while` loop you must always consider the case the condition is never met. For example, if you write the loop above but forget to increment the loop counter, this will result into an infinite loop (which will stall the program forever, consuming the CPU at 100%, until the operating system kills it).

To show an example of a less deterministic loop I've written a `while` loop still based on a counter, but one that is increased randomly. To accomplish this, I've called the `Random` function with a range value of 100. The result of this function is a number between 0 and 99, chosen randomly. The series of random numbers control how many times the while loop is executed:

```
var
  I: Integer;
begin
  Randomize;
  I := 1;
  while I < 500 do
  begin
    Show ('Random Number: ' + IntToStr (I));
    I := I + Random (100);
  end;
end;
```

If you remember to add a call the `Randomize` procedure, which resets the random number generator at a different point for each program execution, each time you run the program, the numbers will be different. The following is the output of two separate executions, displayed side by side:

```
Random Number: 1                        Random Number: 1
Random Number: 40                       Random Number: 47
Random Number: 60                       Random Number: 104
Random Number: 89                       Random Number: 201
Random Number: 146              Random Number: 223
Random Number: 198              Random Number: 258
Random Number: 223              Random Number: 322
Random Number: 251              Random Number: 349
Random Number: 263              Random Number: 444
Random Number: 303              Random Number: 466
Random Number: 349
Random Number: 366
Random Number: 443
Random Number: 489
```

Notice that not only are the generated numbers different each time, but so is the number of items. This `while` loop is executed a random numbers of times. If you

execute the program several times in a row, you'll see that the output has a different number of lines.

Breaking the Flow with Break and Continue

Despite the differences, each of the loops lets you execute a block of statements a number of times, based on some rules. However, there are scenarios you might want to add some additional behavior. Suppose, as an example, you have a for loop where you search for the occurrence of a given letter (this code is part of the `FlowTest` demo):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
      Found := True;
```

At the end you can check for the value of `found` to see if the given letter was part of the string. The problem is that the program keeps repeating the loop and checking for the given character even after it found an occurrence of it (which would be an issue with a very long string).

A *classic* alternative would be to turn this into a while loop and check for both conditions (the loop counter and the value of `Found`):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  I := Low (S);
  while not Found and (I <= High(S)) do
  begin
    if (S[I]) = 'o' then
      Found := True;
    Inc (I);
  end;
```

While this code is logical and readable, there is more code to write, and if the conditions become multiple and more complex, combining all of the various options would make the code very complicated.

That's why the language (or, to be more precise, its runtime support) has system procedures that let you alter the standard flow of a loop's execution:

- The `Break` procedure interrupts a loop, jumping directly to the first statement following it, skipping any further execution
- The `Continue` procedure jumps to the loop test or counter increment, continuing with the next iteration of the loop (unless the condition is no longer true or the counter has reached its highest value)

Using the `Break` operation, we can modify the original loop for matching a character as follows:

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
    begin
      Found := True;
      Break; // jumps out of the for loop
    end;
```

Two more system procedures, `Exit` and `Halt`, let you immediately return from the current function or procedure or terminate the program. I'll cover `Exit` in the next chapter, while there is basically no reason to ever call `Halt` (so I won't really discuss it in the book).

Here Comes Goto? No Way

There is actually more to breaking the flow than the four system procedures above. The original Pascal language counted among its features the *infamous* goto statement, letting you attach a label to any line of the source code, and jump to that line from another location. Differently from conditional and looping statements, which reveal why you want to diverge from a sequential code flow, goto statements generally look like erratic jumps, and are really completely discouraged. Did I mention they are not supported in Object Pascal? No, I didn't, nor am I going to show you a code example. To me `goto` is long gone.

---

note There are other language statements I haven't covered so far but are part of the language definition. One of them is the with statement, which is specifically tied to records, so I'll cover it in Chapter 5. With is another “debated” language feature, but not hated as much as `goto`.

04: procedures and functions

Another important idea emphasized in the Object Pascal language (along with similar features of the C language) is the concept of the routine, basically a series of statements with a unique name, which can be activated many times. Routines (or functions) are called by their name. This way you avoid having to write the same code over and over, and will have a single version of the code used in many places through the program. From this point of view, you can think of routines as a basic code encapsulation mechanism.

Procedures and Functions

In Object Pascal, a routine can assume two forms: a procedure and a function. In theory, a procedure is an operation you ask the computer to perform, a function is a computation returning a value. This difference is emphasized by the fact that a function has a result, a return value, or a type, while a procedure doesn't. The C language syntax provides for a single mechanism, functions, and in this language a procedure is a function with a `void` (or null) result.

Both types of routines can have multiple parameters of specified data types. As we'll see later, procedures and functions are also the basis of the methods of a class, and also in this case the distinction between the two forms remains. In fact, differently from C, C++, Java, C#, or JavaScript, you need to one of these two keywords when declaring a function or a method.

In practice, even if there are two separate keywords, the difference between functions and procedures is very limited: you can call a function to perform some work and then ignore the result (which might be an optional error code or something like that) or you can call a procedure which passes back a result in one of the parameters (more on reference parameters later in this chapter).

Here are is the definition of a procedure using the Pascal language syntax, which uses the specific `procedure` keyword and is part of the `FunctionTest` project:

```
procedure Hello;
begin
  Show ('Hello world!');
end;
```

As a comparison, this would be the same function written with the C language syntax, which has no keyword, requires the parenthesis even in case there are no parameters, and has a `void` or empty return value to indicate no result:

```
void Hello ()
{
  Show ("Hello world!");
};
```

In fact, in the C language syntax there is no difference between procedure and function. In the Pascal language syntax, instead, a function has a specific keyword and must have a return value (or return type).

---

note There is another very specific syntactical difference between Object Pascal and other languages, that is presence of a semicolon at the end of the function or procedure signature in the definition, before the `begin` keyword.

There are two ways to indicate the result of the function call, assign the value to function name or use the `Result` keyword:

```
// classic style
function DoubleOld (Value: Integer) : Integer;
begin
  DoubleOld := Value * 2;
end;

// modern alternative
function Double (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

note Differently from the classic Pascal language syntax, Object Pascal has actually three ways to indicate the result of a function, including the Exit mechanism discussed in this chapter in the section "Exit with a Result".

The use of `Result` instead of the function name to assign the return value of a function is the most common syntax and tends to make the code more readable. The use of the function name is a classic Pascal notation, now rarely used.

Again, by comparison the same function could be written with the C language syntax as the following:

```
int Double (int Value)
{
  return Value * 2;
};
```

If this is how these routines can be defined, the calling syntax is relatively straightforward, as you type in the identifier followed by the parameters within parenthesis. In case there are no parameters, the empty parenthesis can be omitted (again, unlike languages based on the C syntax). This code snippet and several following ones are part of the `FunctionsTest` project of this chapter:

```
// call the procedure
Hello;

// call the function
X := Double (100);
Y := Double (X);
Show (Y.ToString);
```

This is the encapsulation of code concept that I've introduced. When you call the `Double` function, you don't need to know the algorithm used to implement it. If you later find out a better way to double numbers, you can easily change the code of the

function, but the calling code will remain unchanged (although executing it might become faster).

The same principle can be applied to the `Hello` procedure: We can modify the program output by changing the code of this procedure, and the main program code will automatically change its effect. Here is how we can change the procedure implementation code:

```
procedure Hello;
begin
  Show ('Hello world, again!');
end;
```

Forward Declarations

When you need to use an identifier (of any kind), the compiler must have already seen it, to know to what the identifier refers. For this reason, you usually provide a full definition before using any routine. However, there are cases in which this is not possible. If procedure A calls procedure B, and procedure B calls procedure A, when you start writing the code, you will need to call a routine for which the compiler still hasn't seen a definition.

In this cases (and in many others) you can declare the existence of a procedure or function with a certain name and given parameters, without providing its actual code. One way to declare a procedure or functions without defining it is to write its name and parameters (referred to as the function signature) followed by the `forward` keyword:

```
procedure NewHello; forward;
```

Later on, the code should provide a full definition of the procedure (which must be in the same unit), but the procedure can now be called before it is fully defined. Here is an example, just to give you the idea:

```
procedure DoubleHello; forward;

procedure NewHello;
begin
  if MessageDlg ('Do you want a double message?',
      TMsgDlgType.mtConfirmation,
      [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo],
      0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
end;

procedure DoubleHello;
```

```
begin
  NewHello;
  NewHello;
end;
```

note The `MessageDlg` function called in the previous snippet is a relatively simple way to ask a confirmation of the user in the FM framework (a similar functions exists in the VCL framework as well). The parameters are the message, the type of dialog box, and buttons you want to display. The result is the identifier of the button that was selected.

This approach (which is also part of the FunctionTest demo project) allows you to write mutual recursion: `DoubleHello` calls `Hello`, but `Hello` might call `DoubleHello` too. In other words, if you keep selecting the Yes button the program will continue showing the message, and show each twice for every *Yes*. In recursive code, there must be a condition to terminate the recursion, to avoid a condition known as stack overflow.

note Function calls uses a stack for the parameters, the return value, local variables and more. If a functions keeps calling itself in an endless loop, the memory area for the stack (which is generally of a fixed and predefined size, determined by the linker) will terminate through an error known as a *stack overflow*. Needless to say that the popular developers support site (`www.stackoverflow.com`) took its name from this programming error.

Although a forward procedure declaration is not very common in Object Pascal, there is a similar case that is much more frequent. When you declare a procedure or function in the interface section of a unit, it is automatically considered as a forward declaration, even if the `forward` keyword is not present. Actually you cannot write the body of a routine in the interface section of a unit. At the same time, you must provide in the same unit the actual implementation of each routine you have declared.

A Recursive Function

Given I mentioned recursion and gave a rather peculiar example of it (with two procedures calling each other), let me also show you a classic example of a recursive function calling itself. Using recursion is often an alternative way to code a loop.

To stick with a classic demo, suppose you want to compute the power of a number, and you lack the proper function (which is available in the run-time library, of course). You might remember from math, that 2 at the power of 3 corresponds to multiplying 2 by itself 3 times, that is 2*2*2.

One way to express this in code would be to write a `for` loop that is executed 3 times (or the value of the exponent) and multiplies 2 (or the value of the base) by the current total, starting with 1:

```
function PowerL (Base, Exp: Integer): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 1 to Exp do
    Result := Result * Base;
end;
```

An alternative approach is to repeatedly multiply the base by the power of the same number, with a decreasing exponent, until the exponent is 0, in which case the result is invariably 1. This can be expressed by calling the same function over and over, in a recursive way:

```
function PowerR (Base, Exp: Integer): Integer;
var
  I: Integer;
begin
  if Exp = 0 then
    Result := 1
  else
    Result := Base * PowerR (Base, Exp - 1);
end;
```

The recursive version of the program is likely not faster than the version based on the `for` loop, nor more readable. However there are scenarios such as parsing code structures (a tree structure for example) in which there isn't a fixed number of elements to process, and hence writing a loop is close to impossible, while a recursive functions adapts itself to the role.

In general, though, recursive code is powerful but tends to be more complex. After many years in which recursion was almost forgotten, compared to the early days of programming, new functional languages such Haskell, Erlang and Elixir that make heavy use of recursion and are driving this idea back to popularity. In any case, you can find the two power functions in the code in the FunctionTest demo.

---

note The two power functions of the demo don't handle the use of a negative exponent. The recursive version in such a case will loop forever (an until the program hits a physical constraint). Also, by using integers it is relatively fast to reach the maximum data type size and overflow it. I wrote these functions with these inherent limitations to try to keep their code simple.

What Is a Method?

We have seen how you can write a forward declaration in the interface section of a unit of by using the `forward` keyword. The declaration of a method inside a class type is also considered a forward declaration.

But what exactly is a method? A method is a special kind of function or procedure that is related to a data type, a record or a class. In Object Pascal, every time we handle an event for a visual component, we need to define a method, generally a procedure, but the term method is used to indicate both functions and procedures tied to a class or record.

Here is an empty method automatically added to the source code of a form (which is indeed a class, as we'll explore much later in the book):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {here goes your code}
end;
```

Parameters and Return Values

When you call a function or procedure you need to pass the correct number of parameters and make sure they match the expected type. If not, the compiler will issue an error message, similar to a type mismatch when you assign to a variable a value of a wrong type. Given the previous definition of the `Double` function, taking an Integer parameter, if you call:

```
Double (10.0);
```

The compiler will show the error:

```
[dcc32 Error] E2010 Incompatible types: 'Integer' and 'Extended'
```

tip The editor helps you by suggesting the parameters list of a function or procedure with a fly-by hint as soon as you type its name and the open parenthesis. This feature is called Code Parameters and is part of the Code Insight technology (known in other IDEs as *IntelliSense*).

There are scenarios in which limited type conversion is allowed, similarly to assignments, but in general you should try to use parameters of the specific type (this is compulsory for reference parameters, as we'll see in a while).

When you call a function, you can pass an expression as a parameter instead of a value. The expression is evaluated and its result assigned to the parameter. In sim-

pler cases, you just pass the name of a variable. In this case, the value of the variable is copied to the parameter (which generally has a different name). I strongly discourage you to use the same name for a parameter and for a variable passed as the value of that parameter, because this can be quite confusing.

Finally, notice that you can have a function or procedure with different versions (a feature called *overloading*) and with parameters you can skip to let them use a predefined value (a feature called *default parameters*). These two key features for functions and procedures are detailed in specific sections later in this chapter.

Exit with a Result

We have seen that returning a result from a function uses quite a different syntax compared to the C language (or other languages deriving from it). Not only the syntax is different, but also the behavior. Assigning a value to `Result` (or to the function name) doesn't terminate the function as a `return` statement does.

Object Pascal developers often take advantage of this feature, by using Result as a temporary storage. Rather than writing:

```
function ComputeValue: Integer;
var
  value: Integer;
begin
  value := 0;
  while ...
    Inc (value);
  Result := value;
end;
```

You can omit the temporary variable and directly use `Result` instead. Whatever value `Result` has when the function terminates, would be the value returned by the function:

```
function ComputeValue: Integer;
begin
  Result := 0;
  while ...
    Inc (Result);
end;
```

On the other hand there are situations in which you want to assign a value and exit from the procedure right away, for example in a specific `if` branch. If you need to assign the function result *and* stop the current execution you have to use two separate statements, assign the `Result` and then use the `Exit` keyword.

If you remember the code of the `FlowTest` demo of the last chapter (covered in the section “Breaking the Flow with Break and Continue”), this could be rewritten as a

function, replacing the call to `Break` with a call to `Exit`. I've made this change in the following code snippet, part of the `ParamsTest` demo:

```
function CharInString (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
    begin
      Result := True;
      Exit;
    end;
end;
```

In Object Pascal you can replace the two statements of the `if` block with a special call to `Exit` passing to it the return value of the function, in a way resembling the C language `return` statement. So you can write the code above in a more compact way (also because with a single statement you can avoid the `begin-end` block):

```
function CharInString2 (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
      Exit (True);
end;
```

note Exit in Object Pascal is a function so you must enclose the value to be returned in parentheses whereas return in C-style languages is a compiler keyword not requiring parentheses.

Reference Parameters

In Object Pascal, procedures and functions allow parameter passing by value and by reference. Passing parameters by value is the default: the value is copied on the stack and the routine uses and manipulates this copy of the data, not the original value (as I described earlier in the section “Function Parameters and Return Values”).

Passing a parameter by reference means that its value is not copied onto the stack in the formal parameter of the routine. Instead, the program refers to the original value, also in the code of the routine. This allows the procedure or function to change the actual value of the variable that was passed as parameter. Parameter passing by reference is expressed by the *var* keyword.

This technique is available in most programming languages, also because avoiding a copy often means that the program executes faster. It isn't present in C (where you can just use a pointer), but it was introduced in C++ and other languages based on the C syntax, where you use the `&` (pass by reference) symbol. Here is an example of passing a parameter by reference using the `var` keyword:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

In this case, the parameter is used both to pass a value to the procedure and to return a new value to the calling code. When you write:

```
var
  X: Integer;
begin
  X := 10;
  DoubleTheValue (X);
  Show (X.ToString);
```

the value of the X variable becomes 20, because the function uses a reference to the original memory location of X, affecting its original value.

Compared to general parameters passing rules, passing values to reference parameters is subject to more restrictive rules, given what you are passing is not a value, but an actual variable. You cannot pass a constant value as a reference parameter, an expression, the result of a function, or a property. Another rule is you cannot pass a variable of a slightly different type (requiring automatic conversion). The type of the variable and the parameter must match exactly, or as the compiler error message says:

```
[dcc32 Error] E2033 Types of actual and formal var parameters must be
identical
```

This is the error message you'll get if you write, for example (this is also part of the `ParamsTest` demo, but commented out):

```
var
  C: Cardinal;
begin
  C := 10;
  DoubleTheValue (C);
```

Passing parameters by reference makes sense for ordinal types and for records (as we'll see in the next chapter). These types are often called *value types* because they have by default a pass-by-value and assign-by-value semantic.

Object Pascal objects and strings have a slightly different behavior we'll investigate in more detail later on. Objects variables are references, so you can modify the actual

data of an object passed as parameter. These types are part of the different group, often indicated as *reference* types.

Beside standard and reference (`var`) parameter types, Object Pascal has also a very unusual kind of parameter specifier, `out`. An `out` parameter has no initial value and it's used only to return a value. Except for not having an initial value, `out` parameters behave like `var` parameters.

note The `out` parameters were introduced for supporting the corresponding concept in Windows' Component Object model (or COM). They are rarely used outside of this context; in general, it is better to stick with the more efficient (and easier to understand) `var` parameters.

Constant Parameters

As an alternative to reference parameters, you can use a `const` parameter. Since you cannot assign a new value to a constant parameter inside the routine, the compiler can optimize parameter passing. The compiler can choose an approach similar to reference parameters (or a const reference in C++ terms), but the behavior will remain similar to value parameters, because the original value cannot be modified by the function.

In fact, if you try to compile the following code (available, but commented out in the `ParamsTest` project), the system will issue an error:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;      // compiler error
  Result := Value;
end;
```

The error message you'll see might not be immediately intuitive, as it says:

```
[dcc32 Error] E2064 Left side cannot be assigned to
```

Constant parameters are quite common for strings, because in this case the compiler can disable the reference counting mechanism obtaining a slight optimization. The same is true for passing constant objects in versions of Object Pascal that use ARC (Automatic Reference Counting). More about these topics later on in the book: It is worth mentioning them here anyway because these optimizations are the most common reason for using constant parameters, a features that make limited sense for ordinal and scalar types.

Function Overloading

At times you might want to have two very similar functions with different parameters and a different implementation. While traditionally you'd have to come up with a slight different name for each, modern programming languages let you *overload* a symbol with multiple definitions.

The idea of overloading is simple: The compiler allows you to define two or more functions or procedures using the same name, provided that the parameters are different. By checking the parameters, in fact, the compiler can determine which of the version of the function you want to call. Consider this series of functions extracted from the `System.Math` unit of the run-time library:

```
function Min (A,B: Integer): Integer; overload;
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

When you call `Min (10, 20)`, the compiler determines that you're calling the first function of the group, so the return value will also be an Integer.

There are two basic rules of overloading:

- Each version of an overloaded function (or procedure) must be followed by the `overload` keyword (including the first one).
- Among overloaded functions, there must be a difference in the number or in the type of the parameters. Parameter names are not considered, because they are not indicated during the call. Also, the return type cannot be used to distinguish among two overloaded functions.

---

note There is an exception to the rule you cannot distinguish functions on the return values and it is for the `Implicit` and `Explicit` conversion operators, covered in Chapter 5.

Here are three overloaded versions of a `ShowMsg` procedure I've added to the `OverloadTest` example (an application demonstrating both overloading and default parameters):

```
procedure ShowMsg (str: string); overload;
begin
  Show ('Message: ' + str);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  Show ('Message: ' + Format (FormatStr, Params));
```

```
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  ShowMsg (I.ToString + ' ' + Str);
end;
```

The three functions show a message box with a string, after optionally formatting the string in different ways. Here are the three calls of the program:

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

And this is their effect:

```
Message: Hello
Message: Total = 100.
Message: 10 MBytes
```

tip The Code Parameters technology of the IDE works very nicely with overloaded procedures and functions. As you type the open parenthesis after the routine name, all the available alternatives are listed. As you enter the parameters, the Code Insight technology uses their type to determine which of the alternatives are still available.

What if you try to call the function with parameters that don't match any of the available overloaded versions? You'll get an error message, of course. Suppose you want to call:

```
ShowMsg (10.0, 'Hello');
```

The error you'll see in this case is a very specific one:

```
[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that
can be called with these arguments
```

The fact that each version of an overloaded routine must be properly marked implies that you cannot overload an existing routine of the same unit that is not marked with the `overload` keyword. The error message you get when you try is:

```
Previous declaration of '<name>' was not marked with the 'overload'
directive.
```

You can, however, create a routine with the same name of one that was declared in a *different unit*, given that units act as namespaces. In this case, you are not overloading a function with a new version, but you are replacing the function with a new version, hiding the original one (which can be referenced using the unit name prefix). This is why the compiler won't be able to pick a version based on the parameters, but it will try to match the only version is sees, issuing an error if the parameters types don't match.

Overloading and Ambiguous Calls

When you call an overloaded function, the compiler will generally find a match and work correctly or issue an error if none of the overloaded versions has the right parameters (as we have just seen). But there is also a third scenario: Given the compiler can do some type conversions for the parameters of a function, there might be different possible conversions for a single call. When the compiler finds multiple versions of a function it can call, and there isn't one that is a perfect type match (which would be picked) it issues an error message indicating that the function call is *ambiguous*.

This is not a common scenario, and I had to build a rather illogical example to show it to you, but it is worth considering the case (as it does happen occasionally in real world). Suppose you decide to implement two overloaded functions to add integers and floating point numbers:

```
function Add (N: Integer; S: Single): Single; overload;
begin
  Result := N + S;
end;

function Add (S: Single; N: Integer): Single; overload;
begin
  Result := N + S;
end;
```

These functions are in the `OverloadTest` example. Now you can call them passing the two parameters in any order:

```
Show (Add (10, 10.0).ToString);
Show (Add (10.0, 10).ToString);
```

However the fact is that, in general, a function can accept a parameter of a different type when there is a conversion, like accepting an integer when the function expects a parameter of a floating point type. So what happens if you call:

```
Show (Add (10, 10).ToString);
```

The compiler can call the first version of the overloaded function, but it can also call the second version. Not knowing what you are asking for (and know knowing if calling one function or the other will produce the same effect), it will issue an error:

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
  Related method: function Add(Integer; Single): Single;
  Related method: function Add(Single; Integer): Single;
```

tip In the errors pane of the IDE you'll see an error message with the first line above, and a plus sign on the side you can expand to see the following two lines with the details of which overloaded functions the compiler is considering ambiguous.

If this is a real world scenario, and you need to make the call, you can add a manual type conversions call to solve the problem and indicate to the compiler which of the versions of the function you want to call:

```
Show (Add (10, 10.ToSingle).ToString);
```

A particular case of ambiguous calls can happen if you use variants, a rather peculiar data type I'll cover only later in the book.

Default Parameters

Another feature related to overloading, is the possibility of giving a default value to some of the parameters of a function, so that you can call the function with or without those parameters. If the parameter is missing in the call, it will take the default value.

Let me show an example (still part of the `OverloadTest` example). We can define the following encapsulation of the `Show` call, providing two default parameters:

```
procedure NewMessage (Msg: string;
  Caption: string = 'Message';
  Separator: string = ': ');
begin
  Show (Caption + Separator + Msg);
end;
```

With this definition, we can call the procedure in each of the following ways:

```
NewMessage ('Something wrong here!');
NewMessage ('Something wrong here!', 'Attention');
NewMessage ('Hello', 'Message', '--');
```

This is the output:

```
Message: Something wrong here!
Attention: Something wrong here!
Message--Hello
```

Notice that the compiler doesn't generate any special code to support default parameters; nor does it create multiple (overloaded) copies of the functions or procedure. The missing parameters are simply added by the compiler to the calling code. There is one important restriction affecting the use of default parameters: You cannot "skip" parameters. For example, you can't pass the third parameter to the function after omitting the second one.

There are a few other rules for the definition and the calls of functions and procedures (and methods) with default parameters:

- In a call, you can only omit parameters starting from the last one. In other words, if you omit a parameter you must also omit the following ones.

- In a definition, parameters with default values must be at the end of the parameters list.
- Default values must be constants. Obviously, this limits the types you can use with default parameters. For example, a dynamic array or an interface type cannot have a default parameter other than `nil`; records cannot be used at all.
- Parameters with defaults must be passed by value or as `const`. A reference (`var`) parameter cannot have a default value.

Using default parameters and overloading at the same time makes it more likely to get you in a situation which confuses the compiler, raising an *ambiguous call* error, as mentioned in the previous section. For example, if I add the following new version of the `NewMessage` procedure to the previous example:

```
procedure NewMessage (Str: string; I: Integer = 0); overload;
begin
  Show (Str + ': ' + IntToStr (I))
end;
```

then the compiler won't complain, as this is a legitimate definition. However, the call:

```
NewMessage ('Hello');
```

is flagged by the compiler as:

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'
  Related method: procedure NewMessage(string; string; string);
  Related method: procedure NewMessage(string; Integer);
```

Notice that this error shows up in a line of code that compiled correctly before the new overloaded definition. In practice, we have no way to call the `NewMessage` procedure with one string parameter, as the compiler doesn't know whether we want to call the version with only the string parameter or the one with the string parameter and the integer parameter with a default value. When it has a similar doubt, the compiler stops and asks the programmer to state his or her intentions more clearly.

Inlining

Inlining Object Pascal functions and methods is a low-level language feature that can lead to significant optimizations. Generally, when you call a method, the compiler generates some code to let your program jump to a new execution point. This implies setting up a stack frame and doing a few more operations and might require a dozen or so machine instructions. However, the method you execute might be very short, possibly even an access method that simply sets or returns some private field.

In such a case, it makes a lot of sense to copy the actual code to the call location, avoiding the stack frame setup and everything else. By removing this overhead, your program will run faster, particularly when the call takes place in a tight loop executed thousands of times.

For some very small functions, the resulting code might even be smaller, as the code pasted in place might be smaller than the code required for the function call. However, notice that if a longer function is inlined and this function is called in many different places in your program, you might experience code bloat, which is an unnecessary increase in the size of the executable file.

In Object Pascal you can ask the compiler to inline a function (or a method) with the `inline` directive, placed after the function (or method) declaration. It is not necessary to repeat this directive in the definition. Always keep in mind that the `inline` directive is only a hint to the compiler, which can decide that the function is not a good candidate for inlining and skip your request (without warning you in any way). The compiler might also inline some, but not necessarily all, of the calls of the function after analyzing the calling code and depending on the status of the `$INLINE` directive at the calling location. This directive can assume three different values (notice that this feature is independent from the optimization compiler switch):

- With default value, `{$INLINE ON}`, inlining is enabled for functions marked by the `inline` directive.
- With `{$INLINE OFF}` you can suppress inlining in a program, in a portion of a program, or for a specific call site, regardless of the presence of the `inline` directive in the functions being called.
- With `{$INLINE AUTO}` the compiler will generally inline the functions you mark with the directive, plus automatically inline very short functions. Watch out because this directive can cause code bloat.

There are many functions in the Object Pascal Run-Time Library that have been marked as inline candidates. For example, the `Max` function of the `Math` unit has definitions like:

```
function Max(const A, B: Integer): Integer;
  overload; inline;
```

To test the actual effect of inlining this function, I've written the following loop in the `InliningTest` example:

```
var
  sw: TStopWatch;
  I, J: Integer;
begin
  J := 0;
  sw := TStopWatch.StartNew;
  for I := 0 to LoopCount do
    J := Max (I, J);
```

```
  sw.Stop;
  Show ('Max ' + J.ToString +
      ' [' + sw.ElapsedMilliseconds.ToString + '] ');
```

In this code, the `TStopWatch` record of the `System.Diagnostics` unit, a structure that keep track of the time (or system ticks) elapsed between the `Start` (or `StartNew`) and the `Stop` calls.

The form has two buttons both calling this same exact code, but one of them has inlining disabled at the call site. Notice you need to compile with the Release configuration to see any difference (as inlining is a Release optimization). With twenty million interactions (the value of the `LoopCount` constant), on my computer I get the following results:

```
// on Windows (running in a VM)
Max on 20000000 [17]
Max off 20000000 [45]

// on Android (on device)
Max on 20000000 [280]
Max off 20000000 [376]
```

How can we read this data? On Windows, inlining more than doubles the execution speed, while on Android it makes the program about 35% faster. However, on a device the program runs much slower (an order of magnitude) so while on Windows we shave off 30 milliseconds on my Android device this optimization saves about 100 milliseconds.

The same program does a second similar test with the `Length` function, a compiler-magic function that was specifically modified to be inlined. Again the inlined version is significantly faster on both Windows and Android:

```
// on Windows (running in a VM)
Length inlined 260000013 [11]
Length not inlined 260000013 [40]

// on Android (on device)
Length inlined 260000013 [401]
Length not inlined 260000013 [474]
```

This is the code used by this second testing loop:

```
var
  sw: TStopWatch;
  I, J: Integer;
  sample: string;
begin
  J := 0;
  sample:= 'sample string';
  sw := TStopWatch.StartNew;
  for I := 0 to LoopCount do
    Inc (J, Length(sample));
  sw.Stop;
```

```
  Show ('Length not inlined ' + IntToStr (J) +
      ' [' + IntToStr (sw.ElapsedMilliseconds) + ' ] ');
end;
```

The Object Pascal compiler doesn't define a clear cut limit on the size of a function that can be inlined or a specific list of constructs (`for` or `while` loops, conditional statements) that would prevent inlining. However, since inlining a large function provides little advantage yet exposes you to the risk of some real disadvantages (in terms of code bloat), you should avoid it.

One limitation is that the method or function cannot reference identifiers (such as types, global variables, or functions) defined in the implementation section of the unit, as they won't be accessible in the call location. However, if you are calling a local function, which happens to be inlined as well, then the compiler will accept your request to inline your routine.

A drawback is that inlining requires more frequent recompilations of units, as when you modify an inlined function, the code of each of the calling sites will need to be recompiled as well. Within a unit, you might write the code of the inlined functions before calling them, but better place them at the beginning of the implementation section.

---

note Object Pascal uses a single pass compiler, so it cannot paste in the code of a function it hasn't compiled yet.

Within different units, you need to specifically add other units with inlined functions to your `uses` statements, even if you don't call those methods directly. Suppose your unit A calls an inlined function defined in unit B. If this function in turn calls another inlined function in unit C, your unit A needs to refer to C as well. If not, you'll see a compiler warning indicating the call was not inlined due to the missing unit reference. A related effect is that functions are never inlined when there are circular unit references (through their implementation sections).

Advanced Features of Functions

If what I have covered so far includes the core features related to functions, there are several advanced capabilities worth exploring. If you are really a newbie in terms of software development, however, you might want to skip the rest of this chapter for now and move to the next one.

Object Pascal Calling Conventions

Whenever your code calls a function, the two sides need to agree on the actual practical way parameters are passed from the caller to the callee, something called *calling convention*. Generally, a function call takes place by passing the parameters (and expecting the return value) via the stack memory area. However, the order in which the parameters and return value are placed on the stack changes depending on the programming language and platform, with most languages capable of using multiple different calling conventions.

A long time ago, the 32-bit version of Delphi introduced a new approach to passing parameters, known as "*fastcall*": Whenever possible, up to three parameters can be passed in CPU registers, making the function call much faster. Object Pascal uses this fast calling convention by default although it can also be requested by using the `register` keyword.

The problem is that this is the default convention, and functions using it are not compatible with external libraries, like Windows API functions in Win32. The functions of the Win32 API must be declared using the `stdcall` (*standard call*) calling convention, a mixture of the original `pascal` calling convention of the Win16 API and the `cdecl` calling convention of the C language. All of these calling conventions are supported in Object Pascal, but you'll rarely use something different than the default unless you need to invoke a library written in a different language, like a system library.

The typical case you need to move away from the default fast calling convention is when you need to call the native API of a platform, which requires a different calling convention depending on the operating system. Even Win64 uses a different model to Win32, so Object Pascal supports many different options, not really worth detailing here. Mobile operating systems, instead, tend to expose classes, rather than native functions, although the issue of respecting a given calling convention has to be taken into account even in that scenario.

Procedural Types

Another unique feature of Object Pascal is the presence of procedural types. These are really an advanced language topic, which only a few programmers will use regularly. However, since we will discuss related topics in later chapters (specifically, method pointers, a technique heavily used by the environment to define event handlers, and anonymous methods), it's worth giving a quick look at them here.

In Object Pascal (but not in the more traditional Pascal language) there is the concept of a procedural type (which is similar to the C language concept of a function pointer – something languages like C# and Java have dropped, because it is tied to global functions).

The declaration of a procedural type indicates the list of parameters and, in the case of a function, the return type. For example, you can declare a new procedural type, with an Integer parameter passed by reference, with this code:

```
type
  TIntProc = procedure (var Num: Integer);
```

This procedural type is compatible with any routine having exactly the same parameters (or the same function signature to use C jargon). Here is an example of a compatible routine:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Procedural types can be used for two different purposes: you can declare variables of a procedural type or pass a procedural type (that is, a function pointer) as parameter to another routine. Given the preceding type and procedure declarations, you can write this code:

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

This code has the same effect as the following shorter version:

```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

The first version is clearly more complex, so why and when should we use it? There are cases in which being able to decide which function to call and actually calling it later on can be very powerful. It is possible to build a complex example showing this approach. However, I prefer to let you explore a fairly simple one, called ProcType.

This example is based on two procedures. One procedure is used to double the value of the parameter like the one I've already shown. A second procedure is used to triple the value of the parameter, and therefore is named `TripleTheValue`:

```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
end;
```

Instead of calling these functions directly, one or the other are saved in a procedural type variable. The variable is modified as a users selects a checkbox, and the current procedure is called in this generic way as a user clicks the button. The program uses two initialized global variables (the procedure to be called and the current value), so that these values are preserved over time. This is the full code, save for the definitions of the actual procedures, already shown above:

```
var
  IntProc: TIntProc = DoubleTheValue;
  Value: Integer = 1;

procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  if CheckBox1.IsChecked then
    IntProc := TripleTheValue
  else
    IntProc := DoubleTheValue;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  IntProc (Value);
  Show (Value.ToString);
end;
```

When the user changes the check box status, all following button clicks will call the active function. So if you press the button twice, change the selection, and press the button twice again, you'll first double twice and then triple twice the current value, producing the following output:

```
2
4
12
36
```

Another practical example of the use of procedural types is when you need to pass a function to an operating system like Windows (where they are generally called “callback functions”). As mentioned at the beginning of this section, in addition to procedural types Object Pascal developers use method pointers (covered in Chapter 10) and anonymous methods (covered in Chapter 15).

---

note The most common object oriented mechanism to obtain a late bound function call (that is a function call that can change at runtime) is the use of virtual methods. While virtual methods are very common in Object Pascal, procedural types are seldom used. The technical foundation, though, is somehow similar. Virtual functions and polymorphism are covered in Chapter 8.

External Functions Declarations

Another important element for system programming is represented by external declarations. Originally used to link code to external functions that were written in assembly language, external declarations became commonplace Windows programming to call a function from a DLL (a dynamic link library). An external function declaration implies the ability to call a function not fully available to the compiler or the linker, but requiring the ability to load an external dynamic library and invoke one of its functions.

note Whenever you call an API for a given platform in your Object Pascal code you lose the ability to recompile the application for any other platform than the specific one. The exception is if the call is surrounded by platform specific `$IFDEF` compiler directives.

This is, for example, how you can invoke Windows API functions from an Object Pascal application. If you open the `Winapi.Windows` unit you'll find many function declarations and definitions like:

```
// forward declaration
function GetUserName(lpBuffer: LPWSTR;
  var nSize: DWORD): BOOL; stdcall;

// external declaration (instead of actual code)
function GetUserName; external advapi32
  name 'GetUserNameW';
```

You seldom need to write declarations like the one just illustrated, since they are already listed in the `Windows` unit and many other system units. The only reason you might need to write this external declaration code is to call functions from a custom DLL, or to call Windows functions not translated in the platform API.

This declaration means that the code of the function `GetUserName` is stored in the advapi32 dynamic library (advapi32 is a constant associated with the full name of the DLL, *'advapi32.dll'*) with the name `GetUserNameW`, as this API function has both an ASCII and a WideString version. Inside an external declaration, in fact, we can specify that our function refers to a function of a DLL that originally had a different name.

Delayed Loading of DLL Functions

In the Windows operating system, there are two ways to invoke an API function of the Windows SDK (or any other DLL): you can let the application loader resolve all

references to external functions or you can write specific code that looks for a function and executes it if available.

The former code is easier to write (as we saw in the previous section): as all you need is the external function declaration. However if the library or even just one of the functions you want to call is not available on all versions of Windows, your program will not be able to start on the operating system versions that don't provide that function.

Dynamic loading allows for more flexibility, but implies loading the library manually, using the `GetProcAddress` API for finding the function you want to call, and invoking it after casting the pointer to the proper type. This kind of code is quite cumbersome and error prone.

That's why it is good that the Object Pascal compiler and linker have specific support for a feature now available at the Windows operating system level and already used by some C++ compilers, the delayed loading of functions until the time they are called. The aim of this declaration is not to avoid the implicit loading of the DLL, which takes place anyway, but to allow the delayed binding of that specific function within the DLL.

You basically write the code in a way that's very similar to the classic execution of DLL function, but the function address is resolved the first time the function is called and not at load time. This means that if the function is not available you get a run-time exception, `EExternalException`. However, you can generally verify the current version of the operating system or the version of the specific library you are calling, and decide in advance whether you want to make the call or not.

---

note If you want something more specific and easier to handle at a global level than an exception, you can hook into the error mechanism for the delayed loading call, as explained by Allen Bauer in his blog post: http://blogs.embarcadero.com/abauer/2009/08/29/38896

From the Object Pascal language perspective, the only difference is in the declaration of the external function. Rather than writing:

```
function MessageBox;
  external user32 name 'MessageBoxW';
```

You can now write (again, from an actual example in the `Windows` unit):

```
function WindowFromPhysicalPoint;
  external user32
  name 'WindowFromPhysicalPoint' delayed;
```

At run time, considering that the API was added to Vista (that is, Windows 6.0) for the first time, you might want to write code like the following:

```
  if CheckWin32Version (6, 0) then
```

```
begin
  hwnd := WindowFromPhysicalPoint (aPoint);
```

This is much, much less code than you had to write without delayed loading to be able to run the same program on older versions of Windows.

Another relevant observation is that you can use the same mechanism when building your own DLLs and calling them in Object Pascal, providing a single executable that can bind to multiple versions of the same DLL as long as you use delayed loading for the new functions.

Again, this is mostly related to Windows programming, and doesn't really apply to other operating systems that expose classes and higher level abstractions rather than plain C language functions as the core of the Windows API still does today. Given features like external declarations and delayed loading are technically part of the compiler and the language, though, I though it was worth mentioning them in this chapter.

05: arrays and records

When I introduced data types in Chapter 2, I referred to the fact that in Object Pascal there are both built in data types and type constructors. A simple example of a type constructor is the enumerated type, covered in that chapter.

The real power of type definition comes with more advanced mechanisms, such as arrays, records, and classes. In this chapter I'll cover the first two, which in their essence date back to the early definition of Pascal, but have been changed so much over the years (and made so powerful) that they barely resemble their ancestral type constructors with the same name.

Towards the end of the chapter I'll also briefly introduce some advanced Object Pascal data types as pointers. The real power of custom data types, however, will be unveiled in Chapter 7, where we'll start looking into classes and object-oriented programming.

Array Data Types

Array types define lists with elements of a specific type. These lists can have a fixed number of elements (static arrays) or of a variable number of elements (dynamic

arrays). You generally use an *index* within square brackets to access one of the elements of an array. Square brackets are also used to specify the number of values of a fixed size array.

The Object Pascal language supports different array types, from traditional static arrays to dynamic ones. Use of dynamic arrays is recommended, particularly with the mobile versions of the compiler. I'll introduce static arrays first, and later focus on dynamic ones.

Static Arrays

Traditional Pascal language arrays are defined with a static or fixed size. An example is in the following code snippets, which defines a list of 24 integers, presenting the temperatures during the 24 hours of a day:

```
type
  TDayTemperatures = array [1..24] of Integer;
```

In this classic array definition, you can use a subrange type within square brackets, actually defining a new specific subrange type using two constants of an ordinal type. This subrange indicates the valid indexes of the array. Since you specify both the upper and the lower index of the array, the indexes don't need to be zero-based, as it is the case in C, C++, Java, and most other languages (although 0-based arrays are also quite common in Object Pascal). Notice also that static array indexes in Object Pascal can be numbers, but also other ordinal types like characters, enumerated types, and more. Non-integral indexes are quite rare, though.

note There are languages like JavaScript that make heavy use of associative arrays. Object Pascal arrays are limited to ordinal indexes, so you cannot directly use a string as index. There are ready to use data structures in the RTL implementing Dictionaries and other similar data structures. I'll cover them in the chapter about Generics, in the third part of the book.

Since the array indexes are based on subranges, the compiler can check their range. An invalid constant subrange results in a compile-time error; and an out-of-range index used at run-time results in a run-time error, but only if the corresponding compiler option is enabled.

note This is the *Range checking* option of the *Runtime errors* group of the *Compiling* page of the Project Options dialog of the IDE. I've already mentioned this option in Chapter 2, in the section "Subrange Types".

Using the array definition above, you can set the value of a `DayTemp1` variable of the `TDayTemperatures` type as follows (and as I've done in the ArraysTest program, from which the following code snippets have been extracted):

```
type
  TDayTemperatures = array [1..24] of Integer;

var
  DayTemp1: TDayTemperatures;

begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;

  // The following line causes:
  // E1012 Constant expression violates subrange bounds
  // DayTemp1 [25] := 67;
```

Now a standard way to operate on arrays, given their nature, is to use for cycles. This is an example of a loop used to display all of the temperatures for a day:

```
var
  I: Integer;
begin
  for I := 1 to 24 do
    Show (I.ToString + ': ' + DayTemp1[I].ToString);
```

While this code works, having hard-coded the array boundaries (`1` and `24`) is far from ideal, as the array definition itself might change over time and you might want to move to using a dynamic array.

Array Size and Boundaries

When you work with an array, you can always test its boundaries by using the standard `Low` and `High` functions, which return the lower and upper bounds. Using `Low` and `High` when operating on an array is highly recommended, especially in loops, since it makes the code independent of the current range of the array (which might go from 0 to the length of the array minus one, might start form 1 and reach the array's length, or have any other subrange definition). If you should later change the declared range of the array indexes, code that uses `Low` and `High` will still work. If you write a loop hard-coding the range of an array you'll have to update the code of the loop when the array size changes. `Low` and `High` make your code easier to maintain and more reliable.

note Incidentally, there is no run-time overhead for using `Low` and `High` with static arrays. They are resolved at compile-time into constant expressions, not actual function calls. This compile-time resolution of expressions and function calls also happens for many other system functions.

Another relevant function is `Length`, which returns the number of elements of the array. I've combined these three functions in the following code that computes and displays the average temperature for the day:

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(DayTemp1) to High(DayTemp1) do
    Inc (Total, DayTemp1[I]);
  Show ((Total / Length(DayTemp1)).ToString);
```

This code is also part of the ArraysTest example.

Multi-Dimensional Static Arrays

An array can have more than one dimension, expressing a matrix or a cube rather than a list. Here are two sample definitions:

```
type
  TAllMonthTemps = array [1..24, 1..31] of Integer;
  TAllYearTemps = array [1..24, 1..31, 1..12] of Integer;
```

You can access an element as:

```
var
  AllMonth1: TAllMonthTemps;
  AllYear1: TAllYearTemps;
begin
  AllMonth1 [13, 30] := 55; // hour, day
  AllYear1 [13, 30, 8] := 55; // hour, day, month
```

note Static arrays immediately take up a lot of memory (in the case above on the stack), which should be avoided. The `AllYear1` variable requires 8,928 Integers, taking up 4 bytes each, that is almost 35KB. Allocating such a large block in the global memory or on the stack (as in the demo code) is really a mistake. A dynamic array, instead, uses the heap memory, and offers much more flexibility in terms of memory allocation and management.

Given these two array types are built on the same core types, you should better declare them using the preceding data types, as in the following code:

```
type
  TMonthTemps = array [1..31] of TDayTemperatures;
  TYearTemps = array [1..12] of TMonthTemps;
```

This declaration inverts the order of the indexes as presented above, but it also allows assignment of whole blocks between variables. Let's see how you can assign individual values:

```
Month1 [30][14] := 44;
Month1 [30, 13] := 55; // day, hour
Year1 [8, 30, 13] := 55; // month, day, hour
```

In theory, you should use the first line, selecting one of the array of arrays, and than an element of the resulting array. However, the version with the two indexes within square brackets is also allowed. Or with three indexes, in the “cube” example.

The importance of using intermediate types lies on the fact that arrays are type compatible only if they refer to the same exact type name (that is exactly the same type definition) not if their type definitions happen to refer to the same implementation. This type compatibility rule is the same for all types in the Object Pascal, with only some specific exceptions.

For example, the following statement copies a months's temperatures to the third month of the year:

```
Year1[3] := Month1;
```

Instead, a similar statement based on the stand alone array definitions (which are not type compatible):

```
AllYear1[3] := AllMonth1;
```

would cause the error:

```
Error: Incompatible types: 'array[1..31] of array[1..12] of Integer'
and 'TAllMonthTemps'
```

As I mentioned, static arrays suffer memory management issues, specifically when you want to pass them as parameters or allocate only a portion of a large array. Moreover, you cannot resize them during the lifetime of the array variable. This is why is is preferable to use dynamic arrays, even if they require a little extra management, for example regarding memory allocation.

Dynamic Arrays

In the traditional Pascal language arrays had a fixed-size arrays and you specified the number of elements of the array as you declared the data type. Object Pascal supports also a direct and native implementation of dynamic arrays.

---

note “Direct implementation of dynamic arrays” here is in contrast to using pointers and dynamic memory allocation to obtain a similar effect... with very complex and error-prone code.

Dynamic arrays are dynamically allocated and reference counted (making parameter passing much faster, as only the reference is passed, and not a copy of the complete array). When you are done, you can clear an array by setting its variable to `nil` or its length to zero, but given they are reference counted in most cases the compiler will automatically free the memory for you.

With a dynamic array, you declare an array type without specifying the number of elements and then allocate it with a given size using the `SetLength` procedure:

```
var
  Array1: array of Integer;
begin
  // this would cause a runtime Range Check error
  // Array1 [1] := 100;
  SetLength (Array1, 10);
  Array1 [1] := 100; // this is OK
```

You cannot use the array until you've assigned its length, allocating the required memory on the heap. If you do so, you'd either see a Range Check error (if the corresponding compiler option is active) or an Access Violation (on Windows) or similar memory access error on another platform. The `SetLength` call sets all the values to zero. The initialization code makes it possible to start reading and writing values of the array right away, without any fear of memory errors (unless you violate the array boundaries).

If you need to allocate memory explicitly, you don't need to free it directly. In the code snippet above, as the code terminates and the `Array1` variable goes out of scope, the compiler will automatically free its memory (in this case the ten integers that have been allocated). So while you can assign a dynamic array variable to nil or call `SetLength` with `0` value, this is generally not needed (and rarely done).

Notice that the `SetLength` procedure can also be used to resize an array, without loosing its current content (if you are growing it) or loosing it only partially (if you are shrinking it). As in the initial `SetLength` call you indicate only the number of elements of the array, the index of a dynamic array invariably starts from `0` and goes up to the number of elements minus `1`. In other words, dynamic arrays don't support two features of classic static Pascal arrays, the non-zero low bound and the non-integer indexes. At the same time, they match more closely how arrays work in most languages based on the C syntax.

Just like static arrays, to know about the current size of a dynamic array, you can use the `Length`, `High`, and `Low` functions. For dynamic arrays, however, `Low` always returns `0`, and `High` always returns the length minus one. This implies that for an empty array `High` returns `-1` (which, when you think about it, is a strange value, as it is lower than that returned by `Low`).

So, as an example, in the DynArray demo I've populated and extracted the information from a dynamic array using adaptable loops. This is the type and variable definition:

```
type
  TIntegersArray = array of Integer;

var
  IntArray1: TIntegersArray;
```

The array is allocated and populated with values matching the index, using the following loop:

```
var
  I: Integer;
begin
  SetLength (IntArray1, 20);
  for I := Low (IntArray1) to High (IntArray1) do
    IntArray1 [I] := I;
end;
```

A second button has the code both to display each value and compute the average, similar to the that of the previous example but in a single loop:

```
var
  I: Integer;
  total: Integer;
begin
  Total := 0;
  for I := Low(IntArray1) to High(IntArray1) do
  begin
    Inc (Total, IntArray1[I]);
    Show (I.ToString + ': ' + IntArray1[I].ToString);
  end;
  Show ('Average: ' + (Total / Length(IntArray1)).ToString);
end;
```

The output of this code is quite obvious (and mostly omitted):

```
0: 0
1: 1
2: 2
3: 3
...
17: 17
18: 18
19: 19
Average: 9.5
```

Beside `Length`, `SetLength`, `Low`, and `High`, there are also other common procedures that you can use on arrays, such as the `Copy` function, you can use to copy a portion of an array (or all of it). Notice that you can also assign an array from a variable to another, but in that case you are not making a full copy, but rather having two variables referring to the same array in memory.

The only slightly complex code is in the final part of the DynArray program, which copies one array to the other in two separate ways:

- using the Copy function, which duplicates the array data in a new data structure using a separate memory area
- using the assignment operator, which effectively creates an alias, a new variable referring to the same array in memory

At this point, if you modify one of the elements of the new arrays, you will affect the original version or not depending on the way you made the copy. This is the complete code:

```
var
  IntArray2: TIntegersArray;
  IntArray3: TIntegersArray;
begin
  // alias
  IntArray2 := IntArray1;

  // separate copy
  IntArray3 := Copy (IntArray1, Low(IntArray1), Length(IntArray1));

  // modify items
  IntArray2 [1] := 100;
  IntArray3 [2] := 100;

  // check values for each array
  Show (Format ('[%d] %d -- %d -- %d',
    [1, IntArray1 [1], IntArray2 [1], IntArray3 [1]]));
  Show (Format ('[%d] %d -- %d -- %d',
    [2, IntArray1 [2], IntArray2 [2], IntArray3 [2]]));
```

The output you'll get is like the following:

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

The changes to IntArray2 propagate to IntArray1, because they are just two references to the same physical array; the changes to IntArray3 are separate, because it has a separate copy of the data.

New Native Operations on Dynamic Arrays

There was a recent addition to dynamic arrays, making them even more a prime feature of the language, namely support for assigning constant arrays to dynamic arrays and for dynamic arrays concatenation.

note These extensions to dynamic arrays were added in Delphi XE7 and in the September 2014 release of Appmethod

In practice, you can write code like the following, which is significantly simplified from earlier code snippets:

```
var
  di: array of Integer;
  i: Integer;
begin
  di := [1, 2, 3];     // initialization
  di := di + di;       // concatenation
  di := di + [4, 5];   // mixed concatenation

  for i in di do
  begin
    Show (i.ToString);
  end;
```

Notice the use of a for-in statement to scan the array elements in this code, which is part of the `DynArrayConcat` application project. Notice that these arrays can be based on any data type, from simple integers like in the code above, to record and classes.

There is a second addition that was done along side with assignment and concatenation, but that is part of the RTL more than the language. It is not possible to use on dynamic arrays functions that were common for strings, like `Insert` and `Delete`.

This means you can now write code like the following (part of the same project):

```
var
  di: array of Integer;
  i: Integer;
begin
  di := [1, 2, 3, 4, 5, 6];
  Insert ([8, 9], di, 4);
  Delete (di, 2, 1); // remove the third (0-based)
```

Open Array Parameters

There is a very special scenario for the use of arrays, which is passing a flexible list of parameters to a function. Beside passing an array directly, there are two special syntax structures explained in this and the next section. An example of such a function, by the way, is the `Format` function that I called in the last code snippet and that has an array of values in square brackets as its second parameter.

Unlike the C language (and some of the other languages based on C syntax), in the traditional Pascal language a function or procedure always has a fixed number of

parameters. However, in Object Pascal there is a way to pass a varying number of parameters to a routine using as parameter an array, a technique known as *open array parameters*.

note Historically, open array parameters predate dynamic arrays, but today these two features look so similar in the way they work that they are almost indistinguishable these days. That's why I covered open array parameters only after discussing dynamic arrays.

The basic definition of an open array parameter is the same of a typed dynamic array type, prefixed by the `const` specifier. This means you indicate the type of the parameter(s), but you don't need to indicate how many elements of that type the array is going to have. Here is an example of such a definition, extracted form the `OpenArray` example:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

You can call this function by passing to it an *array-of-Integer* constant expression (which can also include variables as part of the expressions used to compute the individual values):

```
X := Sum ([10, Y, 27*I]);
```

Given a dynamic `array of Integer`, you can pass it directly to a routine requiring an open array parameter of the same base type (Integers in this case). Here is an example, where the complete array is passed as parameter:

```
var
  List: array of Integer;
  X, I: Integer;
begin
  // initialize the array
  SetLength (List, 10);
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

This is if you have a dynamic array. If you have a static array of the matching base type, you can also pass it to a functions expecting an open array parameter, or you can call the `Slice` function to pass only a portion of the existing array (as indicated by its second parameter). The following snippet (also part of the `OpenArray` example) shows how to pass a static array or a portion of it to the `Sum` function:

```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;

  // call
  X := Sum (List);
  Show (X.ToString);

  // pass portion of the array
  X := Sum (Slice (List, 5));
  Show (X.ToString);
```

Type-Variant Open Array Parameters

Besides these typed open array parameters, the Object Pascal language allows you to define type-variant or untyped open arrays. This special kind of array has an undefined number of elements, but also an undefined data type for those elements along with the possibility of passing elements of different types. This is one of the limited areas of the language that is not fully type safe.

Technically, the you can define a parameter of type `array of const` to pass an array with an undefined number of elements of different types to a function. For example, here is the definition of the `Format` function (we'll see how to use this function in Chapter 6, while covering strings, but I've already used it is some demos):

```
function Format (const Format: string;
  const Args: array of const): string;
```

The second parameter is an open array, which receives an undefined number of values. In fact, you can call this function in the following ways:

```
N := 20;
S := 'Total:';
Show (Format ('Total: %d', [N]));
Show (Format ('Int: %d, Float: %f', [N, 12.4]));
Show (Format ('%s %d', [S, N * 2]));
```

Notice that you can pass a parameter as either a constant value, the value of a variable, or an expression. Declaring a function of this kind is simple, but how do you code it? How do you know the types of the parameters? The values of a type-variant open array parameter are compatible with the `TVarRec` type elements.

note Do not confuse the `TVarRec` record with the `TVarData` record used by the `Variant` type. These two structures have a different aim and are not compatible. Even the list of possible types is different, because `TVarRec` can hold Object Pascal data types, while `TVarData` can hold Windows OLE data types. Variants are covered later in this chapter.

The following are the data types supported in a type-variant open array value and by the `TVarRec` record:

```
vtInteger       vtBoolean               vtChar
vtExtended      vtString                vtPointer
vtPChar         vtObject                vtClass
vtWideChar      vtPWideChar             vtAnsiString
vtCurrency      vtVariant               vtInterface
vtWideString    vtInt64                 vtUnicodeString
```

The record structure has a field with the type (`VType`) and variant field you can use to access the actual data (more about records in a few pages, even if this is an advanced usage for that construct).

A typical approach is to use a case statement to operate on the different types of parameters you can receive in such a call. In the `SumAll` function example, I want to be able to sum values of different types, transforming strings to integers, characters to the corresponding ordinal value, and adding 1 for True Boolean values. The code is certainly quite advanced (and it uses pointers dereferences), so don't worry if you don't fully understand it for now:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger:
        Result := Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

I've added this function to the `OpenArray` example, which calls it as follows:

```
var
  X: Extended;
  Y: Integer;
```

```
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34]);
  Show ('SumAll: ' + X.ToString);
end;
```

The output of this call adds the square of `Y`, the ordinal value of *K* (which is 107), 1 for the Boolean value, and the extended number, resulting in:

```
SumAll: 218.34
```

Record Data Types

While arrays define lists of identical items referenced by a numerical index, records define groups of elements of different types referenced by name. In other words, a record is a list of named items, or fields, each with a specific data type. The definition of a record type lists all these fields, giving each field a name used to refer to it.

note Records are available in most programming languages. They are defined with the `struct` keyword in the C language, while C++ has an extended definition including methods, much like Object Pascal has. Some more "pure" object-oriented languages have only the notion of class, not that of a record or structure.

Here is a small code snippet (from the `RecordsDemo` application project) with the definition of a record type, the declaration of a variable of that type, and few statements using this variable:

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
  Show ('Born in year ' + BirthDay.Year.ToString);
```

note The terms records at times is used in a rather loose way to refer to two different elements of the language: a record type definition and a variable of record type (or record instance). Record is used as a synonym of both record type and record instance.

There is way more to this data structure in Object Pascal than a simple list of fields, as the remaining part of this chapter will illustrate, but let's start with this *traditional* approach to records. The memory for a record is generally allocated on the stack for a local variable and in the global memory for a global one. This is highlighted by a call to `SizeOf`, which returns the number of bytes required by a variable or type, like in this statement:

```
Show ('Record size is ' + SizeOf (BirthDay).ToString);
```

which returns 8 (why it does return 8 and not 6–4 bytes for the Integer and two for each byte field—I'll discuss in the section “Fields Alignments”).

In other words, records are value types. This implies that if you assign a record to another, you are making a full copy. If you make a change to a copy, the original record won't be affected. This code snippets explains the concept in code terms:

```
var
  BirthDay: TMyDate;
  ADay: TMyDate;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;

  ADay := Birthday;
  ADay.Year := 2008;

  Show (MyDateToString (BirthDay));
  Show (MyDateToString (ADay));
```

The output (in Japanese or international date format) is:

```
1997.2.14
2008.2.14
```

The same copy operation takes place when you pass a record as parameter to a function, like in the `MyDateToString` I used above:

```
function MyDateToString (MyDate: TMyDate): string;
begin
  Result := MyDate.Year.ToString +  '.' +
    MyDate.Month.ToString +  '.' +
    MyDate.Day.ToString;
end;
```

Each call to this function involves a complete copy of the record's data. To avoid the copy, and to possibly make a change to the original record you have to explicitly use a reference parameter. This is highlighted by the following procedure, that makes some changes to a record passed as parameter:

```
procedure IncreaseYear (var MyDate: TMyDate);
begin
  Inc (MyDate.Year);
```

```
end;

var
  ADay: TMyDate;
begin
  ADay.Year := 2016;
  ADay.Month := 3;
  ADay.Day := 18;

  Increaseyear (ADay);
  Show (MyDateToString (ADay));
```

Given the `Year` field of the original record value is increased by the procedure call, the final output is one year later than the input:

```
2017.3.18
```

Using Arrays of Records

As I mentioned, arrays represent a data structure repeated several times, while records a single structure with different elements. Given these two type constructors are orthogonal, it is very common to use them together, defining arrays of records (while it is possible but uncommon to see records of arrays).

The array code is just like that of any other array, with each array element taking the size of the specific record type. While we'll see later how to use more sophisticated collection or container classes (for lists of elements), there is a lot in terms of data management you can achieve with arrays of records.

In the `RecordsTest` application project I've added an array of the `TMyDate` type, which can be allocated, initialized and used with code like the following:

```
var
  DatesList: array of TMyDate;
  I: Integer;
begin
  // allocate array elements
  SetLength (DatesList, 5);

  // assign random values
  for I := Low(DatesList) to High(DatesList) do
  begin
    DatesList[I].Year := 2000 + Random (50);
    DatesList[I].Month := 1 + Random (12);
    DatesList[I].Day := 1 + Random (27);
  end;

  // display the values
  for I := Low(DatesList) to High(DatesList) do
    Show (I.ToString + ': ' +
```

```
      MyDateToString (DatesList[I]));
```

Given the app uses random data, the output will be different every time, and could be like the following I've captured:

```
0: 2014.11.8
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

Variant Records

Since the early versions of the language, record types can also have a variant part; that is, multiple fields can be mapped to the same memory area, even if they have a different data type. (This corresponds to a *union* in the C language.) Alternatively, you can use these variant fields or groups of fields to access the same memory location within a record, but considering those values from different perspectives (in terms of data types). The main uses of this type were to store similar, but different data and to obtain an effect similar to that of typecasting (something used in the early days of the language, when direct typecasting was not allowed). The use of variant record types has been largely replaced by object-oriented and other modern techniques, although some system libraries use them internally in special cases.

The use of a variant record type is not type-safe and is not a recommended programming practice, particularly for beginners. You won't need to tackle them until you are really an Object Pascal expert, anyway... and that's why I decided to avoid showing you actual samples and covering this feature in more detail. If you really want a hint, have a look at the use of `TvarRec` I did in the demo of the section "Type-Variant Open Array Parameters".

Fields Alignments

Another advanced topic related with records is the way their fields are aligned, which also helps understand the actual size of a record. If you look into libraries, you'll often see the use of the `packed` keyword applied to records: this implies the record should use the minimum possible amount of bytes, even if this result in slower data access operations.

The difference is traditionally related to 16-bit or 32-bit alignment of the various fields, so that a byte followed by an integer might end up taking up 32 bits even if

only 8 are used. This is because accessing the following integer value on the 32-bit boundary makes the code faster to execute.

In general field alignment is used by data structures like records to improve the access speed to individual fields for some CPU architectures. There are different parameters you can apply to the `$ALIGN` compiler directive to change it.

With `{$ALIGN 1}` the compiler will save on memory usage by using all possible bytes, like when you use the `packed` specifier for a record. At the other extreme, the `{$ALIGN 16}` will use the largest alignment. Further options use 4 and 8 alignments.

As an example, if I go back to the `RecordsTest` project and add the keyword `packed` to the record definition:

```
type
  TMyDate = packed record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
```

the output to the call `SizeOf` will now return 6 rather than 8.

As a more advanced example, which you can skip if you are not already a fluent Object Pascal developer, let's consider the following structure (available in the `AlignTest` application project):

```
type
  TMyRecord = record
    c: Byte;
    w: Word;
    b: Boolean;
    I: Integer;
    d: Double;
  end;
```

With `{$ALIGN 1}` the structure takes 16 bytes (the value returned by `SizeOf`) and the fields will be at the following relative memory addresses:

```
c: 0 w: 1 b: 3 i: 4 d: 8
```

note Relative addresses are computed by allocating the record and computing the difference between the numeric value of a pointer to the structure and that of a pointer to the given field, with an expression like: `Integer(@MyRec.w) - Integer(@MyRec1)`. Pointers and the address of (`@`) operator are covered later in this chapter.

In contrast, if you change the alignment to 4 (which can lead to optimized data access) the size will be 20 bytes and the relative addresses:

```
c: 0 w: 2 b: 4 i: 8 d: 12
```

If you go to the extreme option and use `{$ALIGN 16}`, the structure requires 24 bytes and maps the fields as follow:

```
c: 0 w: 2 b: 4 i: 8 d: 16
```

What About the With Statement?

Another traditional language statement I failed to mention so far, because it is used only to work with records or classes, is the `with` statement. This keyword used to be peculiar to the Pascal syntax, but it was later introduced in JavaScript and Visual Basic. This is a keyword that can come up very handy to write less code, but it can also become very dangerous as it makes code far less readable. You'll find a lot of debate around the `with` statement, and I tend to agree this should be used sparingly, if at all. In any case, I felt it was important to include it in this book anyway (differently from `goto` statements).

note There is some debate about whether it will make sense to remove `goto` statements from the Object Pascal language, and it was also discussed whether to remove `with` from the mobile version of the language. While there are some legitimate usages, given the scoping problems `with` statements can cause, there are good reasons to discontinue this features (or change it so that an alias name is required as in C#).

The `with` statement is nothing but a shorthand. When you need to refer to a `record` type variable (or an object), instead of repeating its name every time, you can use a `with` statement. For example, while presenting the record type, I wrote this code:

```
var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 2008;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Using a `with` statement, I could modify the final part of this code, as follows:

```
  with BirthDay do
  begin
    Year := 2008;
    Month := 2;
    Day := 14;
  end;
```

This approach can be used in Object Pascal programs to refer to components and other classes. When you work with components or classes in general, the `with` statement allows you to skip writing some code, particularly for nested data structures.

So, why am I not encouraging the use of the `with` statement? The reason is it can least to subtle errors that are very hard to capture. While some of these hard-to-find errors are not easy to explain at this point of the book, let's consider a mild scenario, that can still lead to you scratching your head. This is a record and some code using it:

```
type
  TMyRecord = record
    MyName: string;
    MyValue: Integer;
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Record1: TMyRecord;
begin
  with Record1 do
  begin
    MyName := 'Joe';
    MyValue := 22;
  end;

  with Record1 do
    Show (Name + ': ' + MyValue.ToString);
```

Right? The application compiles and runs, but its output is not what you might expect (at least at first sight):

```
Form1: 22
```

The string part of the output is not the record value that was set earlier. The reason is that the second `with` statement erroneously uses the `Name` field, which is not the record field but another field that happens to be in scope (specifically the name of the form object the `Button2Click` method is part of).

If you had written:

```
Show (Record1.Name + ': ' + Record1.MyValue.ToString);
```

the compiler would have shown an error message, indicating the given record structure hasn't got a `Name` field.

In general, we can say that since the `with` statement introduces new identifiers in the current scope, we might hide existing identifiers, or wrongfully access another identifier in the same scope. This is a good reason for discouraging the use of the `with` statement. Even more you should avoid using multiple `with` statements, such as:

```
with MyRecord1, MyDate1 do...
```

The code following this would probably be highly unreadable, because for each field used in the block you would need to think about which record it refers to.

Records with Methods

In Object Pascal records are more powerful than in the original Pascal language or than structs are in the C language. Records, in fact, can have procedure and functions (called methods) associated with them. They can even redefine the language operators in custom ways (a feature called *operator overloading*), as you'll see in the next section.

A record with methods is somewhat similar to a class, as we'll find out later, with the most important difference being the way these two structures manage memory. Records in Object Pascal have two fundamental features of modern programming languages:

- **Methods**, which are functions and procedures connected with the record data structure and having direct access to the record fields. In other words, methods are function and procedures declared (or having a forward declaration) within the record type definition.
- **Encapsulation**, which is the ability to hide direct access to some of the fields (or methods) of a data structure from the rest of the code. You can obtain encapsulation using the `private` access specifier, while fields and methods visible to the outside as marked as `public`. The default access specifier for a record is `public`.

Now that you have the core concepts around extended records, let's look at the definition of a sample record, taken from the `RecordMethods` demo:

```
type
  TMyRecord = record
  private
    Name: string;
    Value: Integer;
    SomeChar: Char;
  public
    procedure Print;
    procedure SetValue (NewString: string);
    procedure Init (NewValue: Integer);
  end;
```

You can see the record structure is divided in two parts, private and public. You can have multiple sections, as the private and public keywords can be repeated as many times as you want, but a clear division of these two sections certainly helps readability. The methods are listed in the record definition (like in a class definition) without their complete code. In other words, the record has a forward declaration of the method.

How do you write the actual code of a method, its complete definition? Almost in the same way you'd code a global function or procedure. The differences are in the way

you write the method name, which is a combination of the record type name and the actual record name and on the fact you can directly refer to the fields and the other methods of the record directly, with no need to write the name of the record:

```
procedure TMyRecord.SetValue (NewString: string);
begin
  Name := NewString;
end;
```

In this code `one` is the local field and `aString` is the method's only parameter.

---

note While it might seem tedious having to write the definition of the method first and its full declaration next, you can use the Ctrl+Shift+C combination in the IDE editor to generate one from the other automatically. Also you can use the Ctrl+Shift+Up/Down Arrow keys to move from a method declaration to the corresponding definition and vice verse.

Here is the code of the other methods of this record type:

```
procedure TMyRecord.Init(NewValue: Integer);
begin
  Value := NewValue;
  SomeChar := 'A';
end;

function TMyRecord.ToString: string;
begin
  Result := Name + ' [' + SomeChar + ']: ' + Value.ToString;
end;
```

Here is a sample snippet of how you can use this record:

```
var
  MyRec: TMyRecord;
begin
  MyRec.Init(10);
  MyRec.SetValue ('hello');
  Show (MyRec.ToString);
```

As you might have guessed, the output will be:

```
hello [A]: 10
```

Now what if you want to use the fields from the code that uses the record, like in the snippet above:

```
  MyRec.Value := 20;
```

This actually compiles and works, which might be surprising as we declared the field in the private section, so that only the record methods can access to it. The truth is that in Object Pascal the `private` access specifier is actually enabled only between different units, so that line wouldn't be legal in a different unit, but can be used in the unit that originally defined the data type. As we will see, this is also true for classes.

Self: The Magic Behind Records

Suppose you have two records, like `myrec1` and `myrec2` of the same record type. When you call a method and execute its code, how does the method know which of the two copies of the record it has to work with? Behind the scenes, when you define a method the compiler adds a hidden parameter to it, a reference to the record you have applied the method to.

In other words, the call to the method above is converted by the compiler in something like:

```
// you write
MyRec.SetValue ('hello');

// the compiler generates
SetValue (@MyRec, 'hello');
```

In this pseudo code, the `@` is the *address of* operator, used to get the memory location of a record instance.

note Again, the *address of* operator is shortly covered at the end of this chapter in the (advanced) section titled "What About Pointers?"

This is how the calling code is translated, but how can the actual method call refer and use this hidden parameter? By implicitly using a special keyword called *self*. So the method's code could be written as:

```
procedure TMyRecord.SetValue (NewString: string);
begin
  self.Name := NewString;
end;
```

While this code compiles, it makes little sense to use `self` explicitly, unless you need to refer to the record as a whole, for example passing the record as parameter to another function. This happens more frequently with classes, which have the same exact hidden parameter for methods and the same `self` keyword.

One situation in which using an explicit `self` parameter can make the code more readable (even if it is not required) is when you are manipulating a second data structure of the same type, as in case you are testing a value from another instance:

```
function TMyRecord.IsSameName (ARecord: TMyRecord): Boolean;
begin
  Result := (self.Name = ARecord.Name);
end;
```

note The "hidden" self parameter is called `this` in C++ and Java, but it is called `self` in Objective-C (and in Object Pascal, of course).

Records and Constructors

When you define a variable of a record type (or a record instance) as a global variable its fields are initialized, but when you define one on the stack (as a local variable of a function or procedure, it isn't). So if you write code like this (also part of the `RecordMethods` project):

```
var
  MyRec: TMyRecord;
begin
  Show (MyRec.ToString);
```

its output will be more or less random. While the string is initialized to an empty string, the character field and the integer field will have the data that happened to be at the given memory location (just as it happens in general for a character or integer variable on the stack). In general, you'd get different output depending on the actual compilation or execution, such as:

```
[¥]: 1637580
```

That's why it is important to initialize a record (as most other variables) before using it, to avoid the risk of reading illogical data, which can even potentially crash the application.

Records support a special type of methods called constructors, that you can use to initialize the record data. Differently from other methods, constructors can also be applied to a record type to define a new instance (but they can still be applied to an existing instance).

This is how you can add a constructor to a record:

```
type
  TMyNewRecord = record
  private
    ...
  public
    constructor Create (NewString: string);
    function ToString: string;
    ...
```

The constructor is a method with code:

```
constructor TMyNewRecord.Create (NewString: string);
begin
  Name := NewString;
  Init (0);
end;
```

Now you can initialize a record with either of the two following coding styles:

```
var
  MyRec, MyRec2: TMyNewRecord;
```

```
begin
  MyRec := TMyNewRecord.Create ('Myself'); // class-like
  MyRec2.Create ('Myself'); // direct call
```

Notice that record constructors must have parameters: If you try with `Create()` you'll get the error message “Parameterless constructors not allowed on record types”.

note According to the documentation the definition of a parameterless constructor for records is reserved for the system (which has its way to initialize some of the records fields, such as strings and interfaces). This is why any user defined constructor must have at least one parameter. Of course, you can also have multiple overloaded constructors or multiple constructors with different names. I'll cover this in more detail when discussing constructors for classes.

Operators Gain New Ground

Another Object Pascal language feature related with records is operator overloading; that is, the ability to define your own implementation for standard operations (addition, multiplication, comparison, and so on) on your data types. The idea is that you can implement an add operator (a special `Add` method) and then use the + sign to call it. To define an operator you use `class operator` keyword combination.

note By reusing existing reserved words, the language designers managed to have no impact on existing code. This is something they've done quite often recently in keyword combinations like `strict private`, `class operator`, and `class var`.

The term *class* here relates to class methods, a concept we'll explore much later (in Chapter 12). After the directive you write the operator’s name, such as `Add`:

```
type
  TPointRecord = record
  public
    class operator Add (
      a, b: TPointRecord): TPointRecord;
```

The operator `Add` is then called with the + symbol, as you’d expect:

```
var
  a, b, c: TPointRecord;
begin
  ...
  c := a + b;
```

So which are the available operators? Basically the entire operator set of the language, as you cannot define brand new language operators:

- **Cast Operators**: `Implicit` and `Explicit`

- **Unary Operators**: `Positive`, `Negative`, `Inc`, `Dec`, `LogicalNot`, `BitwiseNot`, `Trunc`, and `Round`
- **Comparison Operators**: `Equal`, `NotEqual`, `GreaterThan`, `GraterThanOrEqual`, `LessThan`, and `LessThenOrEqual`
- **Binary Operators**: `Add`, `Subtract`, `Multiply`, `Divide`, `IntDivide`, `Modulus`, `ShiftLeft`, `ShiftRight`, `LogicalAnd`, `LogicalOr`, `LogicalXor`, `BitwiseAnd`, `BitwiseOr`, and `BitwiseXor`.

In the code calling the operator, you do not use these names but use the corresponding symbol. You use these special names only in the definition, with the `class operator` prefix to avoid any naming conflict. For example, you can have a record with an `Add` method and add an `Add` operator to it.

When you define these operators, you spell out the parameters, and the operator is applied only if the parameters match exactly. To add two values of different types, you'll have to specify two different `Add` operations, as each operand could be the first or second entry of the expression. In fact, the definition of operators provides no automatic commutativity. Moreover, you have to indicate the type very precisely, as automatic type conversions don't apply. Many times this implies overloading the overloaded operator and providing multiple versions with different types of parameters.

Another important factor to notice is that there are two special operators you can define for data conversion, `Implicit` and `Explicit`. The first is used to define an implicit type cast (or silent conversion), which should be perfect and not lossy. The second, `Explicit`, can be invoked only with an explicit type cast from a variable of a type to another given type. Together these two operators define the casts that are allowed to and from the given data type.

Notice that both the `Implicit` and the `Explicit` operators can be overloaded based on the function return type, which is generally not possible for overloaded methods. In case of a type cast, in fact, the compiler knows the expected resulting type and can figure out which is the typecast operation to apply. As an example, the OperatorsOver demo defines a record with a few operators:

```
type
  TPointRecord = record
  private
    x, y: Integer;
  public
    procedure SetValue (x1, y1: Integer);
    class operator Add (a, b: TPointRecord): TPointRecord;
    class operator Explicit (a: TPointRecord): string;
    class operator Implicit (x1: Integer): TPointRecord;
  end;
```

Here is the implementation of the methods of the record:

```
class operator TPointRecord.Add(
  a, b: TPointRecord): TPointRecord;
begin
  Result.x := a.x + b.x;
  Result.y := a.y + b.y;
end;

class operator TPointRecord.Explicit(
  a: TPointRecord): string;
begin
  Result := Format('(%d:%d)', [a.x, a.y]);
end;

class operator TPointRecord.Implicit(
  x1: Integer): TPointRecord;
begin
  Result.x := x1;
  Result.y := 10;
end;
```

Using such a record is quite straightforward, as you can write code like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: TPointRecord;
begin
  a.SetValue(10, 10);
  b := 30;
  c := a + b;
  Show (string(c));
end;
```

The second assignment (`b:=30;`) is done using the implicit operators, due to the lack of a cast while the `Show` call uses the cast notation to activate an explicit type conversion. Consider also that the operator `Add` doesn't modify its parameters; rather it returns a brand new value.

---

note The fact operators return new values is what make it harder to think of operator overloading for classes. If the operator creates a new dynamic objects who is going to dispose it? With the introduction of ARC in the mobile compiler, though, this feature was made available... but not for the desktop counterpart.

A little known fact is that it is technically possible to call an operator using its fully qualified internal name (like `&op_Addition`), prefixing it with an &, instead of using the operator symbol. For example, you can rewrite the records sum as follows (see the demo for the complete listing):

```
c := TPointRecord.&&op_Addition(a, b);
```

although I can see very few marginal cases in which you might want to do so. (The entire purpose of defining operators is to be able to use a friendlier notation than a method name, not an uglier one as the preceding direct call.)

Implementing Commutativity

Suppose you want implement the ability to add an integer number to one of your records. You can define the following operator (that is available in the code of the `OperatorsOver` application project, for a slightly different record type):

```
class operator TPointRecord2.Add(a: TPointRecord2;
  b: Integer): TPointRecord2;
begin
  Result.x := a.x + b;
  Result.y := a.y + b;
end;
```

note The reason I've defined this operator on a new type rather than the existing one is that the same structure already defines an `Implicit` conversion of an integer to the record type, so I can already add integers and records without defining a specific operator. This issue is explained better in the next section.

Now you can legitimately add a floating point value to a record:

```
var
  a: TPointRecord2;
begin
  a.SetValue(10, 20);
  a := a + 10;
```

However if you try to write the opposite addition:

```
  a := 30 + a;
```

this will fail with the error:

```
[dcc32 Error] E2015 Operator not applicable to this operand type
```

In fact, as I mentioned, commutativity is not automatic for operators applied to variables of different types, but must be specifically implemented either repeating the call or calling (like below) the other version of the operator:

```
class operator TPointRecord2.Add(b: Integer;
  a: TPointRecord2): TPointRecord2;
begin
  Result := a + b; // implement commutativity
end;
```

Implicit Cast and Type Promotions

It is important to notice that the rules related to the resolution of calls involving operators are different from the traditional rules involving methods. With automatic type promotions there's the chance that a single expression will end up calling different versions of an overloaded operator and cause ambiguous calls. This is why you need to take a lot of care when writing `Implicit` operators.

Consider these expressions from the previous example:

```
a := 50;
c := a + 30;
c := 50 + 30;
c := 50 + TPointRecord(30);
```

They are all legal! In the first case, the compiler converts 30 to the proper record type, in the second the conversion takes place after the assignment, and in the third the explicit cast forces an implicit one on the first value, so that the addition being performed is the custom one among records. In other words the result of the second operation is different from the other two, as highlighted in the output and in the expanded version of these statements:

```
// output
(80:20)
(80:10)
(80:20)

// expanded statements
c := a + TPointRecord(30);
// that is: (50:10) + (30:10)

c := TPointRecord (50 + 30);
// that is 80 converted into (80:10)

c := TPointRecord(50) + TpointRecord(30);
// that is: (50:10) + (30:10)
```

Variants

Originally introduced in the language to provide full Windows OLE and COM support, Object Pascal has the concept of a *loosely typed* native data type called `Variant`. Although the name reminds of variant records (mentioned earlier) and the implementation has some similarity with open array parameters, this is a separate feature with a very specific implementation (uncommon in languages outside of the Windows development world).

In this section I won't really refer to OLE and other scenarios in which this data type is used (like fields access for data sets. I'll get back to dynamic types, RTTI, and reflection in Chapter 16, where I'll also cover a related (but type safe) type called `TValue`. Here I want to discuss this data type from a general perspective.

Variants Have No Type

In general, you can use a variable of the variant type to store any of the basic data types and perform numerous operations and type conversions. Automatic type conversions go against the general type-safe approach of the Object Pascal language and is an implementation of a type of dynamic typing originally introduced by languages like Smalltalk and Objective-C, and recently made popular in scripting languages including JavaScript, PHP, Python, and Ruby.

A variant is type-checked and computed at run time. The compiler won't warn you of possible errors in the code, which can be caught only with extensive testing. On the whole, you can consider the code portions that use variants to be interpreted code, because, as with interpreted code, many operations cannot be resolved until run time. In particular this affects the speed of the code.

Now that I've warned you against the use of the `Variant` type, it is time to look at what you can do with it. Basically, once you've declared a variant variable such as the following:

```
var
  V: Variant;
```

you can assign values of several different types to it:

```
V := 10;
V := 'Hello, World';
V := 45.55;
```

Once you have the variant value, you can copy it to any compatible or incompatible data type. If you assign a value to an incompatible data type, the compiler will generally not flag it with an error, but will perform a runtime conversion if this makes sense. Otherwise it will issue a run-time error. Technically a variant stores type information along with the actual data, allowing a number of handy, but slow and unsafe, run-time operations.

Consider the following code (part of the `VariantTest` application project), which is an extension of the code above:

```
var
  V: Variant;
  S: string;
begin
```

```
V := 10;
S := V;
V := V + S;
Show (V);

V := 'Hello, World';
V := V + S;
Show (V);

V := 45.55;
V := V + S;
Show (V);
```

Funny, isn't it? This is the output (not surprisingly):

```
20
Hello, World10
55.55
```

Besides assigning a variant holding a string to the `S` variable, you can assign to it a variant holding an integer or a floating-point number. Even *worse*, you can use the variants to compute values, with the operation `V := V + S;` that gets interpreted in different ways depending on the data stored in the variant. In the code above, that same line can add integers, floating point values, or concatenate strings.

Writing expressions that involve variants is risky, to say the least. If the string contains a number, everything works. If not, an exception is raised. Without a compelling reason to do so, you shouldn't use the `Variant` type; stick with the standard Object Pascal data types and type-checking approach.

Variants in Depth

For those interested in understanding variants in more details, let me add some technical information about how variants work and how you can have more control on them. The RTL includes a variant record type, `TVarData`, which has the same memory layout as the `Variant` type. You can use this to access the actual type of a variant. The `TVarData` structure includes the type of the Variant, indicated as `VType`, some reserved fields, and the actual value.

note For more details look to the TVarData definition in the RTL source code, in the System unit. This is far from a simple structure and I recommend only developers with some experience look into the implementation details of the variant type.

The possible values of the `VType` field correspond to the data types you can use in OLE automation, which are often called OLE types or *variant types*. Here is a complete alphabetical list of the available variant types:

```
varAny          varArray        varBoolean
varByte         varByRef        varCurrency
varDate         varDispatch     varDouble
varEmpty        varError        varInt64
varInteger      varLongWord     varNull
varOleStr       varRecord       varShortInt
varSingle       varSmallint     varString
varTypeMask     varUInt64       varUnknown
varUString      varVariant      varWord
```

Most of these constant names of variant types are easy to understand. Notice that there is the concept of *null value*, you obtain by assigning `NULL` (and not `nil`).

There are also many functions for operating on variants that you can use to make specific type conversions or to ask for information about the type of a variant (see, for example, the `VarType` function). Most of these type conversion and assignment functions are actually called automatically when you write expressions using variants. Other variant support routines actually operate on variant arrays, again a structure used almost exclusively for OLE integration on Windows.

Variants Are Slow!

Code that uses the `Variant` type is slow, not only when you convert data types, but even when you simply add two variant values holding integers. They are almost as slow as interpreted code. To compare the speed of an algorithm based on variants with that of the same code based on integers, you can look at the second button of the VariantTest project.

This program runs a loop, timing its speed and showing the status in a progress bar. Here is the first of the two very similar loops, based on Int64 and variants:

```
const
  maxno = 10000000;  // 10 million

var
  time1, time2: TDateTime;
  n1, n2: Variant;
begin
  time1 := Now;
  n1 := 0;
  n2 := 0;

  while n1 < maxno do
  begin
    n2 := n2 + n1;
    Inc (n1);
  end;

  // we must use the result
```

```
time2 := Now;
Show (n2);
Show ('Variants: ' + FormatDateTime (
  'ss.zzz', Time2-Time1) + ' seconds');
```

The timing code is worth looking at, because it's something you can easily adapt to any kind of performance test. As you can see, the program uses the `Now` function to get the current time and the `FormatDateTime` function to output the time difference, showing only the seconds (*"ss"*) and the milliseconds (*"zzz"*).

In this example the speed difference is actually so great that you'll notice it even without precise timing:

```
49999995000000
Variants: 01.169 seconds
49999995000000
Integers: 00.026 second
```

These are the numbers I get on my Windows virtual machine, and that's about 50 times slower for the variant based code. The actual values depend on the computer you use to run this program, but the relative difference won't change much. Even on my Android phone I get a similar proportion (but much longer times overall):

```
49999995000000
Variants: 07.717 seconds
49999995000000
Integers: 00.157 second
```

On my phone this code takes 6 times as much as on Windows, but now the fact is the net different is over 7 seconds, making the variant based implementation noticeably slow to the user, while the Int64 based one is still extremely fast (a user would hardly notice a tenth of a second).

What About Pointers?

Another fundamental data type of the Object Pascal language is represented by pointers. Some of the object-oriented languages have gone a long way to hide this powerful, but dangerous, language construct, while Object Pascal lets a programmer use it when needed (which is generally not very often).

But what is a pointer, and where does its name come from? Differently than most other data types, a pointer doesn't hold an actual value, but it holds an indirect reference to a variable, which in turn has a value. A more technical way to express this is that a pointer type defines a variable that holds the memory address of another variable of a given data type (or of an undefined type).

note This is an advanced section of the book, added here because pointers are part of the Object Pascal language and should be part of the core knowledge of any developer, although it is not a basic topic and if you are new to the language you might want to skip this section the first time you read the book. Again, there is a chance you might have used programming languages with no (explicit) pointers, so this short section could be an interesting read!

The definition of a pointer type is not based on a specific keyword, but uses a special symbol, the caret (^). For example you can define a pointer to variable of the Integer type with the following declaration:

```
type
  TPointerToInt = ^Integer;
```

Once you have defined a pointer variable, you can assign to it the address of another variable of the same type, using the `@` operator:

```
var
  P: ^Integer;
  X: Integer;
begin
  X := 10;
  P := @X;
  // change the value of X using the pointer
  P^ := 20;
  Show ('X: ' + X.ToString);
  Show ('P^: ' + P^.ToString);
  Show ('P: ' + Integer(P).ToHexString (8));
```

This code is part of the `PointersTest` application project. Given the pointer P refers to the variable X, you can use P^ to refer to the value of the variable, and read or change it. You can also display the value of the pointer itself, that is the memory address of X, by casting the pointer to an integer. Rather than showing the plain numeric value, the code shows the hexadecimal representation, which is more common for memory addresses. This is the output (where the pointer address might depend on the specific compilation):

```
X: 20
P^: 20
P: 0018FC18
```

warn Casting the pointer to an Integer is correct code only on 32-bit platforms, not on 64-bit ones. A better option is to cast to NativeInt, however that type lacks the integer helpers, and that would have made the sample code more complex. So the code of this demo is 32-bit specific.

Let me summarize, for clarity. When you have a pointer `P`:

- By using the pointer directly (with the expression `P`) you refer to the address of the memory location the pointer is referring to

- By dereferencing the pointer (with the expression `P^`) you refer to the actual content of that memory location

Instead of referring to an existing memory location, a pointer can also refer to a new and specific memory block dynamically allocated on the heap with the `New` procedure. In this case, when you don't need the value accessed by the pointer anymore, you'll also have to get rid of the memory you've dynamically allocated, by calling the `Dispose` procedure.

note Memory management in general and the way the heap works in particular are covered in Chapter 13. In short, the heap is a (large) area of memory in which you can allocate and release blocks of memory in no given order. As an alternative to `New` and `Dispose` you can use `GetMem` and `FreeMem`, but the former two are preferable and safer to use.

Here is a code snippet that allocates memory dynamically:

```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  Show (P^.ToString);
  // termination
  Dispose (P);
```

If you don't dispose of the memory after using it, your program may eventually use up all the available memory and crash. The failure to release memory you don't need any more is known as a *memory leak*.

note To be safer the code above should indeed use an exception handling `try-finally` block, a topic I decided not to introduce at this point of the book, but I'll cover later in Chapter 9.

If a pointer has no value, you can assign the `nil` value to it. You can test whether a pointer is `nil` to see if it currently refers to a value with a direct equality test or by using the specific `Assigned` function as shown below.

This kind of test is often used, because dereferencing (that is accessing the value at the memory address stored in the pointer) an invalid pointer causes a memory access violation (with slightly different effects depending on the operating system):

```
var
  P: ^Integer;
begin
  P := nil;
  Show (P^.ToString);
```

You can see an example of the effect of this code by running the PointersTest application. The error you'll see (on Windows) should be similar to:

```
Access violation at address 0080B14E in module 'PointersTest.exe'.
Read of address 00000000.
```

One of the ways to make pointer data access safer, is to add a “pointer is not null” safe-check like the following:

```
if P <> nil then
  Show (P^.ToString);
```

As I mentioned earlier, an alternative way, which is generally preferable for readability reasons, is to use the `Assigned` pseudo-function:

```
if Assigned (P) then
  writeln (P^.ToString);
```

note `Assigned` is not a real function, because it is “resolved” by the compiler producing the proper code. Also, it can be used over a procedural type variable (or method reference) without actually invoking it, but only checking if it is assigned.

Object Pascal also defines a `Pointer` data type, which indicates untyped pointers (such as `void*` in the C language). If you use an untyped pointer you should use `GetMem` instead of `New` (indicating the number of bytes to allocate (given this value cannot be inferred from the type itself). The `GetMem` procedure is required each time the size of the memory variable to allocate is not defined.

The fact that pointers are seldom necessary in Object Pascal is an interesting advantage of this language. Still, having this feature available can help in implementing some extremely efficient low level functions and when calling the API of an operating system. In any case, understanding pointers is important for advanced programming and for a full understanding of language object model, which use pointers (generally called references) behind the scenes.

File Types, Anyone?

The last Object Pascal data type constructor covered (briefly) in this chapter is the *file* type. File types represent physical disk files, certainly a peculiarity of the original Pascal language, given very few old or modern programming languages include the notion of a file as a primitive data type. The Object Pascal language has a `file` keyword, which is a type specifier, like `array` or `record`. You use `file` to define a new type, and then you can use the new data type to declare new variables:

```
type
  IntFile = file of Integers;
var
  IntFile1: IntFile;
```

It is also possible to use the `file` keyword without indicating a data type, to specify an untyped file. Alternatively, you can use the `TextFile` type, defined in the `System` unit of the Run Time Library to declare files of ASCII characters (or, more correctly in these times, files of bytes).

Direct use of files, although still supported, is less and less common these days, as the Run Time Library includes many classes for managing binary and text files at a much higher level (including the support for Unicode encodings for text files, for example). Object Pascal applications generally use the RTL streams (the `TStream` and derived classes) to handle any complex file read and write operations. Streams represent virtual files, which can be mapped to physical files, to a memory block, to a socket, or any other continuous series of bytes.

One area when you still see some of the old time file management routines in use is when you write console applications, where you can use `write`, `writeln`, `read`, and related function for operating with a special file, which is the standard input and standard output (C and C++ have similar support for input and output from the console, and many other languages offer similar services).

06: all about strings

Character strings are one of the most commonly used data types in any programming language. Object Pascal makes string handling fairly simple, yet very fast and extremely powerful. Even if the basics of strings are easy to grasp and I've used strings for output in the previous chapters, behind the scenes the situation is a little more complex than it might seem at first sight. Text manipulation involves several closely related topics worth exploring: to fully understand string processing you need to know about Unicode representations, understand how strings map to arrays of characters, and learn about some of the most relevant string operations of the run time library, including saving string to text files and loading them.

Object Pascal has several options for string manipulation and makes available different data types and approaches. The focus of the chapter will be on the standard string data type, but I'll also devote a little time to older string types you can still use in the desktop compiler (but not in the mobile ones). Before we get to that, though, let me start from the beginning: the Unicode representation.

Unicode: An Alphabet for the Entire World

Object Pascal string management is centered around the Unicode character set and, particularly, the use of one of its representations, called UTF-16. Before we get to the technical details of the implementation, it is worth devoting a few sections to fully understanding the Unicode standard.

The idea behind Unicode (which is what makes it simple and complex at the same time) is that every single character in all known alphabets of the world has its own description, a graphical representation, and a unique numeric value (called a Unicode *code point*).

---

note The reference web site of the Unicode consortium is http://www.unicode.org, which a rich amount of documents. The ultimate reference is "The Unicode Standard" book, which can be found at http://www.unicode.org/book/aboutbook.html.

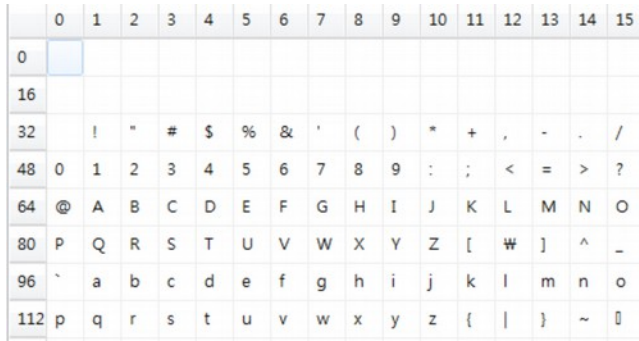
Not all developers are familiar with Unicode, and many still think of characters in terms of older, limited representations like ASCII and in terms of ISO encoding. By having a short section covering these older standards, you'd better appreciate the peculiarities (and the complexity) of Unicode.

Characters from the Past: from ASCII to ISO Encodings

Character representations started with the American Standard Code for Information Interchange (ASCII), which was developed in the early '60s as a standard encoding of computer characters, encompassing the 26 letters of the English alphabet, both lowercase and uppercase, the 10 numerical digits, common punctuation symbols, and a number of control characters (still in use today).

ASCII uses a 7 bit encoding system to represent 128 different characters. Only characters between #32 (Space) and #126 (Tilde) have a visual representation, as show in Figure 6.1 (extracted from an Object Pascal application running on Windows).

Figure 6.1: A table with the printable ASCII character set



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	₩	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	▯

While ASCII was certainly a foundation (with its basic set of 128 characters that are still part of the core of Unicode), it was soon superseded by extended versions that used the 8th bit to add another 128 characters to the set.

Now the problem is that with so many languages around the world, there was no simple way to figure out which other characters to include in the set (at times indicated as ASCII-8). To make the story short, Windows adopted a different set of characters, called a *code page*, with a set of characters depending on your locale configuration and version of Windows. Beside Windows code pages there are many other standards based on a similar *paging* approach, and those pages became part of international ISO standards.

The most relevant was certainly the ISO 8859 standard, which defines several *regional* sets. The most used set (well, the one used in most Western countries to be a little more precise) is the Latin set, referenced as ISO 8859-1.

note Even if partially similar, Windows 1252 code page doesn't fully conform to the ISO 8859-1 set. Windows adds extra characters like the € symbol, extra quotation marks, and more, in the area from 128 to 150. Differently from all other values of the Latin set, those Windows extensions do not conform with the Unicode code points.

Unicode Code Points and Graphemes

If I really want to be precise, I should include one more concept beyond that of code points. At times, in fact, multiple code points could be used to represent a single *grapheme* (a visual character). This is generally not a letter, but a combination of letters or letters and symbols. For example, if you have a sequence of the code point

representing the Latin letter *a* (`#$0061`) followed by the code point representing the grave accent (`#$0300`), this should be displayed as a single accented character.

In Object Pascal coding terms, if you write the following (part of the `CodePoints` application project), the message will have one single accented character, as in Figure 6.2.

```
var
  str: String;
begin
  str := #$0061 + #$0300;
  ShowMessage (str);
```

Figure 6.2: A single grapheme can be the result of multiple code points

In this case we have two characters, representing two code points, but only one grapheme (or visual elements). The fact is that while in the Latin alphabet you can use a specific Unicode code point to represent the given grapheme (*letter a with grave accent is code point $00E0*), in other alphabets combining Unicode code points is the only way to obtain a given grapheme (and the correct output).

Even if the display is that of an accented character, there is no automatic normalization or transformation of the value (only a proper display), so the string internally remains different from one with the character à.

note The rendering of graphemes from multiple code points might depend on specific support from the operating system and on text rendering techniques being used, so you might find out that for some of the graphemes not all operating systems offer the correct output.

From Code Points to Bytes (UTF)

While ASCII used a direct and easy mapping of character to their numeric representation, Unicode uses a more complex approach. As I mentioned, every element of the Unicode alphabet has an associated code point, but the mapping to the actual representation is often more complicated.

One of the elements of confusion behind Unicode is that there are multiple ways to represent the same code point (or Unicode character numerical value) in terms of actual storage, of physical bytes, in memory or on a file. The issue stems from the fact that the only way to represent all Unicode code points in a simple and uniform way is to use four bytes for each code point. This accounts for a fixed-length representation (each character requires always the same amount of bytes), but most developers would perceive this as too expensive in memory and processing terms.

note In Object Pascal the Unicode Code Points can be represented directly in a 4-bytes representation by using the `UCS4Char` data type.

That's why the Unicode standard defines other representations, generally requiring less memory, but in which the number of bytes for each symbol is different, depending its code point. The idea is to use a smaller representation for the most common elements, and a longer one for those infrequently encountered.

The different physical representations of the Unicode code points are called Unicode Transformation Formats (or UTF). These are algorithmic *mappings*, part of the Unicode standard, that map each code point (the absolute numeric representation of a character) to a unique sequence of bytes representing the given character. Notice that the mappings can be used in both directions, converting back and forth between different representations.

The standard defines three of these formats, depending on how many bits are used to represent the initial part of the set (the initial 128 characters): 8, 16, or 32. It is interesting to notice that all three forms of encodings need at most 4 bytes of data for each code point.

- **UTF-8** transforms characters into a variable-length encoding of 1 to 4 bytes. UTF-8 is popular for HTML and similar protocols, because it is quite compact when most characters (like tags in HTML) fall within the ASCII subset.
- **UTF-16** is popular in many operating systems (including Windows and Mac OS X) and development environments. It is quite convenient as most characters fit in two bytes, reasonably compact, and fast to process.
- **UTF-32** makes a lot of sense for processing (all code points have the same length), but it is memory consuming and has limited use in practice.

There is a common misconception that UTF-16 can map directly all code points with two bytes, but since Unicode defines over 100,000 code points you can easily figure out they won't fit into 64K elements. It is true, however, that at times developers use only a subset of Unicode, to make it fit in a 2-bytes-per-characters fixed-length representation. In the early days, this subset of Unicode was called UCS-2, now you

often see it referenced as Basic Multilingual Plane (BMP). However, this is only a subset of Unicode (one of many *planes*).

note A problem relating to multi-byte representations (UTF-16 and UTF-32) is which of the bytes comes first? According to the standard, all forms are allowed, so you can have a UTF-16 BE (big-endian) or LE (little-endian), and the same for UTF-32. The big-endian byte serialization has the most significant byte first, the little-endian byte serialization has the least significant byte first. The bytes serialization is often marked in files along with the UTF representation with a header called Byte Order Mark (BOM).

The Byte Order Mark

When you have a text file storing Unicode characters, there is a way to indicate which is the UTF format being used for the code points. The information is stored in a header or marker at the beginning of the file, called Byte Order Mark (BOM). This is a signature indicating the Unicode format being used and the byte order form (little or big endian – LE or BE). The following table provides a summary of the various BOMs, which can be 2, 3, or 4 bytes long:

`00 00 FE FF`	UTF-32, big-endian
`FF FE 00 00`	UTF-32, little-endian
`FE FF`	UTF-16, big-endian
`FF FE`	UTF-16, little-endian
`EF BB BF`	UTF-8

We'll see later in this chapter how Object Pascal manages the BOM within its streaming classes. The BOM appears at the very beginning of a file with the actual Unicode data immediately following it. So a UTF-8 file with the content *AB* contains five hexadecimal values (3 for the BOM, 2 for the letters):

```
EF BB BF 41 42
```

If a text file has none of these signatures, it is generally considered as an ASCII text file, but it might as well contain text with any encoding.

note On the other hand, when you are receiving data from a web request or through other Internet protocols, you might have a specific header (part of the protocol) indicating the encoding, rather than relying on a BOM.

Looking at Unicode

How do we create a table of Unicode characters like those I displayed earlier for ASCII ones? We can start by displaying code points in the Basic Multilingual Plane (BMP), excluding what are called surrogate pairs.

---

note Not all numeric values are true UTF-16 code points, since there are some non-valid numerical values for characters (called surrogates) used to form a paired code and represent code points above 65535. A good example of a surrogate pair is the symbol used in music scores for the F (or bass) clef, 𝄢. It is code point 1D122 which is represented in UTF-16 by two values, D834 followed by DD22.

Displaying all of the elements of the BMP would requires a 256 * 256 grid, hard to accommodate on screen. This is why the `ShowUnicode` application project has a tab with two pages: The first tab has the primary selector with 256 blocks, while the second page shows a grid with the actual Unicode elements, one section at a time. This program has a little more of a user interface than most others in the book, and you can as well skim through its code if you are only interested in its output (and not the internals).

When the program starts, it fills the ListView control in the first page of the TabControl with 256 entries, each indicating the first and last character of a group of 256. Here is the actual code of the `OnCreate` event handler of the form and a simple function used to display each element, while the corresponding output is in Figure 6.2:

```
// helper function
function GetCharDescr (nChar: Integer): string;
begin
  if Char(nChar).IsControl then
    Result := 'Char #' + IntToStr (nChar) + ' [  ]'
  else
    Result := 'Char #' + IntToStr (nChar) +
      ' [' + Char (nChar) + ']';
end;

procedure TForm2.FormCreate(Sender: TObject);
var
  I: Integer;
  ListItem: TListViewItem;
begin
  for I := 0 to 255 do // 256 pages * 256 characters each
  begin
    ListItem := ListView1.Items.Add;
    ListItem.Tag := I;
    if (I < 216) or (I > 223) then
      ListItem.Text :=
        GetCharDescr(I*256) + '/' + GetCharDescr(I*256+255)
    else
```

```
      ListItem.Text := 'Surrogate Code Points';
  end;
end;
```

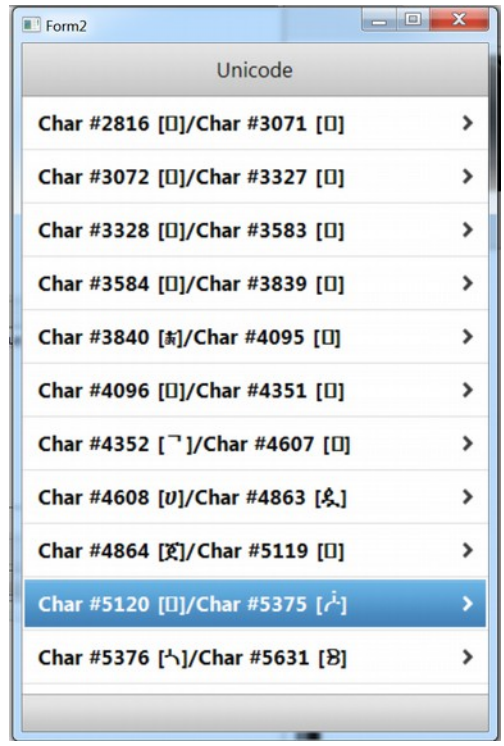


Figure 6.2:
The first page of the ShowUnicode application project has a long list of sections of Unicode characters

Notice how the code saves the number of the "page" in the `Tag` property of the items of the ListView, an information used later to fill a page. As a user selects one of the items, the application moves to the second page of the TabControl, filling its string grid with the 256 characters of the section:

```
procedure TForm2.ListView1ItemClick(const Sender: TObject;
  const AItem: TListViewItem);
var
  I, NStart: Integer;
begin
  NStart := AItem.Tag  * 256;
  for I := 0 to 255 do
  begin
    StringGrid1.Cells [I mod 16, I div 16] :=
      IfThen (not Char(I + NStart).IsControl, Char (I + NStart), '');
  end;
```

```
TabControl1.ActiveTab := TabItem2;
```

The `IfThen` function used in the code above is a two way test: If the condition passed in the first parameter is true, the function returns the value of the second parameter; if not, it returns the value of the third one. The test in the first parameter uses the `IsControl` method of the Char type helper, to filter out non-printable control characters.

note The `IfThen` function operates more or less like the `?:` operator of most programming languages based on the C syntax. There is a version for strings and a separate one for Integers. For the string version you have to include the `System.StrUtils` unit, for the Integer version of `IfThen` the `System.SysUtils` unit.

The grid of Unicode characters produced by the application is visible in Figure 6.3. Notice that the output varies depending on the ability of the selected font and the specific operating system to display a given Unicode character.

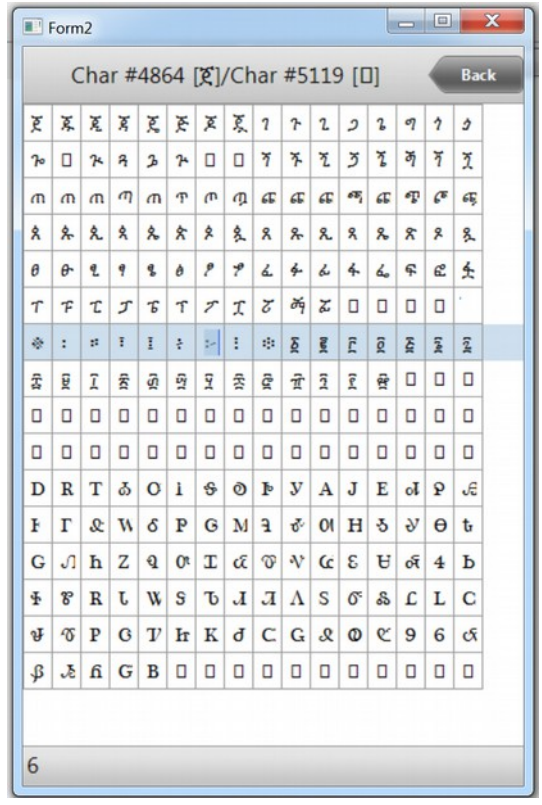


Figure 6.3: The second page of the ShowUnicode application project has some of the actual Unicode characters

The Char Type Revisited

After this introduction to Unicode, let's get back to the real topic of this chapter, which is how the Object Pascal language manages characters and strings. I introduced the Char data type in Chapter 2, and mentioned some of the type helper functions available in the Character unit. Now that you have a better understanding of Unicode, it is worth revisiting that section and going though some more details.

First of all, the Char type does not invariably represent a Unicode code point. The data type, in fact, uses 2 bytes for each element. While it does represent a code point for elements in Unicode'e Basic Multi-language Plane (BMP), a Char can also be part of a pair of surrogate values, representing a code point.

Technically, there is a different type you could use to represent any Unicode code point directly, and this is the UCS4Char type, which used 4 bytes to represent a value). This type is rarely used, as the extra memory required is generally hard to justify, but you can see that the Character unit (cover next) includes also several operations for this data type.

Back to the Char type, remember it is an enumerated type (even if a rather large one), so it has the notion of sequence and offers code operations like `Ord`, `Inc`, `Dec`, `High`, and `Low`. Most extended operations, including the specific type helper, are not part of the basic system RTL units but require the inclusion of the Character unit.

Unicode Operations With The Character Unit

Most of the specific operations for Unicode characters (and also Unicode strings, of course) are defined in a special units called `System.Character`. This unit defines the `TCharHelper` helper for the Char type, which lets you apply operations directly to the type.

---

note The Character unit defines also a `TCharacter` record, which is basically a collection of static class functions, plus a number of global routines mapped to these method. These are older, deprecated functions, given that now the preferred way to work on the Char type at the Unicode level is the use of the class helper.

The unit also defines two interesting enumerated types. The first is called `TUnicodeCategory` and maps the various characters in broad categories like control, space, uppercase or lowercase letter, decimal number, punctuation, math symbol, and many more. The second enumeration is called `TUnicodeBreak` and defines the family of the various spaces, hyphen, and breaks. If you are used to ASCII opera-

tions, this is a big change. Numbers in Unicode are not only the characters between 0 and 9; spaces are not limited to the character #32; and so on for many other assumption of the (much simpler) 256-elements alphabet.

The Char type helper has over 40 methods that comprise many different tests and operations. They can be used for:

- Getting the numeric representation of the character (`GetNumericValue`).
- Asking for the category (`GetUnicodeCategory`) or checking it against one of the various categories (`IsLetterOrDigit`, `IsLetter`, `IsDigit`, `IsNumber`, `IsControl`, `IsWhiteSpace`, `IsPunctuation`, `IsSymbol`, and `IsSeparator`). I used the `IsControl` operation in the previous demo.
- Checking if it is lowercase or uppercase (`IsLower` and `IsUpper`) or converting it (`ToLower` and `ToUpper`).
- Verifying if it is part of a UTF-16 surrogate pair (`IsSurrogate`, `IsLowSurrogate`, and `IsHighSurrogate`) and convert surrogate pairs in various ways.
- Converting it to and from UTF32 (`ConvertFromUtf32` and `ConvertToUtf32`) and UCS4Char type (`ToUCS4Char`).
- Checking if it is part of a given list of characters (`IsInArray`).

Notice that some of these operations can be applied to the type as a whole, rather than to a specific variable. In that can you have to call them using the Char type as prefix, as in the second code snippet below.

To experiment a bit with these operations on Unicode characters, I've create an application project called `CharTest`. One of the examples of this demo is the effect of calling uppercase and lowercase operations on Unicode elements. In fact, the classic UpCase function of the RTL works only for the base 26 English language characters of the ANSI representation, while it fails some Unicode character that do have a specific uppercase representations (not all alphabets have the concept of uppercase, so this is not a universal notion).

To test this scenario, in the `CharTest` application project I've added the following snippet that tries to convert an accented letter to uppercase:

```
var
  ch1: Char;
begin
  ch1 := 'ù';
  Show ('UpCase ù: ' + UpCase(ch1));
  Show ('ToUpper ù: ' + ch1.ToUpper);
```

The traditional `Upcase` call won't convert the latin accented character, while the `ToUpper` function works properly:

```
UpCase ù: ù
```

```
ToUpper ù: Ù
```

There are many Unicode-related features in the Char type helper, like those highlighted in the code below, which defines a string as including also a character outside of the BMP (the low 64K of Unicode code points). The code snippet, also part of the `CharTest` application project, has a few tests on the various elements of the string, all returning True:

```
var
  str1: string;
begin
  str1 := '1.' + #9 + Char.ConvertFromUtf32 (128) +
    Char.ConvertFromUtf32($1D11E);
  ShowBool (str1.Chars[0].IsNumber);
  ShowBool (str1.Chars[1].IsPunctuation);
  ShowBool (str1.Chars[2].IsWhiteSpace);
  ShowBool (str1.Chars[3].IsControl);
  ShowBool (str1.Chars[4].IsSurrogate);
end;
```

The display function used in this case is an adapted version:

```
procedure TForm1.ShowBool(value: Boolean);
begin
  Show(BoolToStr (Value, True));
end;
```

note Unicode code point $1D11E is *musical symbol G clef*.

Unicode Character Literals

We have seen in several examples that you can assign an individual character literal or a string literal to a variable of the string type. In general using the numeric representation of a character with the # prefix is quite simple. There are some exceptions, though.

For backwards compatibility, plain character literals are converted depending on their context. Take the following simple assignment of the numerical value 128, which probably indicates the use of the Euro currency symbol (€):

```
var
  str1: string;
begin
  str1 := #$80;
```

This code is not Unicode compliant, as the code point for that symbol is 8364. The value, in fact, doesn't come from the official ISO codepages but was a specific Microsoft implementation for Windows. To make it easier to move existing code to Unicode, the Object Pascal compiler can treat 2-digit string literals as ANSI charac-

ters (which might depend on your actual code page). Surprisingly enough if you take that value, convert it to a Char, and display it again... the numerical representation will change to the correct one. So by executing the statement:

```
Show (str1 + ' - ' + IntToStr (Ord (str1[1])));
```

I'll get the output:

```
€ - 8364
```

Given you might prefer fully migrating your code and getting rid of older ANSI-based literal values, you can change the compiler behavior by using the special directive `$HIGHCHARUNICODE`. This directive determines how literal values between `#$80` and `#$FF` are treated by the compiler. What I discussed earlier is the effect of the default option (OFF). If you turn it on, the same program will produce this output:

```
 - 128
```

The number is interpreted as an actual Unicode code point and the output will contain a non-printable control character. Another option to express that specific code point (or any Unicode code point below `#$FFFF`) is to use the four-digits notation:

```
str1 := #$0080;
```

This is not interpreted as the Euro currency symbol regardless of the setting of the `$HIGHCHARUNICODE` directive.

What is nice is that you can use the four digits notation to express far eastern characters, like the following two Japanese characters:

```
str1 := #$3042#$3044;
Show (str1 + ' - ' + IntToStr (Ord (str1.Chars[0])) +
  ' - ' + IntToStr (Ord (str1.Chars[1])));
```

displayed as (along with their Integer representation):

```
あい - 12354 – 12356
```

---

note あい translates to “meeting” according to BabelFish, but I'm not 100% sure where I originally found it, and given I don't know Japanese this might as well be wrong.

You can also use literal elements over `#$FFFF` that will be converted to the proper surrogate pair.

The String Data Type

The string data type in Object Pascal is way more sophisticated than a simple array of characters, and has features that go well beyond what most programming languages do with similar data types. In this section I'll introduce the key concepts behind this data type, and in coming sections we'll explore some of these features in more details.

In the following bullet list I've captured the key concepts for understanding how strings work in the language (remember, you can use string without knowing much of this, as the internal behavior is very transparent):

- Data for the string type is ***dynamically allocated*** on the heap. A string variable is just a reference to the actual data. Not that you have to worry much about this, as the compiler handles this transparently. Like for a dynamic array, as you declare a new string, this is empty.
- While you can assign data to a string in many ways, you can also ***allocate a specific memory area*** calling the `SetLength` function. The parameter is the number of characters (of 2 bytes each), the string should be able to have. When you extend a string, the existing data is preserved (but it might be moved to a new physical memory location). When you reduce the size, some of the content will likely be lost. Setting the length of a string is seldom necessary. The only common case is when you need to pass a string buffer to an operating system function for the given platform.
- If you want to ***increase the size*** of a string in memory (by concatenating it with another string) but there is something else in the adjacent memory, then the string cannot grow in the same memory location, and a full copy of the string must therefore be made in another location.
- To clear a string you don't operate on the reference itself, but can simply set it to an empty string, that is `''`. Or you can use the `Empty` constant, which corresponds to that value.
- According to the rules of Object Pascal, the ***length of a string*** (which you can obtain by calling `Length`) is the number of valid elements, not the number of allocated elements. Differently from C, which has the concept of a string terminator (`#0`), all versions of Pascal since the early days tend to favor the use of a specific memory area (part of the string) where the actual length information is stored. At times, however, you'll find strings that also have the terminator.
- Object Pascal strings use a ***reference-counting*** mechanism, which keeps track of how many string variables are referring to a given string in memory. Reference

counting will free the memory when a string isn't used anymore—that is, when there are no more string variables referring to the data... and the reference count reaches zero.

- Strings use a ***copy-on-write*** technique, which is highly efficient. When you assign a string to another or pass one to a string parameter, no data is copied and the reference count in increased. However, if you do change the content of one of the references, the system will first make a copy and than affect only that copy, with the other references remaining unchanged.
- The use of ***string concatenation*** for adding content to an existing string is generally very fast and has no significant drawback. While there are alternative approaches, concatenating strings is fast and powerful. This is not true for many programming languages these days.

Now I can guess this description can be a little confusing, so let's look at the use of strings in practice. In a while I'll get to a demo showcasing some of the operations above, including reference counting and copy-on-write. Before we do so, however, let me get back to the string helper operations and some other fundamental RTL functions for strings management.

Before we proceed further, let me examine some of the elements of the previous list in terms of actual code. Given string operations are quite seamless it is difficult to fully grasp what happens, unless you start looking inside the strings memory structure, which I'll do later in this chapter, as it would be too advanced for now. So let's start with some simple string operations, extracted from the `Strings101` application project:

```
var
  String1, String2: string;
begin
  String1 := 'hello world';
  String2 := String1;
  Show ('1: ' + String1);
  Show ('2: ' + String2);
  String2 := String2 + ', again';
  Show ('1: ' + String1);
  Show ('2: ' + String2);
end;
```

This first snippet, when executed, shows that if you assign two strings to the same content, modifying one won't affect the other. That is, `String1` is not affected by the changes to `String2`:

```
1: hello world
2: hello world
1: hello world
2: hello world, again
```

Still, as we'll figure out better in a later demo, the initial assignment doesn't cause a full copy of the string, the copy is delayed (again, a feature called *copy-on-write*).

Another important feature to understand is how the length is managed. If you ask for the length of a string, you get the actual value (which is stored in the string metadata, making the operation very fast). But if you call SetLength, you are allocating memory, which most often will be not initialized. This is generally used when passing the string as a buffer to an external system function. If you need a blank string, instead, you can use the pseudo-constructor (`Create`). Finally, you can use SetLength to trim a string. All of these are demonstrated by the following code:

```
var
  string1: string;
begin
  string1 := 'hello world';
  Show(string1);
  Show ('Length: ' + string1.Length.ToString);

  SetLength (string1, 100);
  Show(string1);
  Show ('Length: ' + string1.Length.ToString);

  string1 := 'hello world';
  Show(string1);
  Show ('Length: ' + string1.Length.ToString);

  string1 := string1 + string.Create(' ', 100);
  SetLength (string1, 100);
  Show(string1);
  Show ('Length: ' + string1.Length.ToString);
```

The output is more or less the following:

```
hello world
Length: 11
hello world
Length: 100
hello world
Length: 11
hello world
Length: 100
```

The third concept I want to underline in this section is that of an empty string. A string is empty when its content is an empty string. For both assignment and testing you can use two consecutive quotes, or specific functions:

```
var
  string1: string;
begin
  string1 := 'hello world';
  if string1 = '' then
```

```
    Show('Empty')
  else
    Show('Not empty');

  string1 := ''; // or string1.Empty;
  if string1.IsEmpty then
    Show('Empty')
  else
    Show('Not empty');
```

With this simple output:

```
Not empty
Empty
```

Passing Strings as Parameters

As I've explained, if you assign a string to another, you are just copying a reference, while the actual string in memory is not duplicated. However, if you write code that changes that string (and only at that point) the string is first copied, and than modified.

Something very similar happens when you pass a string as parameter to a function or procedure. By default, you get a new reference and if you modify the string in the function, the change doesn't affect the original string. If you want a different behavior, that is the ability to modify the original string in the function, you need to pass the string by reference, using the `var` keyword (as it happens for most other simple and managed data types).

But what if you don't modify the string passed as parameter? In that case, you can apply an actual optimization by using the `const` modifier for the parameter. In this case the compiler won't let you change the string in the function or procedure, but it will also optimize the parameter passing operation. In fact, a const string doesn't require the function to increase the string reference count when it starts and decrease it when it ends. While these operations are very fast, executing those thousands or millions of times will add a slight overhead to your programs. This is why passing string as `const` is recommended in cases where the function doesn't have to modify the value of the string parameter.

In coding terms, these are the declarations of three procedures with a string parameters passed in different ways:

```
procedure ShowMsg1 (str: string);
procedure ShowMsg2 (var str: string);
procedure ShowMsg3 (const str: string);
```

The Use of [] and String Characters Counting Modes

As you are likely to know if you have used Object Pascal or any other programming language, a key string operation is accessing one of the elements of a string, something often achieved using the square brackets notation (`[]`), in the same way you access to the elements of an array.

In Object Pascal there are two slightly different ways to perform these operations:

- The `Chars[]` string type helper operation (the entire list is in the next section) is a read only character access that uses a 0-based index.
- The standard `[]` string operator supports both reading and writing, and can use either a zero-based or a one-based index depending on a compiler setting.

This is a bit confusing, at first, and does require some clarification, which I'm going to provide after a short historical note. The reason for this note, which you can skip if not interested, is that it would be difficult to understand why the language behaves in the current way without looking at what happened over time.

---

note Let me look back in time for a second, to explain you how we got here today. In the early days of the Pascal language, strings were treated like an array of characters in which the first element (that is the 0th element of the array) was used to store the number of valid characters in the string, or the string length. In those days, while the C language had to recompute the length of a string each time, looking for a terminator, Pascal code could just make a direct check to that byte. Given that the byte number 0 was used for the length, it happened that the first actual character stored in the string was at position 1. Over time, almost all other languages had zero-based strings and arrays. Later, Object Pascal adopted 0-based dynamic arrays and most of the RTL and component libraries used 0-based data structures, with strings being a significant exception. While moving to the mobile world, the Object Pascal language designers decided to give “priority” to zero-based strings, allowing developers to still use the older model in case they had existing Object Pascal source code to move over. Needless to say this generated a lot of debate in the developer community.

If we want to draw a comparison to better explain the differences in the base of the index, consider how floors are counted in Europe and in North America (I honestly don't know about the rest of the world). In Europe the ground floor is floor 0, and the first floor is the one above it (at times formally indicated as “floor one above ground”). In North America, the first floor is the ground floor, and the second first is the first above ground level. In other words, America uses a 1-based floor index, while Europe uses a 0-based floor index. For strings, instead, the largest majority of programming languages uses the 0-based notation, regardless of the continent they were invented.

Let me explain the situation with string indexes a little better. As I mentioned above, the `Char[]` invariably uses a zero-based index. So if you write

```
var
  string1: string;
begin
  string1 := 'hello world';
  Show (string1.Chars[1]);
```

the output will be:

```
e
```

What if you use the direct [] notation, that is what will be the output of:

```
  Show (string1[1]);
```

This might be either `h` or `e`, depending on a compiler define, `$ZEROBASEDSTRING`. If this is `ON` the output will be `e`, if it is `OFF` the output will be `h`. Now given this setting was introduced for backwards compatibility, the mobile compiler has it set to `ON` by default, while the Windows compiler has it set to `OFF`. How can we handle this discrepancy? There are a few alternative options:

- For new applications, my suggestion is to enable zero-based strings setting for all of your code (at the project options level) including the desktop code, and follow the standard approach used by most programming languages.
- If you have existing code, tested and verified, currently working with the classic Pascal 1-based notation, bring this over the mobile by disabling the compiler flag for all platforms.
- Make sure your code works in both cases, by abstracting the index, for example using Low(string) as the index of the first value. This works returning the proper value depending on the local compiler setting for the string base. However, while this makes sense for libraries you want to keep readable and usable regardless of the compiler setting (like the RTL libraries), it requires some extra work which is not really required at the application level.

While implementing the first and second strategy is relatively simple, implementing code that works regardless of this setting takes a little effort. I did this in the section covering the `for` loop, for example, when I wrote:

```
var
  s: string;
  I: Integer;
begin
  s := 'Hello world';
  for I := Low (s) to High (s) do
    Show(s[I]);
```

In other words, a string *invariably* has elements ranging from the result of the `Low` function to that of the `High` function applied to the same string. If you want to know, in general, which is the active compiler setting, you can use:

```
Low (string)
```

This returns either 0 or 1 depending on the active option. In any case, always keep in mind that this is just a local compiler setting that determines how the index within square brackets is interpreted.

note A string is just a string, and the concept of a zero-based string I completely wrong. The data structure in memory is not different in any way, so you can pass any string to any function that uses a notation with any base value, and there is no problem at all. In other words, if you have a code fragment accessing to strings with a zero-based notation you can pass the string to a function that is compiled using the settings for a one-based notation.

Concatenating Strings

I have already mentioned that unlike other languages, Object Pascal has full support for direct string concatenation, which is actually a rather fast operation. In this chapter I'll just show you some string concatenations code, while doing some speed testing. Later on, in Chapter 18, I'll briefly document the `TStringBuilder` class, which follows the .NET notation for assembling a string out of different fragments. While there are reasons to use `TStringBuilder`, performance is not the most relevant one (as the following example will show).

So, how do we concatenate strings in Object Pascal? Simply by using the + operator:

```
var
  str1, str2: string;
begin
  str1 := 'Hello,';
  str2 := ' world';
  str1 := str1 + str2;
```

Notice how I used the `str1` variable both on the left and on the right of the assignment, adding some more content to an existing string rather than assigning to a brand new one. Both operations are possible, but adding content to an existing string is where you can get some nice performance.

This type of concatenation can be also done in a loop, like the following extracted from the `LargeString` application project:

```
uses
  Diagnostics;

const
```

```
  MaxLoop = 2000000; // two million

var
  str1, str2: string;
  I: Integer;
  t1: TStopwatch;
begin
  str1 := 'Marco ';
  str2 := 'Cantu ';

  t1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    str1 := str1 + str2;

  t1.Stop;
  Show('Length: ' + str1.Length.ToString);
  Show('Concatenation: ' + t1.ElapsedMilliseconds.ToString);
end;
```

By running this code, I get the following timing on a Windows virtual machines and on an Android device (the computer is quite a bit faster):

```
// Windows (in a VM)
Length: 12000006
Concatenation: 59

// Android (Nexus 4)
Length: 12000006
Concatenation: 991
```

The application project has also similar code based on the `TStringBuilder` class. While I don't want to get to the details of that code (again, I'll describe the class in Chapter 18) I want to share the actual timing, for comparison with the plain concatenation timing just displayed

```
// Windows (in a VM)
Length: 12000006
StringBuilder: 79

// Android (Nexus 4)
Length: 12000006
StringBuilder: 1057
```

As you can see, concatenation can be safely considered the fastest option.

The String Helper Operations

Given the importance of the string type, it should come to no surprise that the helper for this type has a rather long list of operations you can perform. And given

its importance and the commonality of these operations in most applications, I think it is worth going through this list with some care.

I've logically grouped the string helper operations (most of which have many overloaded versions), shortly describing what they do, considering that quite often the names are rather intuitive:

- Copy or partial copy operations like `Copy`, `CopyTo`, `Join`, and `SubString`
- String modification operations like `Insert`, `Remove`, and `Replace`
- For conversion from various data types to string, you can use `Parse` and `Format`
- Conversion to various data types, when possible can be achieved using `ToBoolean`, `ToInteger`, `ToSingle`, `ToDouble`, and `ToExtended` while you can turn a string into an array of characters with `ToCharArray`
- Fill a string white space or specific characters with `PadLeft`, `PadRight`, and one of the overloaded versions of `Create`. At the opposite, you can remove white space at one end of the string or both using `TrimRight`, `TrimLeft`, and `Trim`
- String comparison and equality test (`Compare`, `CompareOrdinal`, `CompareText`, `CompareTo`, and `Equals`)—but keep in mind you can also, to some extent, use the equality operator and the comparison operators
- Changing case with `LowerCase` and `UpperCase`, `ToLower` and `ToUpper`, and `ToUpperInvariant`
- Test the string content with operations like `Contains`, `StartsWith`, `EndsWith`. Search in the string can be done using `IndexOf` for finding the position of a given character (from the beginning or from a given location), the similar `IndexOfAny` (which looks for one of the elements of an array of characters), the `LastIndexOf` and `LastIndexOfAny` operations which work backwards from the end of the string, and the special purpose operations `IsDelimiter` and `LastDelimiter`
- Access to general information about the string with functions like `Length`, which returns the number of characters, `CountChars`, which also takes surrogate pairs into account, `GetHashCode`, which return a hash of the string, and the various tests for “emptiness” which include `IsEmpty`, `IsNullOrEmpty`, and `IsNullOrWhiteSpace`
- String special operations like `Split`, which breaks a string into multiple ones based on a specific character, and removing or adding quotes around the string with `QuotedString` and `DeQuoted`
- And, finally, access to individual characters with `Char[]`, which has the numerical index of the element of the string among square brackets. This can be used only for reading a value (not for changing it) and uses a zero-based index like all other string helper operations.

It is important to notice, in fact, that all of the string helper methods have been build following the string RTL used by other languages, which includes the concept that string elements start with zero and go up to the length of the string minus one. In other words, all string helper operations use zero-based indexes as parameters and return values.

note The `Split` operation is relatively new to the Object Pascal RTL. A previously common approach was loading a string in a string list, after setting a specific line separator, and later access the individual strings, or lines. The `Split` operation is significantly more efficient and flexible.

Given the large amount of operations you can apply directly to strings, I could have created several projects demonstrating these capabilities. Instead, I'll stick to a few relatively simple operations, albeit very common ones.

The `StringHelperTest` application project has two buttons. In each of them the first part of the code builds and displays a string:

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
  Str1 := '';

  // create string
  for I := 1 to 10 do
    Str1 := Str1 + 'Object ';

  Str2:= string.Copy (Str1);
  Str1 := Str2 + 'Pascal ' + Str2.Substring (10, 30);
  Show(Str1);
```

Noticed how I used the `Copy` function, to create a unique copy of the data of the string, rather than an alias... even if in this particular demo it won't have made any difference. The `Substring` call at the end is used to extract a portion of the string. The resulting text is:

```
Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Objec
```

After this initialization, the first button has code for searching for a substring and for repeating such a search, with a different initial index, to count the occurrences of a given string (in the example a single character):

```
  // find substring
  Show('Pascal at: ' +
    Str1.IndexOf ('Pascal').ToString);

  // count occurrences
  I := -1;
  NCount := 0;
```

```
repeat
  I := Str1.IndexOf('O', I + 1); // search from next element
  if I >= 0 then
    Inc (NCount); // found one
until I < 0;

Show('O found: ' +
  NCount.ToString + ' times');
```

I know the repeat loop is not the simplest one: it starts with a negative index, as any following search begins with the index after the current one; it counts occurrences; and its termination is based on the fact that if the element is not found it returns `-1`. The output of the code is:

```
Pascal at: 70
O found: 14 times
```

The second button has code to perform a search and replace one or more elements of a string with something else. In the first part, it creates a new string copying the initial and final part and adding some new text in the middle. In the second, it uses the Replace function that can operate on multiple occurrences simply by passing to it the proper flag (`rfReplaceAll`). This is the code:

```
// single replace
nIndex := str1.IndexOf ('Pascal');
str1 := str1.Substring(0, nIndex) + 'Object' +
  str1.Substring(nIndex + ('Pascal').Length);
Show (str1);

// multi-replace
str1 := str1.Replace('O', 'o', [rfReplaceAll]);
Show (str1);
```

As the output is rather long and not easy to read, here I've listed only the central portion of each string:

```
...Object Pascal ect Object Object...
...Object Object ect Object Object...
...object object ect object object...
```

Again, this is just a minimal sampler of the rich string operations you can perform using the operations available for the string type using the string type helper.

More String RTL

An effect of the decision to implement the string helper following the names of operations common in other programming languages is the fact that the names of the type operations often diverge from the traditional Object Pascal ones (which are still

available as global functions today. The following table has some of the *not-matching* functions names:

global	*string type helper*
Pos	IndexOf
IntToStr	Parse
StrToInt	ToInteger
CharsOf	Create
StringReplace	Replace

note Remember that there is a big difference between the global and the Char helper operations: The first group uses a one-based notation for indexing elements within a string, while the latter group uses a zero-based notation (as explained earlier).

These are only the most commonly used functions of the string RTL that have changed name, while many others still use the same like `UpperCase` or `QuotedString`. The `System.SysUtils` unit has a lot more, and the specific `System.StrUtils` unit has also many functions focused on string manipulation that are not part of the string helper.

Some notable functions part of the `System.StrUtils` unit are:

- `ResemblesText`, which implements a *Soundex* algorithm (finding words with similar sound even if a different spelling);
- `DupeString`, which returns the requested number of copies of the given string;
- `IfThen`, which returns the first string passed if a condition is true, else it will return the second string (I used this function in a code snippet earlier in this chapter);
- `ReserveString`, which returns a string with the opposite characters sequence.

Formatting Strings

While concatenating string with the plus (+) operator and using some of the conversion functions you can indeed build complex strings out of existing values of various data types, there is a different and more powerful approach to formatting numbers, currency values, and other strings into a final string. Complex string formatting can be achieved by calling the `Format` function, a very traditional but still extremely common mechanism, not only in Object Pascal but in most programming languages.

note The family of "print format string" or `printf` functions date back to the early days of programming and languages like FORTRAN 66, COBOL, and ALGOL 68. The specific format string structure still in use today (and used by Object Pascal) is close to the C language `printf` function. For a historical overview you can refer to en.wikipedia.org/wiki/Printf_format_string.

The `Format` function requires as parameters a string with the basic text and some placeholders (marked by the % symbol) and an array of values, generally one for each of the placeholders. For example, to format two numbers into a string you can write:

```
Format ('First %d, Second %d', [n1, n2]);
```

where `n1` and `n2` are two Integer values. The first placeholder is replaced by the first value, the second matches the second, and so on. If the output type of the placeholder (indicated by the letter after the `%` symbol) doesn't match the type of the corresponding parameter, a runtime error occurs. Having no compile-time type checking is actually the biggest drawback of using the `Format` function. Similarly, not passing enough parameters causes a runtime error.

The `Format` function uses an open-array parameter (a parameter that can have an arbitrary number of values or arbitrary data types, as covered in Chapter 5). Besides using `%d`, you can use one of many other placeholders defined by this function and briefly listed the following table. These placeholders provide a default output for the given data type. However, you can use further format specifiers to alter the default output. A width specifier, for example, determines a fixed number of characters in the output, while a precision specifier indicates the number of decimal digits. For example,

```
Format ('%8d', [n1]);
```

converts the number `n1` into an eight-character string, right-aligning the text (use the minus (-) symbol to specify left-justification) filling it with white spaces. Here is the list of formatting placeholders for the various data types:

`d` (decimal) The corresponding integer value is converted to a string of decimal digits.

`x` (hexadecimal) The corresponding integer value is converted to a string of hexadecimal digits.

`p` (pointer) The corresponding pointer value is converted to a string expressed with hexadecimal digits.

`s` (string) The corresponding string, character, or `PChar` (pointer to a character array) value is copied to the output string.

`e` (exponential) The corresponding floating-point value is converted to a string based on scientific notation.

`f` (floating point)	The corresponding floating-point value is converted to a string based on floating point notation.
`g` (general)	The corresponding floating-point value is converted to the shortest possible decimal string using either floating-point or exponential notation.
`n` (number)	The corresponding floating-point value is converted to a floating-point string but also uses thousands separators.
`m` (money)	The corresponding floating-point value is converted to a string representing a currency amount. The conversion is generally based on regional settings.

The best way to see examples of these conversions is to experiment with format strings yourself. To make this easier I've written the `FormatString` application project, which allows a user to provide formatting strings for a few predefined integer values.

The form of the program has an edit box above the buttons, initially holding a simple predefined formatting string acting as a placeholder (`'%d - %d - %d'`). The first button of the application lets you display a more complex sample format string in the edit box (the code has a simple assignment to the edit text of the format string `'Value %d, Align %4d, Fill %4.4'`). The second button lets you apply the format string to the predefined values, using the following code:

```
var
  strFmt: string;
  n1, n2, n3: Integer;
begin
  strFmt := Edit1.Text;
  n1 := 8;
  n2 := 16;
  n3 := 256;

  Show (Format ('Format string: %s', [strFmt]));
  Show (Format ('Input data: [%d, %d, %d]', [n1, n2, n3]));
  Show (Format ('Output: %s', [Format (strFmt, [n1, n2, n3])]));
  Show (''); // blank line
end;
```

If you display the output first with the initial format string and next with the sample format string (that is if you press the second button, the first, and than the second again), you should get an output like the following:

```
Format string: %d - %d - %d
Input data: [8, 16, 256]
Output: 8 - 16 - 256

Format string: Value %d, Align %4d, Fill %4.4d
Input data: [8, 16, 256]
Output: Value 8, Align   16, Fill 0256
```

However the idea behind the program is to edit the format string and experiment with it, to see all of the various available formatting options.

The Internal Structure of Strings

While you can generally use strings without knowing much about their internals, it is interesting to have a look to the actual data structure behind this data type. In the early days of the Pascal language, strings had a maximum of 255 elements of one byte each and would use the first byte (or zero byte) for storing the string length. A lot of time has passed since those early days, but the concept of having some extra information about the string stored as part of its data remains a specific approach of the Object Pascal language (unlike many languages that derive from C and use the concept of a string terminator).

note ShortString is the name of the traditional Pascal string type, a string of one byte characters or `AnsiChar` limited to 255 characters. The ShortString type is still available in the desktop compilers, but not in the mobile ones. You can represent a similar data structure with a dynamic array of bytes, or `TBytes`, or a plain static arrays of `Byte` elements.

As I already mentioned, a string variable is nothing but a pointer to a data structure allocated on the heap. Actually, the value stored in the string is not a reference to the beginning of the data structure, but a reference to the first of the characters of the string, with string metadata data available at negative offsets from that location. The in-memory representation of the data of the string type is the following:

-12	-10	-8	-4	String reference address
Code page	Elem size	Ref count	Length	First char of string

The first element (counting backwards from the beginning of the string itself) is an Integer with the string length, the second element holds the reference count. Further fields (used on desktop compilers) are the element size in bytes (either 1 or 2 bytes) and the code page for older Ansi-based string types (available on the desktop compilers).

Quite surprisingly, it is possible to access to most of these fields with specific low-level string metadata functions, beside the rather obvious `Length` function:

```
function StringElementSize(const S: string): Word;
function StringCodePage(const S: string): Word;
function StringRefCount(const S: string): Longint;
```

As an example, you can create a string and ask for some information about it, as I did in the `StringMetaTest` example:

```
var
  str1: string;
begin
  str1 := 'F' + string.Create ('o', 2);

  Show ('SizeOf: ' + SizeOf (str1).ToString);
  Show ('Length: ' + str1.Length.ToString);
  Show ('StringElementSize: ' +
    StringElementSize (str1).ToString);
  Show ('StringRefCount: ' +
    StringRefCount (str1).ToString);
  Show ('StringCodePage: ' +
    StringCodePage (str1).ToString);
  if StringCodePage (str1) = DefaultUnicodeCodePage then
    Show ('Is Unicode');
  Show ('Size in bytes: ' +
    (Length (str1) * StringElementSize (str1)).ToString);
  Show ('ByteLength: ' +
    ByteLength (str1).ToString);
```

note There is a specific reason the program builds the *'Foo'* string dynamically rather than assigning a constant, and that is because constant string have the reference count disabled (or set to -1). In the demo I preferred showing a proper value for the reference count, hence the dynamic string construction.

This program produces output similar to the following when running on Windows:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

The following is the output if you run the same program on Android:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

The code page returned by a UnicodeString is 1200, a number stored in the global variable `DefaultUnicodeCodePage`. In the code above (and its output) you can clearly notice the difference between the size of a string variable (invariably 4), the logical length, and the physical length in bytes. This can be obtained by multiplying the size in bytes of each character times the number of characters, or by calling Byte-

Length. This latter function, however, doesn't support some of the string types of the older desktop compiler.

Looking at Strings in Memory

The ability to look into a string's metadata can be used to better understand how string memory management works, particularly in relationship with the reference counting. For this purpose, I've added some more code to the `StringMetaTest` application project.

The program has two global strings: `MyStr1` and `MyStr2`. The program assigns a dynamic string to the first of the two variables (for the reason explained earlier in the note) and then assigns the second variable to the first:

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);
MyStr2 := MyStr1;
```

Besides working on the strings, the program shows their internal status, using the following `StringStatus` function:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Addr: ' +
    IntToStr (Integer (Str)) +
    ', Len: ' +
    IntToStr (Length (Str)) +
    ', Ref: ' +
    IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Val: ' + Str;
end;
```

It is important in the `StringStatus` function to pass the string parameter as a `const` parameter. Passing this parameter by copy will cause the side effect of having one extra reference to the string while the function is being executed. By contrast, passing the parameter via a reference (`var`) or constant (`const`) doesn't imply a further reference to the string. In this case I've used a `const` parameter, as the function is not supposed to modify the string.

To obtain the memory address of the string (useful to determine its actual identity and to see when two different strings refer to the same memory area), I've simply made a hard-coded typecast from the string type to the Integer type. Strings are references-in practice, they're pointers: Their value holds the actual memory location of the string not the string itself.

The code used for testing what happens to the string is the following:

```
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
```

```
MyStr1 [1] := 'a';
Show ('Change 2nd char');
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
```

Initially, you should get two strings with the same content, the same memory location, and a reference count of 2.

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```

As the application changes the value of one of the two strings (it doesn't matter which one), the memory location of the updated string will change. This is the effect of the copy-on-write technique. This is the second part of the output:

```
Change 2nd char
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

You can freely extend this example and use the *StringStatus* function to explore the behavior of long strings in many other circumstances, with multiple reference, when they are passed as parameters, assigned to local variables, and more.

Strings and Encodings

As we have seen the string type in Object Pascal is mapped to the Unicode UTF16 format, with 2-bytes per element and management of surrogate pairs for code points outside of the BMP (Basic Multi-language Plane). There are many cases, though, in which you need to save to file, load from file, transmit over a socket connection, or receive form a connection textual data that uses a different representation, like ANSI or UTF8.

To convert files and in memory data among different formats (or encodings), the Object Pascal RTL has a handy `TEncoding` class, defined in the `System.SysUtils` unit along with several inherited classes.

---

note There are several other handy classes in the Object Pascal RTL that you can use for reading and writing data in text formats. For example, the `TStreamReader` and `TStreamWriter` classes offer support for text files with any encoding. These classes will be introduced in Chapter 18.

Although I still haven't introduced classes and inheritance, this set of encoding classes is very easy to use, as there is already a global object for each encoding, automatically created for you. In other words, an object of each of these encoding classes is available within the `TEncoding` class, as a class property:

```
type
  TEncoding = class
    ...
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
      read GetBigEndianUnicode;
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

note The `Unicode` encoding is based on the `TUnicodeEncoding` class that uses the same UTF-16 LE (Little Endian) format used by the string type. The BigEndianUnicode, instead, uses the less common Big Endian representation. If you are not familiar with "Endianness" this is a terms used to indicate the sequence of two bytes making a code point (or any other data structure). Little Endian has the most significant byte first, and Big Endian has the most significant byte last. For more information, see en.wikipedia.org/wiki/Endianness.

Again, rather than exploring these classes in general, something a little difficult at this point of the book, let's focus on a couple of practical examples. The `TEncoding` class has methods for reading and writing Unicode strings to byte arrays, performing appropriate conversions.

To demonstrate UTF format conversions via `TEncoding` classes, but also keep my example simple and focused and avoid working with the file system, in the `EncodingsTest` application project I've created an UTF-8 string in memory using some specific data, and converted it to UTF-16 with a single function call:

```
var
  Utf8string: TBytes;
  Utf16string: string;
begin
  // process Utf8data
  SetLength (Utf8string, 3);
  Utf8string[0] := Ord ('a'); // single byte ANSI char < 128
  Utf8string[1] := $c9; // double byte, reversed latin a
  Utf8string[2] := $90;

  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Unicode: ' + Utf16string);
```

The output should be:

```
Unicode: aɐ
```

Now to better understand the conversion and the difference in the representations, I've added the following code:

```
  Show ('Utf8 bytes:');
  for AByte in Utf8String do
    Show (AByte.ToString);
```

```
  Show ('Utf16 bytes:');
  UniBytes := TEncoding.Unicode.GetBytes (Utf16string);
  for AByte in UniBytes do
    Show (AByte.ToString);
```

This code produces a memory dump, with decimal values, for the two representations of the string, UTF8 (a one byte and a two byte code point) and UTF16 (with both code points being 2 bytes):

```
Utf8 bytes:
97
201
144
Utf16 bytes:
97
0
80
2
```

Notice that direct character to byte conversion, for UTF-8, work only for ANSI-7 characters, that is values up to 127. For higher level ANSI characters there is no direct mapping and you must perform a conversion, using the specific encoding (which will however fail on multi-byte UTF-8 elements). So both of the following produce wrong output:

```
  // error: cannot use char > 128
  Utf8string[0] := Ord ('à');
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Wrong high ANSI: ' + Utf16string);
  // try different conversion
  Utf16string := TEncoding.ANSI.GetString(Utf8string);
  Show ('Wrong double byte: ' + Utf16string);

  // output
  Wrong high ANSI:
  Wrong double byte: àÉ
```

The encoding classes let you convert in both directions, so in this case I'm converting from Unicode to UTF-8, doing some processing of the UTF-8 string (something to be done with care, given the variable length nature of this format), and convert back to UTF-16:

```
var
  Utf8string: TBytes;
  Utf16string: string;
  I: Integer;
begin
  Utf16string := 'This is my nice string with à and Æ';
  Show ('Initial: ' + Utf16string);

  Utf8string := TEncoding.UTF8.GetBytes(Utf16string);
  for I := 0 to High(Utf8string) do
```

```
    if Utf8string[I] = Ord('i') then
      Utf8string[I] := Ord('I');
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Final: ' + Utf16string);
```

The output is:

```
Initial: This is my nice string with à and Æ
Final: ThIs Is my nIce strIng wIth à and Æ
```

Other Types for Strings

While the string data type is by far the most common and largely used type for representing strings, Object Pascal desktop compilers had and still have a variety of string types. Some of these types can be used also on mobile applications, but the general recommendation is to do the appropriate conversion or just use `TBytes` directly to manipulate string with a 1-byte representation, as in the application project described in the last section.

While developers who used Object Pascal in the past might have a lot of code based on these pre-Unicode types (or directly managing UTF-8), modern applications really require full Unicode support. Also while some types, like UTF8String, are available in the desktop compilers, their support in terms of RTL is severely limited. Again, you can use an array of bytes to represent a similar type and adapt existing code to handle it, but the recommendation is to move to plain and standard Unicode strings.

---

note While there has been a lot of discussion and criticism about the lack of native types like AnsiString and UTF8String in the Object Pascal mobile compilers, honestly there is almost no other programming language out there that has more than one native or intrinsic string type. Multiple string types are more complex to master, can cause unwanted side effects (like extensive automatic conversion calls that slow down programs), and cost a lot for the maintenance of multiple version of all of the string management and processing functions.

The UCS4String type

An interesting but little used string type is the UCS4String type, available on all compilers. This is just an UTF32 representation of a string, and no more than an array of UTF32Char elements, or 4-bytes characters. The reason behind this type, as

mentioned earlier, is that is offers a direct representation of all of the Unicode code points. The obvious drawback is such a string takes twice as much memory than a UTF-16 string (which already takes twice as much than an ANSI string)..

Although this data type can be used in specific situations, it is not particularly suited for general circumstances. Also, this types doesn't support copy-on-write nor has any real system functions and procedures for processing it.

---

note Whilst the UCS4String guarantees one UTF32Char per Unicode code point, it cannot guarantee one UTF32Char per grapheme, or "visual character".

Older, Desktop Only String Types

As mentioned, the desktop versions of the Object Pascal compilers offer support for some older, traditional string types. These include

- The `ShortString` type, which corresponds to the original Pascal language string type. These strings have a limit of 255 characters. Each element of a short string is of type `ANSIChar` (a type also available only in desktop compilers).
- The `ANSIString` type, which corresponds to variable-length strings. These strings are allocated dynamically, reference counted, and use a copy-on-write technique. The size of these strings is *almost* unlimited (they can store up to two billion characters!). Also this string type is based on the `ANSIChar` type.
- The `WideString` type is similar to a 2-bytes Unicode string in terms of representation, is based on the `Char` type, but unlike the standard string type is doesn't use copy-on-write and it is less efficient in terms of memory allocation. If you wonder why it was added to the language, the reason was for compatibility with string management in Microsoft's COM architecture.
- `UTF8String` is a string based on the variable character length UTF-8 format. As I mentioned there is little run-time library support for this type.
- `RawByteString` is an array of characters with no code page set, on which no character conversion is ever accomplished by the system (thus logically resembling a `TBytes` structure, but allowing some direct string operations an array of bytes currently lacks).
- A string construction mechanism allowing you to define a 1-byte string associated with a specific ISO code page, a remnant of the pre-Unicode past.

Again, all of these string types can be used on desktop compilers, but are available only for backwards compatibility reason. The goal is to use Unicode, `TEncoding`, and other modern string management techniques whenever possible.

part ii: oop in object pascal

Many modern programming languages support some form of *object-oriented programming* (OOP) paradigm. Many of them use a class-based one that is based on three fundamental concepts:

- Classes, data types with a public interface and a private data structure, implementing encapsulation; instances of these data types are generally called objects,
- Class extensibility or inheritance, which is the ability to extend a data type with new features without modifying the original one,
- Polymorphism or late binding, which is the ability to refer to objects of different classes with a uniform interface, and still operate on objects in the way defined by their specific type.

note Other languages such as IO, JavaScript, Lua and Rebol use a prototype based object-oriented paradigm, which has a concept similar to a class, no inheritance, and dynamic typing that can be used to implement polymorphism, even if in a rather different way.

You can write Object Pascal applications even without knowing a lot about object oriented programming. As you create a new form, add new components, and handle events, the IDE prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you under-

stand precisely what the system is doing and allow you to master the language completely.

You also be able to create complex architectures within your applications, and even entire libraries, and embrace and extend the components that come with the development environment.

The second part of the book is focused on core object-oriented programming (OOP) techniques. The aim of this part of the book is both to teach the fundamental concepts of OOP and to detail how Object Pascal implements them, comparing it with other similar OOP languages.

---

note A language-neutral overview of the concepts behind OOP is available in Appendix D.

Summary of Part II

Chapter 7: Objects

Chapter 8: Inheritance

Chapter 9: Handling Exceptions

Chapter 10: Properties and Events

Chapter 11: Interfaces

Chapter 12: Manipulating Classes

Chapter 13: Objects and Memory

07: objects

Even if you don't have a detailed knowledge of object-oriented programming (OOP), this chapter will introduce each of the key concepts. If you are already fluent in OOP, you can probably go through the material relatively quickly and focus on Object Pascal language specifics, in comparison to other languages you might already know.

The OOP support in Object Pascal has a lot of similarities to languages like C# and Java, it also has some resemblances with C++ and other static and strongly-typed languages. Dynamic languages, instead, tend to offer a different interpretation of OOP, as they treat the type system in a more loose and flexible way.

note A lot of the conceptual similarities between C# and Object Pascal are due to the fact that the two languages share the same designer, Anders Hejlsberg. He was the original author of the Turbo Pascal compilers, of the first version of Delphi's Object Pascal, and later moved to Microsoft and designed C# (and more recently the JavaScript derivative TypeScript). You can read more about the Object Pascal language history in Appendix A.

Introducing Classes and Objects

Class and *object* are two terms commonly used in Object Pascal and other OOP languages. However, because they are often misused, let's be sure we agree on their definitions from the very beginning:

- A *class* is a user-defined data type, which has a state (or a representation) and defines some operations (or behaviors). In other terms, a class has some internal data and some methods, in the form of procedures or functions. A class usually describes the characteristics and behavior of a number of similar objects, although there are special purpose classes that are meant for a single object.
- An *object* is an instance of a class, that is a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation.

The relationship between an object and a class is the same as the one between any other variable and its data type. Only, in this case variables have a special name.

---

note The OOP terminology dates back to the first few languages that adopted the model, like Smalltalk. Most of the original terminology, however, was later dropped in favor of terms in use in proce - dural languages. So while terms like classes and objects are still commonly used, you'd generally hear the term invoking a method more often than the original term sending a message to a receiver (an object). A full and detailed guide to the OOP jargon and how it evolved over time could be interesting, but would take too much space in this book.

The Definition of a Class

In Object Pascal you can use the following syntax to define a new class data type (`TDate`), with some local data fields (`Month`, `Day`, `Year`) and some methods (`SetValue`, `LeapYear`):

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

---

note We have already seen a similar structure for records, which are quite similar to classes in term of definition. There are differences in memory management and other areas, as detailed later in this chapter. Historically, though, in Object Pascal this syntax was first adopted for classes and later ported back to records.

The convention in Object Pascal is to use the letter *T* as a prefix for the name of every class you write, like for any other type (*T* stands for *Type*, in fact). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier to understand by other programmers.

Unlike other languages, the class definition in Object Pascal doesn't include the actual implementation (or definition) of the methods, but only their signature (or declaration). This makes the class code more compact and significantly more readable.

note Although it might look that getting to the actual implementation of the method is more time consuming the editor allows you to use the combination of the Shift and Up and Down arrow keys to navigate from the method declarations to their implementations and vice verse. Moreover, you can let the editor generate a skeleton of the definition of the methods, after you write the class definition, by using Class Completion (pressing the Ctrl+C keys while the cursor is within the class definition).

Also keep in mind that beside writing the definition of a class (with its fields and methods) you can also write a declaration. This has only the class name, as in:

```
type
  TMyDate = class;
```

The reason for such a declaration lies in the fact that you might need to have two classes referencing each other. Given in Object Pascal you cannot use a symbol until it is defined, to refer a not-yet-defined class you need a declaration. I wrote the following code fragment only to show you the syntax, not that it makes any sense:

```
type
  THusband = class;

  TWife = class
    husband: THusband;
  end;

  THusband = class
    wife: TWife;
  end;
```

You'll encounter similar cross-references in real code, which is why this syntax is important to keep in mind. Notice, that like for methods, a class declared in a unit must be fully defined later in the same unit.

Classes in Other OOP Languages

As a comparison, this is the `TDate` class written in C# and in Java (which in this simplified case happen to be the same) using a more appropriate set of naming rules, with the code of the methods omitted:

```
// C# and Java language

class Date
{
  int   month;
  int   day;
  int   year;

  void setValue (int m, int d, int y)
  {
    // code
  }

  bool leapYear()
  {
    // code
  }
}
```

In Java and C# the methods' code comes within the class definition, while in Object Pascal the methods declared in a class should be fully defined in the implementation portion of the same unit that includes the class definition. In other words, in Object Pascal a class is always completely defined in a single unit (while a unit can, of course, contain multiple classes). By contrast, while in C++ methods are separately implemented like in Object Pascal, but a header file containing a class definition has no strict correspondence to an implementation file with the method's code. A corresponding C++ class would look like:

```
// C++ language

class Date
{
  int   month;
  int   day;
  int   year;

  void setValue (int m, int d, int y);
  BOOL leapYear();
}
```

The Class Methods

Like with records, when you define the code of a method you need to indicate which class it is part of (in this example the `TDate` class) by using the class name as a prefix and the dot notation, as in the following code:

```
procedure TDate.SetValue(m, d, y: Integer);
begin
  Month := m;
  Day := d;
  Year := y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas
  Result := IsLeapYear (Year);
end;
```

Differently from most other OOP languages that define methods as functions, Object Pascal brings over the core distinction between procedures and functions, depending on the presence of a return value, also for methods. This is not the case in C++, where a separately defined method implementation would look like:

```
// C++ method
void Date::setValue(int m, int d, int y)
{
  month = m;
  day = d;
  year = y;
};
```

Creating an Object

After this comparison with other popular languages, let's get back to Object Pascal to see how you can use a class. Once the class has been defined, we can create an object of this type and use it as in the following code snippet (extracted from the `Dates1` application project like all of the code in this section):

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;
  // use
  ADay.SetValue (1, 1, 2016);
  if ADay.LeapYear then
    Show ('Leap year: ' + IntToStr (ADay.Year));
```

The notation used is nothing unusual, but it is powerful. We can write a complex function (such as `LeapYear`) and then access its value for every `TDate` object as if it were a primitive data type. Notice that `ADay.LeapYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. As we'll see in Chapter 10, the notation used by Object Pascal to access properties is again the same.

note Calls of methods with no parameters in most programming languages based on the C language syntax require parenthesis, like in `ADay.LeapYear()`. This syntax is legal also in Object Pascal, but rarely used. Methods with no parameters are generally called without the parenthesis. This is very different from many languages in which a reference to a function or method with no parenthesis returns the function address. As we have see in the section "Procedural Types" in Chapter 4, Object Pascal uses the same notation for calling a function or reading its address, depending on the context of the expression.

The output of the code snippet above is fairly trivial:

```
Leap year: 2016
```

Again, let me compare the object creation with similar code written in other programming languages:

```
// C# and Java languages (object reference model)
Date aDay = new Date();

// C++ language (two alternative styles)
Date aDay; // local allocation
Date* aDay = new Date(); // "manual" reference
```

The Object Reference Model

In some OOP languages like C++, declaring a variable of a class type creates an instance of that class (more or less like it happens with records in Object Pascal). The memory for a local object is taken from the stack, and released when the function terminates. In most cases, though, you have to explicitly use pointers and references to have more flexibility in managing the lifetime of an object, adding a lot of extra complexity.

The Object Pascal language, instead, is based on an *object reference model*, exactly like Java or C#. The idea is that each variable of a class type does not hold the actual value of the object with its data (to store the day, month, and year, for example). Rather, it contains only a reference, or a *pointer,* to indicate the memory location where the actual object data is stored.

note In my opinion, adopting the object reference model was one of the best design decisions made by the compiler team in the early days of the language, when this model wasn't so common in programming languages (in fact, at the time Java wasn't available and C# didn't exist).

This is why in these languages you need to explicitly create an object and assign it to a variable, as objects are not automatically initialized. In other words, when you declare a variable, you don't create an object in memory, you only reserve the memory location for a reference to the object. Object instances must be created manually and explicitly, at least for the objects of the classes you define. (In Object Pascal, though, instances of components you place on a form are built automatically by the run time library.)

In Object Pascal, to create an instance of an object, we can call its special `Create` method, which is a constructor or another custom constructor defined by the class itself. Here is the code again:

```
ADay := TDate.Create;
```

As you can see, the constructor is applied to the class (the type), not to the object (the variable). That's because you are asking the class to create a new instance of its type, and the result is a new object you'd generally assign to a variable.

Where does the `Create` method come from? It is a constructor of the class `TObject`, from which all the other classes inherit, so it is universally available. It is very common to add custom constructors to your classes, though, as we'll see later in this chapter.

Disposing Objects and ARC

In languages that use an object reference model, you need a way to create an object before using it, and you also need a means of releasing the memory it occupies when it is no longer needed. If you don't dispose of it, you end filling up memory with objects you don't need any more, causing a problem known as a *memory leak*. To solve this issue languages like C# and Java, based on a virtual execution environment (or virtual machine) adopt garbage collection. While this make developer's life easier, this approach is subject to some complex performance-related issues that it isn't really relevant in explaining Object Pascal. So interesting as the issues are I don't wantto delve into them here.

In Object Pascal, you generally release the memory of an object by calling its special `Free` method (again, a method of `TObject`, available in each class). Free removes the object from memory after calling its destructor (which can have special clean up code). So you can complete the code snippet above as:

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;
  // use
  ...
  // free the memory
  ADay.Free;
end;
```

While this is the standard approach, the component library adds concepts like object ownership to significantly lessen the impact of manual memory management, making this a relatively simple issue to handle.

To further simplify memory management, the Object Pascal compilers for the mobile platforms introduce an additional mechanism called Automatic Reference Counting (or ARC). The ARC model uses reference counting and some other advanced techniques to automatically dispose of objects that are not needed any longer (or have no references pointing to them). So on these platforms, the call to `Free` an object is generally superfluous: as the execution of the code above reaches the `end` statement, the `ADay` variable goes out of scope and the referenced object is automatically deleted. In any case, if you keep the `Free` statement in the code, it does no harm at all and everything will work smoothly both with the desktop and mobile compilers.

---

note Automatic Reference Counting (ARC) is a standard memory management technique for iOS development in ObjectiveC and Swift, the preferred languages in Apple's Xcode. Object Pascal borrowed from that model, including weak references and other elements, but extends it in a few ways and has a very efficient implementation.

There is much more to memory management and ARC that you need to know, but given this is a rather important topic and not a simple one, I decided to offer only a short introduction here and have a full chapter focused on this topic, namely Chapter 13. In that chapter I'll show you in detail the different techniques used on each platform and those that work across all platforms.

What's Nil?

As I've mentioned, a variable can refer to an object of a given class. But it might not be initialized yet, or the object it used to refer to might not be available any longer. This is where you can use `nil`. This is a constant value indicating that the variable is not assigned to any object (or it is assigned to a `0` memory location). When a variable of a class type has no value, you can initialize it this way:

```
ADay := nil;
```

To check if an object has been assigned the variable, you can write either of the following expressions:

```
if ADay <> nil then ...
if Assigned (ADay) then ...
```

Do not make the mistake of assigning `nil` to an object to remove it from memory. Setting an object to nil and freeing it are two different operations, at least on the desktop compilers (ARC makes things a little different and it might free an object when you set a reference to `nil`). So you often need to both free an object and set its reference to `nil`, or call a special purpose procedure that does both operations at once, called `FreeAndNil`. Again, more information and some actual demos will be coming in Chapter 13.

Records vs. Classes in Memory

As I've mentioned earlier, one of the main differences between records and objects relates to their memory model. Record type variables use local memory, they are passed as parameters to functions by value by default, and they have a "copy by value" behavior on assignments. This contrasts with class type variables that are allocated on the dynamic memory heap, are passed by reference, and have a "copy by reference" behavior on assignments (thus copying the reference to the same object in memory, not the actual data).

note A consequence of this different memory management is that records lack inheritance and polymorphisms, two features we'll be focusing on in the next chapter.

For example, when you declare a record variable on the stack, you can start using it right away, without having to call its constructor. This means record variables are leaner (and more efficient) on the memory manager than regular objects, as they do not participate in the management of the dynamic memory and ARC. These are the key reasons for using records instead of objects for small and simple data structures.

Regarding the difference in the way records and objects are passed as parameters, consider that the default is to make a full copy of the memory block representing the record (including all of its data) or of the reference to the object (while the data is not copied). Of course, you can use `var` or `const` record parameters to modify the default behavior for passing record type parameters.

Private, Protected, and Public

A class can have any amount of data fields and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated,* inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30th. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Proper encapsulation is particularly important because it gives the class writer the freedom to modify the internal representation in a future version.

The concept of encapsulation is quite simple: just think of a class as a "black box" with a small, visible portion. The visible portion, called the *class interface,* allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Rather you use the methods to access the data of an object or act on it.

Encapsulation using private and protected members is the object-oriented solution to a classic programming goal known as *information hiding*.

Object Pascal has three basic access (or visibility) specifiers: `private`, `protected`, and `public`. A fourth one, `published`, will be discussed in the Chapter 10. Here are the three basic ones:

- The `private` access specifier denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
- The `public` access specifier denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.
- The `protected` access specifier is used to indicate methods and fields with limited visibility. Only the current class and its derived classes (or subclasses) can access `protected` elements. We'll discuss this keyword again in the "Protected Fields and Encapsulation" section of the next chapter.

---

note Two further access specifiers, *strict private* and *strict protected* were added to the language to match the behavior of other OOP languages. They'll be discussed shortly. They are not listed here because they are not very commonly used, despite their roles.

Generally, the fields of a class should be `private`; the methods are usually `public`. However, this is not always the case. Methods can be `private` or `protected` if they

are needed only internally to perform some partial operations. Fields can be `protected` if you are fairly sure that their type definition is not going to change and you might want to manipulate them directly in derived classes (as explained in the next chapter), although this is rarely recommended.

As a general rule, you should invariably avoid `public` fields, and generally expose some direct access to data using properties, as we'll see in detail in Chapter 10. Properties are an extension to the encapsulation mechanism of other OOP languages and are very important in Object Pascal.

As mentioned, access specifiers only restrict code outside a unit from accessing certain members of classes declared in that unit. This means that if two classes are in the same unit, there is no protection for their private fields, something covered in the next section in more detail.

An Example of Private Data

As an example of the use of these access specifiers for implementing encapsulation, consider this new version of the `TDate` class:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

In this version, the fields are now declared to be `private`, and there are some new methods. The first, `GetText`, is a function that returns a string with the date. You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which simply return the corresponding `private` data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation, weaken the abstraction, and make it harder to modify the internal implementation of a class later on. Access functions should be provided only if they are part of the logical interface of the class you are implementing, not because there are matching fields.

The second new method is the `Increase` procedure, which increases the date by one day. This is far from simple, because you need to consider the different lengths of the various months as well as leap and non-leap years. What I'll do to make it easier to write the code is to change the internal implementation of the class to use Object

Pascal `TDateTime` type for the internal implementation. So the actual class will change to the following code you can find in the `Dates2` application project:

```
type
  TDate = class
  private
    FDate: TDateTime;
  public
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the `private` portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

---

note In this new version of the class, the (only) field has an identifier that starts with the letter "F". This is a fairly common convention in Object Pascal and one I'll often use in the book. While this is the official style, there is another alternative and commonly used convention, which is use the letter "f" lowercase as field prefix.

To end this section, let me finish describing the project, by listing the source code of the class methods, which rely on a few system functions for mapping dates to the internal structure and vice verse:

```
procedure TDate.SetValue (m, d, y: Integer);
begin
  fDate := EncodeDate (y, m, d);
end;

function TDate.GetText: string;
begin
  Result := DateToStr (fDate);
end;

procedure TDate.Increase;
begin
  fDate := fDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils and YearOf in DateUtils
  Result := IsLeapYear (YearOf (fDate));
end;
```

Notice also how the code to use the class cannot refer to the Year value any more, but it can only return information about the date object as allowed by its methods:

```
var
```

```
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;

  // use
  ADay.SetValue (1, 1, 2016);
  ADay.Increase;

  if ADay.LeapYear then
    Show ('Leap year: ' + ADay.GetText);

  // free the memory (for non ARC platforms)
  ADay.Free;
```

The output is not much different than before:

```
Leap year: 1/2/2016
```

When Private Is Really Private

I have already mentioned that, differently from most other OOP languages, in Object Pascal class access specifiers like `private` and `protected` only restrict access to given class members from code outside the unit in which the class is declared. In other words, any global function or any method of a class written in the same unit, can access the private data of any class in the unit. To overcome this anomaly (compared to other languages and compared to the concept of encapsulation), the language introduced also the `strict private` and `strict protected` specifiers.

These two specifiers work in the way you'd probably expect and followed by most other OOP languages, which means that other classes even within the same unit cannot access strict private symbols of a class and can access strict protected symbols only if they inherit from that class.

Even if these *strict* access specifier offer a better and safer implementation of encapsulation, most Object Pascal developers tend to stick with the classic, loose version, and simply avoid bypassing the rules in their code.

note The C++ language has the concept of friend classes, that is classes allowed to access another class private data. Following this terminology, we can say that in Object Pascal all classes in the same unit are automatically considered as friend classes.

Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a field of a class, you only need to change the code of some methods of that class referring to the given field, and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with a practical example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of the form:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This works but has two problems. First, the data (`nClicks`) is not connected to a specific instance of the form, but to the entire program. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

The second problem is that if you define the data as a global variable or as a public field of a form, you won't be able to modify its implementation in the future without affecting the code that uses the data. For example, if you only have to read the current value from other forms, you can declare the data as private and provide a method to read the value:

```
type
  TForm1 = class(TForm)
    // components and event handlers here
  public
    function GetClicks: Integer;
  private
    nClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := nClicks;
end;
```

An even better solution is to add a property to the form, as we'll see in Chapter 10. You can experiment with this code by opening the `ClicksCount` application project. In short, the form of this project has two buttons and a label at the top, with most of the surface empty for a user to click (or tap) onto it. In this case, the count is increased and the label is updated with the new value:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  Inc (nClicks);
  Label1.Text := nClicks.ToString;
end;
```

You can see the application in action in Figure 7.1. The project's form also has two buttons, one for creating a new form of the same type and the second to close it (so you can give focus back to the previous form). This is done to emphasize how different instances of the same form type each have their own clicks count. This is the code of the two methods:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewForm: TForm1;
begin
  NewForm := TForm1.Create(Application);
  NewForm.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Figure 7.1: The form of the ClicksCount application project showing the number of clicks or taps on a form (tracked using private form data)

The Self Keyword

We've seen that methods are very similar to procedures and functions. The real difference is that methods have an extra, implicit parameter. This is a reference to the current object, the object the method is applied to. Within a method you can refer to this parameter—the current object—using the `Self` keyword. This extra hidden parameter is needed when you create several objects of the same class, so that each time you apply a method to one of the objects, the method will operate only on its own data and not affect the other sibling objects.

note We have already seen the role of the `Self` keywords in Chapter 5, while discussing records. The concept and its implementation are very similar. Again, historically `Self` was first introduced for classes and later extended to records, when methods were added also to this data structure.

For example, in the `SetValue` method of the `TDate` class, listed earlier, we simply use `Month`, `Year`, and `Day` to refer to the fields of the current object, something you might express as:

```
Self.Month := m;
Self.Day := d;
```

This is actually how the Object Pascal compiler translates the code, *not* how you are supposed to write it. The `Self` keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to make tricky code more readable.

---

note The C++, Java, C#, and JavaScript languages have a similar feature based on the keyword this. However in JavaScript using this is a method to refer to object fields is compulsory, unlike in C++, C# and Java.

All you really need to know about `Self` is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have that extra hidden parameter, `Self`. Because all this happens behind the scenes, you don't need to know how `Self` works at this time.

The second important think to know is that you can explicitly use `Self` to refer to the current object as a whole, for example passing the current object as parameter to another function.

Creating Components Dynamically

As an example of what I've just mentioned, the `Self` keyword is often used when you need to refer to the current form explicitly in one of its methods. The typical example is the creation of a component at run time, where you must pass the owner of the component to its `Create` constructor and assign the same value to its `Parent` property. In both cases, you have to supply the current form object as parameter or value, and the best way to do this is to use the `Self` keyword.

---

note The ownership of a component indicates a lifetime and memory management relationship between two objects. When the owner of a component is freed the component will also be freed. Parenthood refers to visual controls hosting a child control within their surface.

To demonstrate this kind of code, I've written the `CreateComps` application project. This application has a simple form with no components and a handler for its `OnMouseDown` event, which also receives as its parameter the position of the mouse

click. I need this information to create a button component in that position. Here is the code of the method:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Position.X := X;
  Btn.Position.Y := Y;
  Btn.Height := 35;
  Btn.Width := 135;
  Btn.Text := Format ('At %d, %d', [X, Y]);
end;
```

Notice you need to add the `FMX.StdCtrls` unit to the uses statement to compile this event handler.

The effect of this code is to create buttons at mouse-click positions, with a caption indicating the exact location, as you can see in Figure 7.2. In the code above, notice in particular the use of the `Self` keyword, as the parameter of the `Create` method and as the value of the `Parent` property.

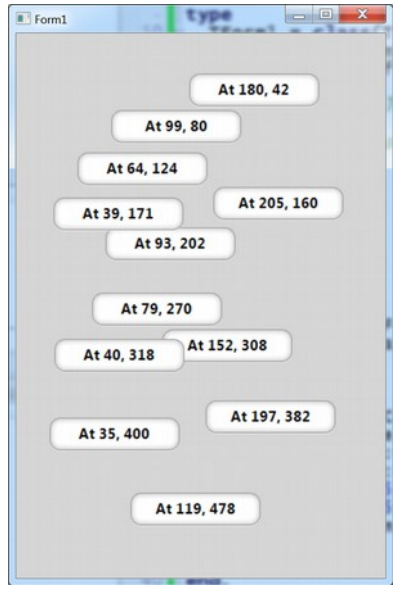


Figure 7.2: The output of the CreateComps application project example on a mobile device

When writing a procedure like the code you've just seen, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't

make any practical difference (although it won't be good coding practice), but if there are multiple instances of a form, using `Form1` would really be an error.

In fact, if the `Form1` variable refers to a form of that type being created (generally the first one) and if you create two instances of the same form type, by clicking on any following forms the new button will always be displayed in the first one. Its owner and Parent will be `Form1` and not the form on which the user has clicked.

In general, writing a method in which you refer to a particular instance of the same class when the current object is required is a really a bad, bad OOP practice.

Constructors

In the code above, to create an object of a class (or allocate the memory for an object), I've called the `Create` method. This is a *constructor,* a special method that you can apply to a class to allocate memory for a new instance of that class:

```
ADay := TDate.Create;
```

The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on.

When you are creating an objet, its memory is initialized. All of the data of the new instance to set to zero (or nil, or empty string, or the proper “default” value for a given data type).

If you want your instance data to start out with a nonzero value (particular when a zero value makes little sense as a default), then you need to write a custom constructor to do that. The new constructor can be called `Create`, or it can have any other name. What determines its role is not the name but the use of a specific keyword, `constructor`.

note In other words, Object Pascal supports named constructors, while in many OOP languages the constructor must be named after the class itself. With named constructors, you can have more than one constructor with the same parameters (beside overloading the `Create` symbol – overloading is covered in the next section). Another very special feature of the language, quite unique among OOP languages, is that constructors can also be virtual. I'll show some examples the cover the consequences later in the book.

The main reason to add a custom constructor to a class is to initialize its data. If you create objects without initializing them, calling methods later on may result in odd behavior or even a run-time error. Instead of waiting for these errors to appear, you should use preventive techniques to avoid them in the first place. One such tech-

nique is the consistent use of constructors to initialize objects' data. For example, we must call the `SetValue` procedure of the `TDate` class after we've created the object. As an alternative, we can provide a customized constructor, which creates the object and gives it an initial value:

```
constructor TDate.Create;
begin
  FDate := Today;
end;

constructor TDate.CreateFromValues (m, d, y: Integer);
begin
  FDate := SetValue (m, d, y);
end;
```

You can use these constructors as follows, as I've done in the `Date3` application project, in the code attached to two separate buttons:

```
ADay1 := TDate.Create;
ADay2 := TDate.CreateFromValues (12, 25, 2015);
```

Although in general you can use any name for a constructor, keep in mind that if you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available. If you are developing and distributing code for others to use, a programmer calling this default `Create` constructor might bypass the initialization code you've provided. By defining a `Create` constructor with some parameters (or none, as in the example above), you replace the default definition with a new one and make its use compulsory.

In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the `destructor` keyword and invariably called `Destroy`. This destructor method which can perform some resource cleanup before an object is destroyed, but in many cases a custom destructor is not required.

Just as a constructor call allocates memory for the object, a destructor call frees the memory. Custom destructors are really only needed for objects that acquire resources in their constructors or during their lifetime.

Differently from the default `Create` constructor, the default `Destroy` destructor is virtual and it is highly recommended that developer override this virtual destructor (virtual methods are covered in the next chapter).

That's because instead of a calling destructor directly to free an object, it is a good a common Object Pascal programming practice to call the special `Free` method of the `TObject` class, which in turn calls `Destroy` only if the object exists—that is, if it is not `nil`. So, it you define a destructor with a different name, it won't be called by `Free`. Again, more on this topic when we'll focus on memory management in Chapter 13.

note As covered in the next chapter, `Destroy` is a virtual method. You can replace its base definition with a new one in an inherited class marking it with the `override` keyword. By the way, having a static method that calls a virtual one is a very common programming style, called the *template pattern*. In a destructor, you should generally only write resource cleanup code. Try to avoid more complex operations, likely to raise exceptions or to take a significant amount of time.

Managing Local Class Data with Constructors and Destructors

Even if I'll cover more complex scenarios later in the book, here I want to show you a simple case of resource protection using a constructor and a destructor. This is the most common scenario for using a destructor. Suppose you have a class with the following structure (also part of the `Date3` application project):

```
type
  TPerson = class
  private
    FName: string;
    FBirthDate: TDate;
  public
    constructor Create (name: string);
    destructor Destroy; override;
    // some actual methods
    function Info: string;
  end;
```

This class has a reference to another, internal object called `FDate`. When an instance of the `TPerson` class is created, this internal (or child) object should also be created, and when the instance is destroyed, the child object should also be disposed of. Here is how you can write the code of the constructor and overridden destructor, and of the internal method that can always take for granted that the internal object exists:

```
constructor TPerson.Create (name: string);
begin
  FName := Name;
  FBirthDate := TDate.Create;
end;

destructor TPerson.Destroy;
begin
  FBirthDate.Free;
  inherited;
end;

function TPerson.Info: string;
begin
  Result := FName + ': ' + FBirthDate.GetText;
```

```
end;
```

note To understand the `override` keyword used to define the destructor and the `inherited` keyword within its definition, you'll have to wait until the next chapter. For now suffice to say the first is used to indicate that the class has a new definition replacing the base `Destroy` destructor, while the latter is used to invoke that base class destructor. Notice also that `override` is used in the method declaration, but not in the method implementation code.

Now you can use an object of the external class as in the following scenario, and the internal object will be properly created when the `TPerson` object is created and destroyed in a timely fashion when `TPerson` is destroyed:

```
var
  Person: TPerson;
begin
  Person := TPerson.Create ('John');
  // use the class and its internal object
  Show (Person.Info);
  Person.Free;
end;
```

Again, you can find this code as part of the `Dates3` application project.

While this is the standard way of writing such code in Object Pascal, the ARC-enabled compilers won't require the explicit calls to `Free` in the destructor or in the sample code above. So under ARC the destructor won't be required, although (again) it won't harm.

note Currently the only Object Pascal compilers that enable ARC are the Android compilers, the iOS device compiler and the iOS Simulator compiler. More might become available in the future. The differences in memory usage and coding style are highlighted in Chapter 13.

Overloaded Methods and Constructors

Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. We have already seen how overloading works for global functions and procedures the same rules apply to methods. By checking the parameters, the compiler can determine which version of the method you want to call.

Again, there are two basic rules for overloading:

- Each version of the method must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters or both. The return type, instead, cannot be used to distinguish among two methods.

If overloading can be applied to all of the methods of a class, this feature is particularly relevant for constructors, because we can have multiple constructors and call them all `Create`, which makes them easy to remember.

note Historically, overloading was added to C++ specifically to allow the use of multiple constructors, given they must have the same name (the name of the class). In Object Pascal, this feature could have been considered unnecessary, simply because multiple constructors can have different specific names, but was added to the language anyway as it also proved to be useful in many other scenarios.

As an example of overloading, I've added to the `TDate` class two different versions of the `SetValue` method:

```
type
  TDate = class
  public
    procedure SetValue (Month, Day, Year: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;

procedure TDate.SetValue (Month, Day, Year: Integer);
begin
  FDate := EncodeDate (Year, Month, Day);
end;

procedure TDate.SetValue(NewDate: TDateTime);
begin
  FDate := NewDate;
end;
```

After this simple step, I've added to the class two separate `Create` constructors, one with no parameters, which hides the default constructor, and one with the initialization values. The constructor with no parameters uses today's date as the default value:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (Month, Day, Year: Integer); overload;

constructor TDate.Create (Month, Day, Year: Integer);
begin
  FDate := EncodeDate (Year, Month, Day);
end;

constructor TDate.Create;
begin
  FDate := Date;
end;
```

Having these two constructors makes it possible to define a new `TDate` object in two different ways:

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create (1999, 12, 25);
  Day2 := TDate.Create; // today
```

This code is part of the Dates4 application project.

The Complete TDate Class

Throughout this chapter, I've shown you bits and pieces of the source code for different versions of a `TDate` class. The first version was based on three integers to store the year, the month, and the day; a second version used a field of the `TDateTime` type provided by Delphi. Here is the complete interface portion of the unit that defines the `TDate` class:

```
unit Dates;

interface

type
  TDate = class
  private
    FDate: TDateTime;
  public
    constructor Create; overload;
    constructor Create (Month, Day, Year: Integer); overload;
    procedure SetValue (Month, Day, Year: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string;
  end;

implementation
...
```

The aim of the new methods, `Increase` and `Decrease` (which have a default value for their parameter), is quite easy to understand. If called with no parameter, they change the value of the date to the next or previous day. If a `NumberOfDays` parameter is part of the call, they add or subtract that number:

```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
  FDate := FDate + NumberOfDays;
end;
```

The method `GetText` returns a string with the formatted date, using the `DateToStr` function for the conversion:

```
function TDate.GetText: string;
begin
  GetText := DateToStr (FDate);
end;
```

We've already seen most of the methods in the previous sections, so I won't provide the complete listing; you can find it in the code of the ViewDate example I've written to test the class. The form is a little more complex than others in the book, and it has a caption to display a date and six buttons, which can be used to modify the value of the object. You can see the main form of the ViewDate application project at run time in Figure 7.2. To make the label component look nice, I've given it a big font, made it as wide as the form, set its `Alignment` property to `taCenter`, and set its `AutoSize` property to `False`.

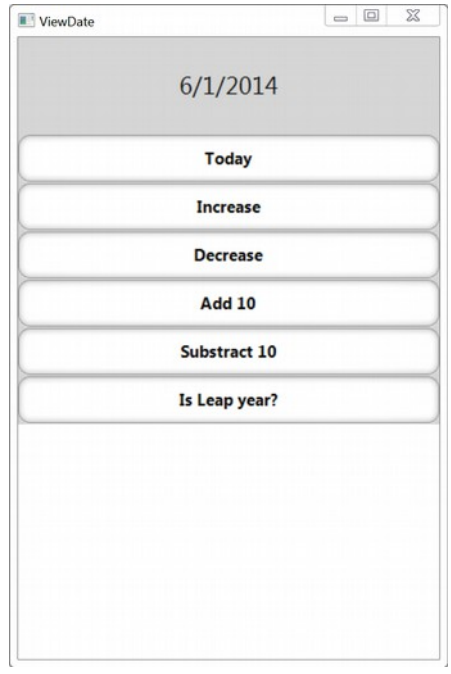


Figure 7.2:
The output of the ViewDate application at start-up

The start-up code of this program is in the `OnCreate` event handler of the form. In the corresponding method, we create an instance of the `TDate` class, initialize this object, and then show its textual description in the `Text` of the label, as shown in Figure 7.2.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TDate.Create;
  LabelDate.Text := ADay.GetText;
end;
```

`ADay` is a private field of the class of the form, `TDateForm`. By the way, the name for the class is automatically chosen by the development environment when you change the `Name` property of the form to `DateForm`.

The specific date object is created when the form is created (setting up the same relationship we saw earlier between the person class and its date sub-object) and is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  ADay.Free;
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the `ADay` object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  ADay.SetValue (Today);
  LabelDate.Text := ADay.GetText;
end;
```

An alternative way to write the last method is to destroy the current object and create a new one:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  ADay.Free;
  ADay := TDate.Create;
  LabelDate.Text := ADay.GetText;
end;
```

In this particular circumstance, this is not a very good approach (because creating a new object and destroying an existing one entails a lot of time overhead, when all we need is to change the object's value), but it allows me to show you a couple of Object Pascal techniques. The first thing to notice is that we destroy the previous object before assigning a new one. The assignment operation, in fact, replaces the reference, leaving the object in memory (even if no pointer is referring to it). When you assign an object to another object, the compiler simply copies the reference to the object in memory to the new object reference.

This is what happens on desktop platforms. On ARC-enabled mobile platforms, instead, the code above could be written without the call to `Free`:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  ADay := TDate.Create;
```

```
  LabelDate.Text := ADay.GetText;
end;
```

As the new object is assigned, in fact, the old one looses the only reference and is automatically disposed.

One side issue is how do you copy the data from one object to another. This case is very simple, because there is only one field and a method to initialize it. In general if you want to change the data inside an existing object, you have to copy each field, or provide a specific method to copy all of the internal data. Some classes have an `Assign` method, which does this *deep-copy* operation.

---

note To be more precise, in the runtime library all of the classes inheriting from `TPersistent` have the `Assign` method, but most of those inheriting from `TComponent` don't implement it, raising an exception when it is called. The reason lies in the streaming mechanism supported by the runtime libraries and the support for properties of `TPersistent` types, but this is way too complex to delve into at this point of the book.

Nested Types and Nested Constants

Object Pascal allows you to declare new classes in the interface section of a unit, allowing other units of the program to reference them, or in the implementation section, where they are accessible only from methods of other classes of the same unit or from global routines implemented in that unit after the class definition.

A more recent additional option is the possibility to declare a class (or any other type) within another class. As any other member of the class, the nested class and other nested types can have a restricted visibility (say, private or protected). Relevant examples of nested types include enumerations used by the same class and other implementation-support classes.

A related syntax allows you to define a nested constant, a constant value associated with the class (again usable only internally if private or from the rest of the program if public). As an example, consider the following declaration of a nested class (extracted from the NestedClass unit of the NestedTypes application project):

```
type
  TOne = class
  private
    someData: Integer;
  public
    // nested constant
    const foo = 12;
    // nested type
```

```
    type
      TInside = class
      public
        procedure InsideHello;
      private
        Msg: string;
      end;
  public
    procedure Hello;
  end;

procedure TOne.Hello;
var
  ins: TInside;
begin
  ins := TInside.Create;
  ins.Msg := 'hi';
  ins.InsideHello;
  Show ('constant is ' + IntToStr (foo));
  ins.Free;
end;

procedure TOne.TInside.InsideHello;
begin
  msg := 'new msg';
  Show ('internal call');
  if not Assigned (InsIns) then
    InsIns := TInsideInside.Create;
  InsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
  Show ('this is a method of a nested/nested class');
end;
```

The nested class can be used directly within the class (as demonstrated in the listing) or outside the class (if it is declared in the public section), but with the fully qualified name `TOne.TInside`. The *full name* of the class is used also in the definition of the method of the nested class, in this case `TOne.TInside`. The hosting class can have a field of the nested class type immediately after you've declared the nested class (as you can see in the code of the NestedClass application project).

The class with the nested classes is used as follows:

```
var
  One: TOne;
begin
  One := TOne.Create;
  One.Hello;
  One.Free;
```

This produces the following output:

```
internal call
this is a method of a nested/nested class
constant is 12
```

How would you benefit from using a nested class in the Object Pascal language? The concept is commonly used in Java to implement event handler delegates and makes sense in C# where you cannot hide a class inside a unit. In Object Pascal nested classes are the only way you can have a field of the type of another private class (or inner class) without adding it to the global name space and making it globally visible.

If the internal class is used only by a method, you can achieve the same effect by declaring the class within the implementation portion of the unit. But if the inner class is referenced in the interface section of the unit (for example because it is used for a field or a parameter), it must be declared in the same interface section and will end up being visible. The trick of declaring such a field of a generic or base type and then casting it to the specific (private) type is much less clean than using a nested class.

note In chapter 10 there is a practical example in which nested classes come in handy, namely implementing a custom iterator for a `for in` loop.

08: inheritance

If the key reason for writing classes is encapsulation, the key reason for using inheritance among classes is flexibility. Combine the two concepts and you can have data types you can use and are not going to change with the ability to create modified versions of those types, in what was originally known as the "*open-close principle*". Now it is true that inheritance is a very strong binding leading to tight coupled code, but it is also true it offers great power to the developer (and, yes, more responsibility). Rather than opening up a debate on this feature, however, my goal here is to describe you how type inheritance works and specifically how it works in the Object Pascal language.

Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us. For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (and you have found it in a library), you might want to keep your changes separate.

A typical old-school alternative for having two similar versions of a class is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: in duplicating the code you also duplicate the bugs; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made over time. Moreover, this approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

To solve these kinds of problems in expressing similarities between classes, Object Pascal allows you to define a new class directly from an existing one. This technique is known as *inheritance* (or *subclassing,* or *type derivation*) and is one of the fundamental elements of object-oriented programming languages. To inherit from an existing class, you only need to indicate that class at the beginning of the declaration of the subclass. For example, this is done automatically each time you create a new form:

```
type
  TForm1 = class(TForm)
  end;
```

This simple definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can apply any public method of the `TForm` class to an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` class (which is the base class of all classes).

By comparison C++ and C# and Java would use something like:

```
class Form1 : TForm
{
   ...
}
```

As a simple example of inheritance, we can change the `ViewDate` application project of the last chapter slightly, deriving a new class from `TDate` and modifying one of its functions, `GetText`. You can find this code in the `DATES.PAS` file of the `DerivedDates` application project.

```
type
  TNewDate = class (TDate)
  public
     function GetText: string;
  end;
```

In this example, `TNewDate` is derived from `TDate`. It is common to say that `TDate` is an *ancestor* class or *base* class or *parent* class of `TNewDate` and that `TNewDate` is a *subclass, descendant* class, or *child* class of `TDate`.

To implement the new version of the `GetText` function, I used the `FormatDateTime` function, which uses (among other features) the predefined month names. Here is the `GetText` method, where *'dddddd'* stands for the long data format:

```
function TNewDate.GetText: string;
begin
  Result := FormatDateTime ('dddddd', fDate);
end;
```

Once we have defined the new class, we need to use this new data type in the code of the form of the `DerivedDates` project. Simply define the `ADay` object of type `TNewDate`, and call its constructor in the `FormCreate` method:

```
type
  TDateForm = class(TForm)
    ...
  private
    ADay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TNewDate.Create; // updated line
  DateLabel.text := TheDay.GetText;
end;
```

Without any other changes, the new application will work properly. The `TNewDate` class inherits the methods to increase the date, add a number of days, and so on. In addition, the older code calling these methods still works. Actually, to call the new version of the `GetText` method, we don't need to change the source code! The Object Pascal compiler will automatically bind that call to a new method. The source code of all the other event handlers remains exactly the same, although its meaning changes considerably, as the new output demonstrates (see Figure 8.1).

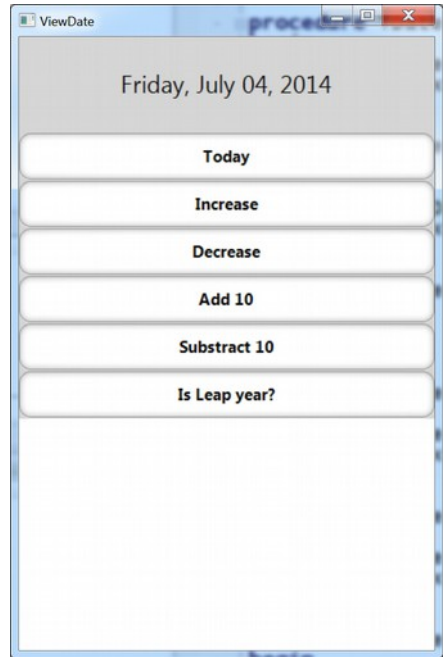


Figure 8.1:
The output of the DerivedDates program, with the name of the month and of the day depending on Windows regional settings

A Common Base Class

We have seen that if you can inherit from a given base class by writing:

```
type
  TNewDate = class (TDate)
    ...
  end;
```

But what happens if you omit a base class and write:

```
type
  TNewDate = class
    ...
  end;
```

In this case your class inherits from a base class, called `TObject`. In other words Object Pascal has a single-rooted class hierarchy, in which all classes directly or indirectly inherit from a common ancestor class. The most commonly used methods of `TObject` are `Create`, `Free`, and `Destroy`; but there are many others I'll use

throughout the book. A complete description of this fundamental class (that could be considered both part of the language and also part of the runtime library) with a reference to all of its methods is available in Chapter 17.

---

note The concept of a common ancestor class is present also in the C# and Java languages, where this is simply called `Object`. The C++ language, on the other hand, hasn't got such an idea, and a C++ program generally has multiple independent class hierarchies.

Protected Fields and Encapsulation

The code of the `GetText` method of the `TNewDate` class compiles only if it is written in the same unit as the `TDate` class. In fact, it accesses the `fDate` private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the `fDate` field as protected or add a simple, possibly protected method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write subclasses. However, this violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If ten derived classes are accessing this data, changing its definition means potentially modifying the code in each of the ten classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail below in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the subclasses, then your design might not follow the object-oriented principles.

Remember also that protected fields share the same access rules of private ones, so that any other class in the same unit can always access protected members of other classes. As mentioned in the previous chapter, you can use stronger encapsulation by using the strict protected access specifier.

Using the "Protected Hack"

> *If you are new to Object Pascal and to OOP, this is a rather advanced section you might want to skip the first time you are reading this book, as it might be quite confusing.*

Given how unit protection works, even protected members of base classes of classes declared in the current unit can be directly accesses. This is the rationale behind what it generally called "*the protected hack*", that is the ability to define a derived class identical to its base class for the only purpose of gaining access to the protected member of the base class. Here is how it works.

We've seen that the `private` and `protected` data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this simple class (part of the Protection application project):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

The `GetValue` method simply returns a string with the two integer values:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Once you place this class in its own unit, you won't be able to access its protected portion from other units directly. Accordingly, if you write the following code,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20;  // won't compile
  Show (Obj.GetValue);
  Obj.Free;
end;
```

the compiler will issue an error message, *Undeclared identifier: "ProtectedData."* At this point, you might think there is no way to access the protected data of a class defined in a different unit. However, there is a way around it. Consider what happens if you create an apparently useless derived class, such as

```
type
  TFake = class (TTest);
```

Now, in the same unit where you have declared it, you can call any protected method of the `TFake` class. In fact you can call protected methods of a class declared in the same unit. How does this helps using an object of class `TTest`, though? Considering that the two classes share the same exact memory layout (as there are no differences) you can force the compiler to treat an object of a class like one of the other, with what is generally a type-unsafe cast:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  Show (Obj.GetValue);
  Obj.Free;
end;
```

This code compiles and works properly, as you can see by running the Protection application project. Again, the reason is that the `TFake` class automatically inherits the protected fields of the `TTest` base class, and because the `TFake` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are rare times when using this technique is the best solution, as you'll see by looking at the library source code and the code of many components.

Overall, this technique is a *hack* and it should be avoided whenever possible, although it can be considered to all effects as part of the language specification and is available on all platforms and in all present and past versions of Object Pascal.

From Inheritance to Polymorphism

Inheritance is a nice technique in terms of letting you avoid code duplication and share code methods among different classes. Its true power, however, comes from the ability to handle objects of different classes in a uniform manner, something often indicated in object-oriented programming languages by the term *polymorphism* or referenced as *late binding*.

There are several elements we have to explore to fully understand this feature: type compatibility among derived classes, virtual methods, and more, as covered in the next few sections.

Inheritance and Type Compatibility

As we have seen to some extent, Object Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a `Boolean` variable, at least not without an explicit typecast. The basic rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type has the same name and their definition comes from the same unit.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. That is because a dog is an animal! So, although this might surprise you, the following constructor calls are both legal:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

In more precise terms, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
MyAnimal := MyDog;  // This is OK
MyDog := MyAnimal;  // This is an error!!!
```

In fact, while we can always say that a dog is an animal, we cannot assume that any given animal is a dog. This might be true at times, but not always. This is quite logical, and the language type compatibility rules follow this same logic.

Before we look at the implications of this important feature of the language, you can try out the `Animals1` application project, which defines the two simple `TAnimal` and `TDog` classes, inheriting one from the other:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    FKind: string;
  end;
```

```
  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

The two `Create` methods simply set the value of `FKind`, which is returned by the `GetKind` function.

The form of this example, shown in Figure 8.2, has two radio buttons (hosted by a panel) to pick an object of one or the other class. This object is stored in the private field `MyAnimal` of type `TAnimal`. An instance of this class is created and initialized when the form is created and re-created each time one of the radio buttons is selected (here I'm showing only the code of the second radio button):

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

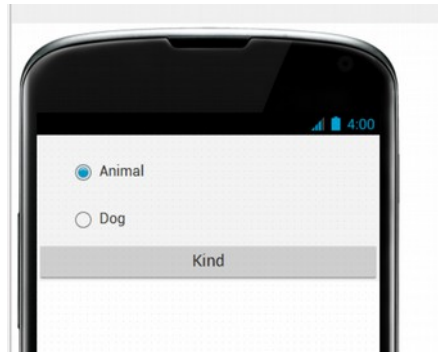


Figure 8.2: The form of the Animals1 application project in the development environment

Finally, the Kind button calls the `GetKind` method for the current animal and displays the result in the memo covering the bottom part of the form:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  Show(MyAnimal.GetKind);
end;
```

Late Binding and Polymorphism

Object Pascal functions and procedures are usually based on *static binding,* which is also called *early binding*. This means that a method call is resolved by the compiler or the linker, which replaces the request with a call to the specific memory location where the compiled function or procedure resides. (This is also known as the *address* of the function.) Object-oriented programming languages allow the use of another form of binding, known as *dynamic binding,* or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, simply because of the type-compatibility rule discussed in the previous section.

note Object Pascal methods default to early binding, like C++ and C#. One of the reasons is this is more efficient. Java, instead, defaults to late binding (and offers ways to indicate to the compiler it can optimize a method using early binding).

Suppose that a class and its subclass (let's say `TAnimal` and `TDog`, again) both define a method, and this method has late binding. Now you can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The Animals2 application project extends the Animals1 project to demonstrate this technique. In the new version, the `TAnimal` and the `TDog` classes have a new method: `Voice`, which means to output the sound made by the selected animal, both as text and as sound. This method is defined as `virtual` in the `TAnimal` class and is later overridden when we define the `TDog` class, by the use of the `virtual` and `override` keywords:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

Of course, the two methods also need to be implemented. Here is a simple approach:

```
function TAnimal.Voice: string;
```

```
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

Now what is the effect of the call `MyAnimal.Voice`? It depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice` instead. This happens only because the function is `virtual`.

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined after this method call or outside its scope. The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` subclasses.

This is the key technical reason why object-oriented programming languages favor reusability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition—the ancestor classes of the hierarchy need to be designed very carefully.

The Animals2 project demonstrates the use of these new classes and has a form similar to that of the previous example. This code is executed by clicking on the button, showing the output and also producing some sound:

```
begin
  Show (MyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + MyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

note The application uses a MediaPlayer component to play one of the two sound files that come with the application (the sound files are named after the actual sounds, that is the values returned by the `Voice` method). A rather random noise for the generic animal, and some barking for the dog. Now the code works easily on Windows, as long as the files are in the proper folder, but it requires some effort for the deployment on mobile platforms.

Take a look at the actual demo to see how the deployment and the folders were structured.

Overriding, Redefining, and Reintroducing Methods

As we have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as `virtual` in the ancestor class. Otherwise, if it was a static method, there is no way to activate late binding, other than by changing the code of the ancestor class.

note You might remember I used the same keyword also in the last chapter to override the `Destroy` default destructor, inherited from the base `TObject` class.

The rules are simple: A method defined as static remains static in every subclass, unless you hide it with a new virtual method having the same name. A method defined as `virtual` remains late-bound in every subclass. There is no way to change this, because of the way the compiler generates different code for late-bound methods.

To redefine a static method, you simply add a method to a subclass having the same parameters or different parameters than the original one, without any further specifications. To override a `virtual` method, you must specify the same parameters and use the `override` keyword:

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; // static method
  end;

  TMySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

The redefined method, `Two`, has no late binding. So when you apply it to a variable of the base class, it calls the base class method no matter what (that is, even if the variable is referring to an object of the derived class, that has a different version for that method).

There are typically two ways to override a method. One is to replace the method of the ancestor class with brand a new version. The other is to add some more code to the existing method. This second approach can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write

```
procedure TMySubClass.One;
```

```
begin
  // new code
  ...
  // call inherited procedure TMyClass.One
  inherited One;
end;
```

You might wonder why you need to use the `override` keyword. In other languages, when you redefine a virtual method in a subclass, you automatically override the original one. However, having a specific keyword allows the compiler to check the correspondence between the name of the method in the ancestor class and name of the method in the subclass (misspelling a redefined function is a common error in some other OOP languages), check that the method was virtual in the ancestor class, and so on.

note There is another popular OOP language that has the same override keyword, C#. This is not surprising, given the fact the languages share a common designer. Anders Hejlsberg has some lengthly articles explaining why the override keyword is a fundamental versioning tool for designing libraries, as you can read at `http://www.artima.com/intv/nonvirtual.html`. More recently, Apple's Swift language has also adopted the override keyword to modify methods in derived classes.

Another advantage of this keyword is that if you define a static method in any class inherited by a class of the library, there will be no problem, even if the library is updated with a new virtual method having the same name as a method you've defined. Because your method is not marked by the `override` keyword, it will be considered a separate method and not a new version of the one added to the library (something that would probably break your existing code).

The support for overloading adds some further complexity to this picture. A subclass can provide a new version of a method using the `overload` keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
  end;
```

Notice that the method doesn't need to be marked as `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning *Method 'One' hides virtual method of base type 'TMyClass'.* To avoid this message

from the compiler and to instruct the compiler more precisely on your intentions, you can use the specific `reintroduce` directive:

```
type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); reintroduce; overload;
  end;
```

You can find this code in the `ReintroduceTest` application project and experiment with it further.

note A scenario in which the reintroduce keyword is used is when you want to add a custom `Create` constructor to a component class, that already inherits a virtual `Create` constructor from the `TComponent` base class.

Inheritance and Constructors

As we have seen, you can use the `inherited` keyword the invoke same name method (or also a different method) in a method of a derived class. The same is true also for constructors. While in other languages like C++, C# or Java, the call to the base class constructor is implicit and compulsory (when you have to pass parameters to the base class constructor), in Object Pascal calling a base class constructor is not strictly required.

In most cases, however, manually calling the base class constructor is extremely important. This is the case, for example, for any component class, as the component initialization is actually done a the `TComponent` class level:

```
constructor TMyComponent.Create (Owner: TComponent);
begin
  inherited Create (Owner);
  // specific code...
end;
```

This is particularly important because for components Create is a virtual method. Similarly for all classes, the `Destroy` destructor is a virtual method and you should remember calling `inherited` in it.

One question remains: If you are creating a class, which only implicitly inherits from `TObject`, in its constructors do you need to call the base `TObject` `Create` constructor? From a technical point of view, and answer is “no” given that constructor is empty. However, I consider it s good habit to always call the base class constructor, no matter what. If you are a performance maniac, however, I'll concede this can

needlessly slow down your code... by a completely unnoticeable fraction of microsecond.

Jokes aside, there are good reasons for both approaches, but particularly for a beginner with the language I recommend always calling the base class constructor as good programming habit, promoting safer coding.

Virtual versus Dynamic Methods

In Object Pascal, there are two different ways to activate late binding. You can declare a method as `virtual`, as we have seen before, or declare it as `dynamic`. The syntax of these two keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

Virtual methods are based on a *virtual method table* (or VMT, but colloquially also known as a *vtable*). A virtual method table is an array of method addresses. For a call to a virtual method, the compiler generates code to jump to an address stored in the *n*th slot in the object's virtual method table.

Virtual method tables allow fast execution of the method calls. Their main drawback is that they require an entry for each virtual method for each descendant class, even if the method is not overridden in the subclass. At times, this has the effect of propagating virtual method table entries throughout a class hierarchy (even for methods that aren't redefined). This might require a lot of memory just to store the same method address a number of times.

Dynamic method calls, on the other hand, are dispatched using a unique number indicating the method. The search for the corresponding function is generally slower than the simple one-step table lookup for virtual methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method. For large or deep object hierarchies, using dynamic methods instead of virtual methods can result in significant memory savings with only a minimal speed penalty.

From a programmer's perspective, the difference between these two approaches lies only in a different internal representation and slightly different speed or memory usage. Apart from this, virtual and dynamic methods are the same.

Now having explained the difference between these two models, it is important to underline that in the largest number of cases, application developers use `virtual` rather than `dynamic`.

Message Handlers on Windows

When you are building applications for Windows, a special purpose late-bound method can be used to handle a Windows system message. For this purpose Object Pascal provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter of the proper type. The `message` directive is followed by the number of the Windows message the method wants to handle. For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `wm_User` Windows constant:

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage); message wm_User;
  end;
```

The name of the procedure and the actual type of the parameters are up to you, as log as the physical data structure matches with the Windows message structure. The units use to interface with the Windows API include a number of predefined record types for the various Windows messages. This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions, but it absolutely not compatible with other operating systems (like OS X, iOS, and Android).

Abstracting Methods and Classes

When you are creating a hierarchy of classes, at times it is difficult to determine which is the base class, given it might not represent an actual entity, but only be used to hold some shared behavior. An example would be an animal base class for something like a cat or a dog class. Such a class for which you are not expected to create any object is often indicated as an *abstract* class, because it has no concrete and complete implementation. An abstract class can have abstract methods, methods that don't have an actual implementation.

Abstract Methods

The `abstract` keyword is used to declare virtual methods that will be defined only in subclasses of the current class. The `abstract` directive fully defines the method; it is

not a forward declaration. If you try to provide a definition for the method, the compiler will complain.

In Object Pascal, you can create instances of classes that have `abstract` methods. However, when you try to do so, the compiler issues the warning message: *Constructing instance of <class name> containing abstract methods*. If you happen to call an `abstract` method at run time, Delphi will raise a specific runtime exception.

note C++, Java, and other languages use a more strict approach: in these languages, you cannot create instances of abstract classes.

You might wonder why you would want to use `abstract` methods. The reason lies in the use of polymorphism. If class `TAnimal` has the `virtual abstract` method `Voice`, every subclass can redefine it. The advantage is that you can now use the generic `MyAnimal` object to refer to each animal defined by a subclass and invoke this method. If this method was not present in the interface of the `TAnimal` class, the call would not have been allowed by the compiler, which performs static type checking. Using a generic `MyAnimal` object, you can call only the method defined by its own class, `TAnimal`.

You cannot call methods provided by subclasses, unless the parent class has at least the declaration of this method—in the form of an `abstract` method. The next application project, Animals3, demonstrates the use of `abstract` methods and the abstract call error. Here are the interfaces of the classes of this new example:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  private
    Kind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

The most interesting portion is the definition of the class `TAnimal`, which includes a `virtual abstract` method: `Voice`. It is also important to notice that each derived class overrides this definition and adds a new virtual method, `Eat`. What are the implications of these two different approaches? To call the `Voice` function, we can simply write the same code as in the previous version of the program:

```
Show (MyAnimal.Voice);
```

How can we call the `Eat` method? We cannot apply it to an object of the `TAnimal` class. The statement

```
Show (MyAnimal.Eat);
```

generates the compiler error *Field identifier expected*.

To solve this problem, you can use a dynamic and safe type cast to treat the `TAnimal` object as a `TCat` or as a `TDog` object, but this would be a very cumbersome and error-prone approach:

```
begin
  if MyAnimal is TDog then
    Show (TDog(MyAnimal).Eat)
  else if MyAnimal is TCat then
    Show (TCat(MyAnimal).Eat);
```

This code will be explained later in the section “Safe Type Cast Operators”. Adding the virtual method definition to the `TAnimal` class is a typical solution to the problem, and the presence of the `abstract` keyword favors this choice. The code above looks ugly, and avoiding such a code is precisely the reason for using polymorphism.

Finally notice that when a class has an abstract method, it is often considered to be an abstract class. However you can also specifically mark a class with the abstract directive (and it will be considered an abstract class even if it has no abstract methods). Again, in Object Pascal this won't prevent you from creating an instance of the class, so in this language the usefulness of an abstract class declaration is quite limited.

Sealed Classes and Final Methods

As I mentioned, Java has a very dynamic approach with late binding (or virtual functions) being the default. For this reason the language introduced concepts like classes you cannot inherit from (*sealed*) and methods you cannot override in derived classes (*final methods*, or non-virtual methods).

Sealed classes are classes you cannot further inherit from. This might make sense if you are distributing components (without the source code) or runtime packages and you want to limit the ability of other developers to modify your code. One of the

original goals was also to increase runtime security, something you won't generally need in a fully compiled language like Object Pascal.

Final methods are virtual methods you cannot further override in inherited classes. Again, while they do make sense in Java (where all methods are virtual by default and final methods are significantly optimized) they were adopted in C# where virtual functions are explicitly marked and are much less important. Similarly, they were added to Object Pascal, were they are rarely used.

In terms of syntax, this is the code of a sealed class:

```
type
  TDeriv1 = class sealed (TBase)
    procedure A; override;
  end;
```

Trying to inherit from it causes the error, "*Cannot extend sealed class TDeriv1*". This is the syntax of a final method:

```
type
  TDeriv2 = class (TBase)
    procedure A; override; final;
  end;
```

Inheriting from this class and overriding the `A` method causes the compiler error, "*Cannot override a final method*".

Safe Type Cast Operators

As we have seen earlier, the language type compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned, the reverse is not possible.

Now suppose that the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, you might want to be able to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

To solve the problem, we can use techniques based on run-time type information. Essentially, because each object at run time "knows" its type and its parent class. We can ask for this information with the `is` operator or using some of the methods of

the `TObject` class. The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then
  ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from and compatible with `TDog`. This means that if you test whether a `TDog` object stored in a `TAnimal` variable is really a `TDog` object, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

note The actual implementation of the is operator is provided by the `InheritsFrom` method of the `TObject` class. So you could write the same expression as `MyAnimal.InheritsFrom(TDog)`. The reason to use this method directly comes from the fact that it can also be applied to class references and other special purpose types than don't support the `is` operator.

Now that you know for sure that the animal is a dog, you can use a direct type cast (that would in general be unsafe) by writing the following code:

```
if MyAnimal is TDog then
begin
  MyDog := TDog (MyAnimal);
  Text := MyDog.Eat;
end;
```

This same operation can be accomplished directly by another related type cast operator, `as`, which converts the object only if the requested class is compatible with the actual one. The parameters of the `as` operator are an object and a class type, and the result is an object "*converted*" to the new class type. We can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If we only want to call the `Eat` function, we might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second one checks the actual type of the object and raises an exception if the type is not compatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described in the next chapter).

note By contrast, in the C# language the `as` expression will return `nil` if the object is not type-compatible, while the direct type cast will raise an exception. So basically the two operations are reversed compared to Object Pascal.

To avoid this exception, use the `is` operator and, if it succeeds, make a plain typecast (in fact there is no reason to use `is` and `as` in sequence, doing the type check twice – although you'll often see the combined use of `is` and `as`):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both type cast operators are very useful in Object Pascal because you often want to write generic code that can be used with a number of components of the same type or even of different types. For example, when a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`), so you often need to cast it back to the original component type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

This is a common technique I'll use it in some later examples (events are introduced in Chapter 10).

The two type cast operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism first. Only in special cases, where polymorphism alone cannot be applied, should you try using the type conversion operators to complement it.

note The use of the type cast operators has a slight negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As we have seen, virtual method calls just require a memory lookup, which is much faster.

Visual Form Inheritance

Inheritance is not only used in library classes or for the classes you write, but it's quite pervasive of the entire development environment based around Object Pascal. As we have seen, when you create a form in the IDE, this is an instance of a class that inherits from `TForm`. So any visual application has a structure based on inheritance, even in case you end up writing most your code in simple event handlers.

What is less known, though, even by more experienced developers, is that you can inherit a new form from one you've already created, a feature generally called *visual*

form inheritance (and something quite peculiar to Object Pascal development environment).

The interesting element here is you can visually see the power on inheritance in action, and directly figure out its rules! In this useful also in practice? Well, it mostly depends on the kind of application you are building. If it has a number of forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. Another common scenario is to use visual form inheritance to customize some of the forms of an applications for specific companies, without duplicating any source code (which is the core reason for using inheritance in the first place).

You can also use visual form inheritance to customize an application for different operating systems and form factors (phone to tablets, for example), without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism. You can add a virtual method to a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

note Another approach in building forms with the same elements is to rely on frames, that is on visual composition of form panels. In both cases at design time you can work on two versions of a form. However, in visual form inheritance, you are defining two different classes (parent and derived), whereas with frames, you work on a frame class and an instance of that frame hosted by a form.

Inheriting From a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multiple

properties, you can resynchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To demonstrate how visual form inheritance works, I've built a very simple example, called `VisualInheritTest`. I'll describe step-by-step how to build it. First, start a new mobile project, and add two buttons to its main form. Then select File ➢ New, and choose the "Inheritable Items" page in the New Items dialog box (see Figure 8.3). Here you can choose the form from which you want to inherit.

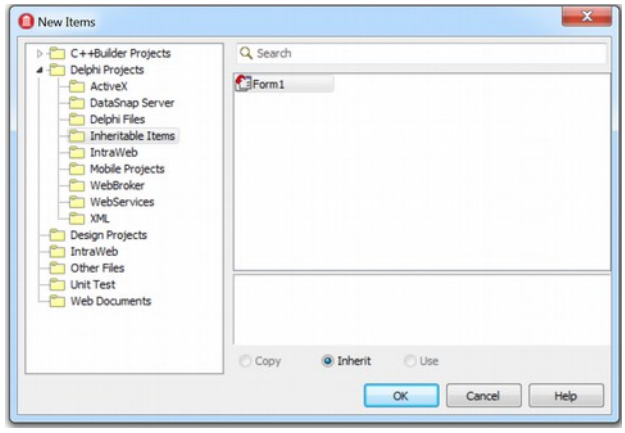


Figure 8.3: The New Items dialog box allows you to create an inherited form.

The new form has the same two buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
end
```

And here is its initial class declaration, where you can see that the base class is not the usual `TForm` but the actual base class form:

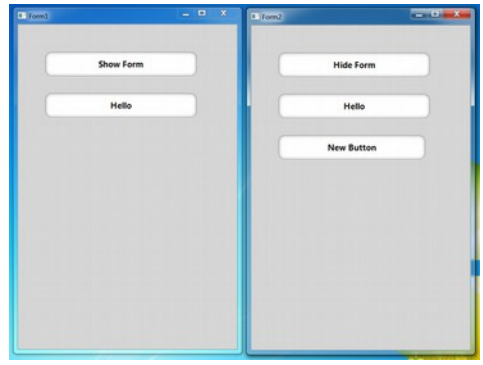
```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Notice the presence of the `inherited` keyword in the textual description; also notice that the form indeed has some components, although they are defined in the base class form. If you change the caption of one of the buttons and add a new button the textual description will change accordingly:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
  inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
    ...
    Text = 'New Button'
    OnClick = Button3Click
  end
end
```

Only the properties with a different value are listed, because the others are simply inherited as they are.

Figure 2.7:
The two forms of the VirtualInheritTest example at run time



Each of the buttons of the first form has an `OnClick` handler, with simple code. The first button shows the second form calling its `Show` method; the second button a simple message.

What happens in the inherited form? First we should change the behavior of the Show button to implement it as a Hide button. This implies not executing the base class event handler (so I've commented out the default `inherited` call). For the Hello button, instead, we can add a second message to the one displayed by the base class, by leaving the `inherited` call:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  // inherited;
```

```
  Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hello from Form2');
end;
```

Remember that differently from an inherited method, that can use the `inherited` keyword to call the base class method with the same name, in an event handler the `inherited` keyword stands for a call to the corresponding event handler of the base form (regardless of the event handler method name).

Of course, you can also consider each method of the base form as a method of your form, and call them freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at more complex real-world examples than this book has room to explore.

09: handling exceptions

Before we proceed with the coverage of other features of classes in the Object Pascal language, we need to focus on one particular group of objects used to handle error conditions, known as *exceptions*.

The idea of *exception handling* is to make programs more robust by adding the capability of handling software or hardware errors (and any other type of error) in a simple and uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by the component libraries. At run time, the system raises exceptions when something goes wrong. If your code has been written properly, it can acknowledge the problem and try to solve it; otherwise, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, the system generally handles it, by displaying a standard error message and trying to continue the program. In the unusual scenario your code is executed

outside of any exception handling block, raising an exception will cause the program to terminate.

The whole mechanism of exception handling in Object Pascal is based on five separate keywords:

- `try` delimits the beginning of a protected block of code
- `except` delimits the end of a protected block of code and introduces the exception-handling code
- `on` marks the individual exception handling statements, tied to specific exceptions, each having the syntax `on exception-type do statement`
- `finally` is used to specify blocks of code that must always be executed, even when exceptions occur
- `raise` is the statement used to trigger an exception and has as parameter an exception object (this operation is called `throw` in other programming languages)

This is a simple comparison table of exception handling keywords in Object Pascal with languages based on the C++ exceptions syntax (like C# and Java):

`try`	`try`
`except on`	`catch`
`finally`	`finally`
`raise`	`throw`

In general terms, using the C++ language terminology, you throw an exception object and catch it by type. This is the same in Object Pascal, where you pass to the `raise` statement an exception object and you receive it as a parameter of the `except on` statements.

Try-Except Blocks

Let me start with a rather simple try-except example (part of the `ExceptionsTest` application project), one that has a generic exception handling block:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // raises exception if B equals 0
    Inc (Result);
  except
    Result := 0;
  end;
  //more
```

```
end;
```

note When you run a program in the debugger, the debugger will stop the program by default when an exception is encountered, even if there is an exception handler. This is normally what you want, of course, because you'll know where the exception took place and can see the call of the handler step-by-step. If you just want to let the program run when the exception is properly handled, and see what a user would see, run the program with the "Run without debugging" command.

Not that "silencing" the exception and setting the result to 0 really makes a lot of sense, but the code is good enough to understand the core mechanism in a reasonable coding scenario. This is the code of the event handler used to call the function:

```
var
  N: Integer;
begin
  N := DividePlusOne (10, Random(3));
  Show (N.ToString);
```

As you can see the program uses a randomly generated value so that when you click the button you can be in a valid situation (2 times out of 3) or in an invalid one. This way there can be two different program flows:

- If B is not zero, the program does the division, executes the increment, and then skips the except block up the end statement following it (`//more`)
- If B is zero, the division raises an exception, all of the following statements are skipped (well, only one in this case) up to the first enclosing try-expect block, which gets executed instead. After the exception block, the program won't get back to the original statement, but skips until after the except block executing the first statement after it (`//more`).

A way to describe this exception model is to say it follows an approach of non-resumption. In case of an error, trying to handle the error condition and getting back to the statement that caused it, is very dangerous, as the status of the program at that point is probably undefined. Exceptions significantly change the execution flow, skipping execution of the following statement and rolling back the stack until the proper error handling code is found.

The code above had a very simple except block, with no `on` statement. When you need to handle multiple types of exceptions (or multiple exception class types) or want to access to the exception object passed to the block, you need to have one or more on statements:

```
function DividePlusOneBis (A, B: Integer): Integer;
begin
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
  except
```

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

In the exception-handling statement, we catch the `EDivByZero` exception, which is defined by the run-time library. There are a number of these exception types referring to run-time problems (such as a division by zero or a wrong dynamic cast), to system problems (such as out-of-memory errors), or to component errors (such as a wrong index). All of these exceptions classes inherit from the base class `Exception`, which offers some minimal features like the `Message` property I used in the code above. These classes form an actual hierarchy with some logical structure.

note Notice that while types in Object Pascal are generally marked with an initial letter T, exception classes take an *exception* to the rule and generally start with the letter E.

The Exceptions Hierarchy

Here is a partial list of the core RTL exception classes defined in the `SysUtils` unit (most of the other system libraries add their own exception types):

```
Exception
  EArgumentException
    EArgumentOutOfRangeException
    EArgumentNilException
  EPathTooLongException
  ENotSupportedException
  EDirectoryNotFoundException
  EFileNotFoundException
  EPathNotFoundException
  EListError
  EInvalidOpException
  ENoConstructException
  EAbort
  EHeapException
    EOutOfMemory
    EInvalidPointer
  EInOutError
  EExternal
    EExternalException
    EIntError
      EDivByZero
      ERangeError
      EIntOverflow
    EMathError
```

```
      EInvalidOp
      EZeroDivide
      EOverflow
      EUnderflow
    EAccessViolation
    EPrivilege
    EControlC
    EQuit
  EInvalidCast
  EConvertError
  ECodesetConversion
  EVariantError
  EPropReadOnly
  EPropWriteOnly
  EAssertionFailed
  EAbstractError
  EIntfCastError
  EInvalidContainer
  EInvalidInsert
  EPackageError
  ECFError
  EOSError
  ESafecallException
  EMonitor
    EMonitorLockException
    ENoMonitorSupportException
  EProgrammerNotFound
  ENotImplemented
  EObjectDisposed
  EJNIException
```

note I don't know about you, but I still have to figure out the exact usage scenario of what I consider the most odd exception class, the `EProgrammerNotFound` exception.

Now that you have seen the core exceptions hierarchy, I can add one piece of information to the previous description of the `except-on` statements. These statements are evaluated in sequence until the system finds an exception class matching the type of the exception object that was raised. Now the matching rule used is the type compatibility rule we examined in the last chapter: an exception object is compatible with any of the base types of its own specific type (like a `TDog` object was compatible with the `TAnimal` class).

This means you can have multiple exception handler types that match the exception. If you want to be able to handle the more granular exceptions (the lower classes of the hierarchy) along with the more generic one in case none of the previous matches, you have to list the handler blocks from the more specific to the more generic (or from the child exception class up to its parent classes). Also, if you write a handler for the type `Exception` it will be a catch-all clause. Here is a code snippet with two handlers in one block:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero error',
        mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
      MessageDlg (E.Message,
        mtError, [mbOK], 0);
    end;
  end; // end of except block
end;
```

In this code there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence as explained above.

Keep in mind that using a handler for every possible exception is not usually a good choice. It is better to leave unknown exceptions to the system. The default exception handler generally displays the error message of the exception class in a message box, and then resumes normal operation of the program.

---

note You can actually modify the normal exception handler by providing a method for the `Application.OnException` event, for example logging the exception message in a file rather than displaying it to the user.

Raising Exceptions

Most exceptions you'll encounter in your Object Pascal programming will be generated by the system, but you can also raise exceptions in your own code when you discover invalid or inconsistent data at run time.

In most cases, for a custom exception you'll define your own exception type. Simply create a new subclass of the default exception class or one of its existing subclasses we saw above:

```
type
  EArrayFull = class (Exception);
```

In most cases, you don't need to add any methods or fields to the new exception class and the declaration of an empty derived class will suffice.

The scenario for this exception type would be a method that adds elements to an array raising an error when the array is full. This is accomplished by creating the exception object and passing it to the `raise` keyword:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This `Create` method (inherited from the base `Exception` class) has a string parameter to describe the exception to the user.

note You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

There is a second scenario for using the raise keyword. Within an except block you might want to perform some actions but don't trap the exception, letting it flow to the enclosing exception handler block. In this case, you can call `raise` with no parameters. The operation is called *re-raising* an exception.

Exceptions and the Stack

When the program raises an exception and the current routine doesn't handle it, what happens to your method and function call stack? The program starts searching for a handler among the functions already on the stack. This means that the program exits from existing functions and does not execute the remaining statements. To understand how this works, you can either use the debugger or add a number of simple output lines, to be informed when a certain source code statement is executed. In the next application project, `ExceptionFlow`, I've followed this second approach.

For example, when you press the `Raise1` button in the form of the `ExceptionFlow` project, an exception is raised and not handled, so that the final part of the code will never be executed:

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  // unguarded call
  AddToArray (24);
  Show ('Program never gets here');
end;
```

Notice that this method calls the `AddToArray` procedure, which invariably raises the exception. When the exception is handled, the flow starts again after the handler and not after the code that raises the exception. Consider this modified method:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
    Show ('Program never gets here');
  except
  on EArrayFull do
      Show ('Handle the exception');
  end;
  Show ('ButtonRaise1Click call completed');
end;
```

The last `Show` call will be executed right after the second one, while the first is always ignored. I suggest that you run the program, change its code, and experiment with it to fully understand the program flow when an exception is raised.

---

note Given the code location where you handle the exception is different than the one the exception was raised, it would be nice to be able to know in which method the exception was actually raised. While there are ways to get a stack trace when the exception is raised and make that information available in the handler, this is really an advanced topic I don't plan to cover here. In most cases, Object Pascal developers rely on third party libraries and tools (like JclDebug from Jedi Compo - nent Libray, MadExcept, or EurekaLog).

The Finally Block

There is a fourth keyword for exception handling that I've mentioned but haven't used so far, `finally`. A `finally` block is used to perform some actions (usually cleanup operations) that should always be executed. In fact, the statements in the `finally` block are processed whether or not an exception takes place. The plain code following a `try` block, instead, is executed only if an exception was not raised or if it was raised and handled. In other words, the code in the `finally` block is always executed after the code of the `try` block, even if an exception has been raised.

Consider this method (part of the `ExceptFinally` application project), which performs some time-consuming operations and shows in the form caption its status:

```
procedure TForm1.btnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
```

```
  Caption := 'Calculating';

  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;

  Caption := 'Finished';
  Show ('Total: ' + J.ToString);
end;
```

Because there is an error in the algorithm (as the variable `I` can reach a value of 0 and is also used in a division), the program will break, but it won't reset the form caption. This is what a `try-finally` block is for:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  finally
    Caption := 'Finished';
  end;
end;
```

When the program executes this function, it always resets the cursor, whether an exception (of any sort) occurs or not. The drawback to this version of the function is that it doesn't handle the exception.

Finally And Except

Curiously enough, in the Object Pascal language a `try` block can be followed by either an `except` or a `finally` statement but not both at the same time. Given you'd often want to have both blocks, the typical solution is to use two nested `try` blocks, associating the internal one with a `finally` statement and the external one with an `except` statement or vice versa, as the situation requires. Here is the code of this third button of the `ExceptFinally` example:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try try
```

```
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  except
    on E: EDivByZero do
    begin
      // re-raise the exception with a new message
      raise Exception.Create ('Error in Algorithm');
    end;
  end;
  finally
    Caption := 'Finished';
  end;
end;
```

Exceptions in the Real World

Exceptions are a great mechanism for error reporting and error handling at large (that is not within a single code fragment, but as part of a larger architecture). Exceptions in general should not be a substitute for checking a local error condition (although some developers use them this way). For example, if you are not sure about a file name, checking if a file exists before opening is generally considered a better approach than opening the file anyway using exceptions to handle the scenario the file is not there. However, checking if there is still enough memory disk space before writing to the file, is a type of check that makes little sense to do all over the places, as that is an extremely rare condition.

One way to put it is that a program should check for common error conditions and leave the unusual and unexpected ones to the exception handling mechanism. Of course, the line between the two scenarios are often blurred, and different developers will have different ways to judge.

Where you'd invariably use exceptions is for letting different classes and modules pass error conditions to each other. Returning error codes is extremely tedious and error prone compared to using exceptions. Raising exceptions is more common in a component or library class than in an event handler. You can end up writing a lot of code without raising on handling exceptions.

What is extremely important and very common in every day code, instead, is using finally blocks to protect resources in case of an exception. You should always protect blocks that refer to external resource with a `finally` statement, to avoid resource leaks in case an exception is raised. Every time you open and close, connect and dis-

connect, create and destroy something in a single block, a finally statement is required.

Ultimately, a finally statement let you keep a program stable even in case an exception is raised, letting the user continue to use or (in case of more significant issues) orderly shut down the application.

Global Exceptions Handling

If an exception raised by an event handler stops the standard flow of execution, will it also terminate the program if no exception handler is found? This is really the case for a console application or other special purpose code structures, while most visual applications (included those based on the VCL or FireMonkey libraries) have a global message handling loop that wraps each execution in a `try-except` block, so that if an exception is raise in an event handler, this is trapped.

What happens at this point depends on the library, but there is a generally a programmatic way to intercept those exceptions with global handlers or a way to display an error message. While some of the details differ, this is true for both VCL and FireMonkey. In the previous demos, you saw a simple error message displayed when an exception was raised.

If you want to change that behavior you can handle the `OnException` event of the global `Application` object. Although this operation pertains more to the visual library and event handling side of the application, it is also tied to the exception handling that it is worth to cover it here.

I've taken the previous example, called it `ErrorLog`, and I've added a new method to the main form:

```
  public
    procedure LogException (Sender: TObject; E: Exception);
```

In the `OnCreate` event handler I've added the code to hook a method to the global `OnException` event, and after that I've written the actual code of the global handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := LogException;
end;

procedure TForm1.LogException(Sender: TObject; E: Exception);
begin
  Show('Exception ' + E.Message);
end;
```

note You'll learn the details of how you can assign a method pointer to an event (like I did above) in the next chapter.

With the new method in the global exceptions handler, the program writes to the output the error message, without stopping the application with an error message.

Exceptions and Constructors

There is a slightly more advanced issue surrounding exceptions, namely what happens when an exception is raised within the constructor of an object. Not all Object Pascal programmers know that in such circumstances the destructor of that object (if available) will be called.

This is important to know, because it implies that a destructor might be called for a partially initialized object. Taking for granted that internal objects exist in a destructor because they are created in the constructor might get you into some dangerous situations in case of actual errors (that is, raising another exception before the first one is handled).

This also implies that the proper sequence for a `try-finally` should involve creating the object outside of the block, as it is automatically protected by the compiler. So if the constructor fails there is no need to `Free` the object. This is why the standard coding style in Object Pascal is to protect an object by writing:

```
AnObject := AClass.Create;
try
  // use the object...
finally
  AnObject.Free;
end;
```

note Something similar also happens for two special methods of the `TObject` class, `AfterDestruction` and `BeforeConstruction`, a pseudo-constructor and a pseudo-destructor introduced for C++ compatibility (but seldom used in Object Pascal). Similarly to what happens with the plain constructor and destructor, if an exception is raised in the `AfterConstruction`, `BeforeDestruction` is called (and also the plain destructor, of course).

Notice that you don't need to use the finally block when using the ARC-enabled Object Pascal compilers, as in that case the release of the object memory is automatic, as I'll detail in Chapter 13. Given I've often witnessed errors in properly disposing of an object in a destructor, let me further clarify the issue with an actual demo showing the problem... along with the actual fix. Suppose you have a class

including a string list, and that you write the following code to create and destroy the class (part of the `ConstructorExcept` code):

```
type
  TObjectWithList = class
  private
    FStringList: TStringList;
  public
    constructor Create (Value: Integer);
    destructor Destroy; override;
  end;

constructor TObjectWithList.Create(Value: Integer);
begin
  if Value < 0 then
    raise Exception.Create('Negative value not allowed');

  FStringList := TStringList.Create;
  FStringList.Add('one');
end;

destructor TObjectWithList.Destroy;
begin
  FStringList.Clear;
  FStringList.Free;
  inherited;
end;
```

At first sight, the code seems correct. The constructor is allocating the sub-object and the destructor is properly disposing it (which is required when using a non ARC-enabled Object Pascal compiler). Moreover, the calling code is written in a way that if an exception is raise after the constructor, the `Free` method is called, but if the exception is in the constructor nothing happens:

```
var
  Obj: TObjectWithList;
begin
  Obj := TObjectWithList.Create (-10);
  try
    // do something
  finally
    Show ('Freeing object');
    Obj.Free;
  end;
```

So does this work? Absolutely not! When this code is involved an exception is raise in the constructor before creating the string list, and the system immediately invokes the destructor, which tries to clear the non-exiting list raising an access violation or a similar error.

Why would this happen? Again, if you reverse the sequence in the constructor (create the string list first, raise the exception later) everything works properly because

the destructor indeed needs to free the string list. But that is not the real fix, only a workaround. What you should always consider is protecting the code of a destructor in a way it never assumes the constructor was completely executed. This could be an example:

```
destructor TObjectWithList.Destroy;
begin
  if Assigned (FStringList) then
  begin
    FStringList.Clear;
    FStringList.Free;
  end;
  inherited;
end;
```

Advanced Exceptions

> *This is one of the sections of the book you might want to skip the first time you read it, as it might be a little too complex. Unless you have already a good knowledge of the language, I suggest moving to the next chapter.*

In the final part of the chapter, I'm going to cover some more advanced topics related with exceptions handling. I'll cover nested exceptions (`RaiseOuterException`) and intercepting exceptions of a class (`RaisingException`). These features were not part of the early versions of Delphi, and add significative power to the system.

Nested Exceptions and the InnerException Mechanism

What happens if you raise an exception within an exception handler? The traditional answer is that the new exception will replace the existing one, which is why it is a common practice to combine at least the error messages, writing code like this (lacking any actual operation, and showing only the exceptions-related statements):

```
procedure TFormExceptions.ClassicReraise;
begin
  try
    // do something...
    raise Exception.Create('Hello');
  except on E: Exception do
    // try some fix...
    raise Exception.Create('Another: ' + E.Message);
```

```
  end;
end;
```

This code is part of the `AdvancedExcept` application project. When calling the method and handling the exception, you'll see a single exception with the combined message:

```
procedure TFormExceptions.btnTraditionalClick(
  Sender: TObject);
begin
  try
    ClassicReraise;
  except
    on E: Exception do
      Show ('Message: ' + E.Message);
  end;
end;
```

The (quite obvious) output is:

```
Message: Another: Hello
```

Now in Object Pascal there is system-wide support for nested exceptions. Within an exception handler, you can create and raise a new exception and still keep the current exception object alive, connecting it to the new exception. To accomplish this, the `Exception` class has an `InnerException` property, referring to the previous exception, and a `BaseException` property that lets you access the first exception of a series, as exception nesting can be recursive. These are the elements of the `Exception` class related to the management of nested exceptions:

```
type
  Exception = class(TObject)
  private
    FInnerException: Exception;
    FAcquireInnerException: Boolean;
  protected
    procedure SetInnerException;
  public
    function GetBaseException: Exception; virtual;
    property BaseException: Exception read GetBaseException;
    property InnerException: Exception read FInnerException;
    class procedure RaiseOuterException(E: Exception); static;
    class procedure ThrowOuterException(E: Exception); static;
  end;
```

note Static class methods are a special form of class methods. Both of these language features will be explained in Chapter 12.

From the perspective of a user, to raise an exception while preserving the existing one you should call the `RaiseOuterException` class method (or the identical `ThrowOuterException`, which uses C++-oriented naming). When you handle a simi-

lar exception you can use the new properties to access further information. Notice that you can call `RaiseOuterException` only within an exception handler as the *source code-based* documentation tells:

> *Use this function to raise an exception instance from within an exception handler and you want to "acquire" the active exception and chain it to the new exception and preserve the context. This will cause the `FInnerException` field to get set with the exception currently in play.*
>
> *You should only call this procedure from within an except block where this new exception is expected to be handled elsewhere.*

For an actual example you can refer to the `AdvancedExcept` application project. In this example I've added a method that raises a nested exception in the new way (compared to the `ClassicReraise` method listed earlier):

```
procedure TFormExceptions.MethodWithNestedException;
begin
  try
    raise Exception.Create ('Hello');
  except
    Exception.RaiseOuterException (
      Exception.Create ('Another'));
  end;
end;
```

Now in the handler for this outer exception we can access both exception objects (and also see the effect of calling the new `ToString` method):

```
  try
    MethodWithNestedException;
  except
    on E: Exception do
    begin
      Show ('Message: ' + E.Message);
      Show ('ToString: ' + E.ToString);
      if Assigned (E.BaseException) then
        Show ('BaseException Message: ' +
          E.BaseException.Message);
      if Assigned (E.InnerException) then
        Show ('InnerException Message: ' +
          E.InnerException.Message);
    end;
  end;
```

The output of this call is the following:

```
Message: Another
ToString: Another
Hello
BaseException Message: Hello
InnerException Message: Hello
```

There are two relevant elements to notice. The first is that in the case of a single nested exception the `BaseException` property and the `InnerException` property both refer to the same exception object, the original one. The second is that while the message of the new exception contains only the actual message, by calling `ToString` you get access to the combined messages of all the nested exceptions, separated by an `sLineBreak` (as you can see in the code of the method `Exception.ToString`). The choice of using a line break in this case produces odd looking output, but once you know about it you can format it the way you like, replacing the line breaks with a symbol of your choice or assigning them to the `Text` property of a string list.

As a further example, let me show you what happens when raising two nested exceptions. This is the new method:

```
procedure TFormExceptions.MethodWithTwoNestedExceptions;
begin
  try
    raise Exception.Create ('Hello');
  except
    begin
      try
        Exception.RaiseOuterException (
          Exception.Create ('Another'));
      except
        Exception.RaiseOuterException (
          Exception.Create ('A third'));
      end;
    end;
  end;
end;
```

This called a method that is identical to the one we saw previously and produces the following output:

```
Message: A third
ToString: A third
Another
Hello
BaseException Message: Hello
InnerException Message: Another
```

This time the `BaseException` property and the `InnerException` property refer to different objects and the output of `ToString` spans three lines.

Intercepting an Exception

Another advanced feature added over time to the exception handling system is the method:

```
procedure RaisingException(P: PExceptionRecord); virtual;
```

According to the source code documentation:

> *This virtual function will be called right before this exception is about to be raised. In the case of an external exception, this is called soon after the object is created since the "raise" condition is already in progress.*

The implementation of the function in the `Exception` class manages the inner exception (by calling the internal `SetInnerException`), which probably explains why it was introduced in the first place, at the same time as the inner exception mechanism.

In any case, now that we have this feature available we can take advantage of it. By overriding this method, in fact, we have a single post-creation function that is invariably called, regardless of the constructor used to create the exception. In other words, you can avoid defining a custom constructor for your exception class and let users call one of the many constructors of the base `Exception` class, and still have custom behavior. As an example, you can log any exception of a given class (or subclass).

This is a custom exception class (defined again in the `AdvancedExcept` example) that overrides the `RaisingException` method:

```
type
  ECustomException = class (Exception)
  protected
    procedure RaisingException(
      P: PExceptionRecord); override;
  end;

procedure ECustomException.
  RaisingException(P: PExceptionRecord);
begin
  // log exception information
  FormExceptions.Show('Exception Addr: ' + IntToHex (
    Integer(P.ExceptionAddress), 8));
  FormExceptions.show('Exception Mess: ' + Message);

  // modify the message
  Message := Message +  ' (filtered)';

  // standard processing
  inherited;
end;
```

What this method implementation does is to log some information about the exception, modify the exception message and then invoke the standard processing of the base classes (needed for the nested exception mechanism to work). The method is invoked after the exception object has been created but before the exception is raised. This can be noticed because the output produced by the `Show` calls is gener-

ated before the exception is caught by the debugger! Similarly, if you put a breakpoint in the `RaisingException` method, the debugger will stop there before catching the exception.

Again, nested exceptions and this intercepting mechanism are not commonly used in application code, as they are language features more meant for library and component developers.

part iii: advanced features

Now that we've delved into the language foundations and into the object-oriented programming paradigm, it is time to discover some of the latest and more advanced features of the Object Pascal language. Generics, anonymous methods, and reflection open up to developing code using new paradigms that extend object-oriented programming in significant ways.

Some of these more advanced language features, in fact, let developers embrace new ways of writing code, offering even more type and code abstractions, and allowing for a more dynamic approach to coding using reflection to its fullest potential.

The last part of the section will expand on these language features by offering an overview of core run-time library elements, which are so core to the Object Pascal development model to make the distinction between language to library quite blurred. We'll inspect, for example, the `TObject` class that, as we saw earlier, is the base class of all classes you write: far to prominent a role to be confined to a library implementation detail.

Chapters of Part III

14: generics

The strong type checking provided by Object Pascal is useful for improving the correctness of the code, a topic I've stressed a lot in this book. Strong type checking, though, can also be a nuisance, as you might want to write a procedure or a class that can act on different data types. This issue is addressed by a feature of the Object Pascal language, available also in similar languages like C# and Java, called *generics*.

The concept of generic or template classes come from the C++ language. This is what I wrote in 1994 in a book about C++:

> *You can declare a class without specifying the type of one or more data members: this operation can be delayed until an object of that class is actually declared. Similarly, you can define a function without specifying the type of one or more of its parameters until the function is called.*

note The book is "Borland C++ 4.0 Object-Oriented Programming", written by me with Steve Tendon in the early 90ies.

This chapter delves into the topic, starting with the foundations but also covering some advanced usage scenarios, and even indicating how generics can even be applied to standard visual programming.

Generic Key-Value Pairs

As a first example of a generic class, I've implemented a key-value pair data structure. The first code snippet below shows the data structure written in a traditional fashion, with an object used to hold the value:

```
type
  TKeyValue = class
  private
    FKey: string;
    FValue: TObject;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: TObject);
  public
    property Key: string read FKey write SetKey;
    property Value: TObject read FValue write SetValue;
  end;
```

To use this class you can create an object, set its key and value, and use it, as in the following snippets of various methods of the main form of the `KeyValueClassic` application project:

```
// FormCreate
kv := TKeyValue.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender;

// Button2Click
kv.Value := self; // the form

// Button3Click
ShowMessage('[' + kv.Key +',' + kv.Value.ClassName + ']');
```

What if you need a similar class, holding an Integer rather than an object? Well, either you make a very unnatural (and dangerous) type cast, or you create a new and separate class to hold a string key with a numeric value. Although copy and paste of the original class might sound a solution, you end up with two copies for a very similar piece of code, you are going against good programming principles... and you'll have to update with new features or correct the same bugs two, or three or twenty times.

Generics make it possible to use a much broader definition for the value, writing a single generic class. Once you instantiate the key-value generic class, it becomes a specific class, tied to a given data type. So you still end up with two, or three, or twenty classes compiled into your application, but you have a single source code definition for all of them, still replying on proper string type checking and without a

runtime overhead. But I'm getting ahead of myself: let's start with the syntax used to define the generic class:

```
type
  TKeyValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
  end;
```

In this class definition, there is one unspecified type, indicated by the placeholder `T`. The symbol `T` is frequently used by convention, but as far as the compiler is concerned you can use just any symbol you like. Using T generally makes the code more readable when the generic class uses only one parametric type; in case the class needs multiple parametric types it is common to name them according to their actual role, rather than using a sequence of letters (`T`, `U`, `V`) as it happened in C++ during the early days.

---

note "T" has been the standard name, or placeholder, for a generic type since the days the C++ language introduced *templates* in the early 1990s. Depending on the authors, the "T" stands for either "Type" or "Template type".

The generic `TKeyValue<T>` class uses the unspecified type as the type of one of its two fields, the property value, and the setter method parameter. The methods are defined as usual, but notice that regardless of the fact they have to do with the generic type, their definition contains the complete name of the class, including the generic type:

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TKeyValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

To use the class, instead, you have to fully qualify it, providing the actual type of the generic type. For example, you can now declare a key-value object hosting buttons as values by writing:

```
kv: TKeyValue<TButton>;
```

The full name is required also when creating an instance, because this is the actual type name (while the generic, uninstantiated type name is like a type construction mechanism).

Using a specific type of the value of the key-value pair makes the code much more robust, as you can now only add `TButton` (or derived) objects to the key-value pair and can use the various methods of the extracted object. These are some snippets from the main form of the `KeyValueGeneric` application project:

```
// FormCreate
kv := TKeyValue<TButton>.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender as TButton;

// Button2Click
kv.Value := Sender as TButton; // was "self"

// Button3Click
ShowMessage ('[' + kv.Key + ',' + kv.Value.Name + ']');
```

When assigning a generic object in the previous version of the code we could add either a button or a form, now we can use only button, a rule enforced by the compiler. Likewise, rather than a generic `kv.Value.ClassName` in the output we can use the component `Name`, or any other property of the `TButton` class.

Of course, we can also mimic the original program by declaring the key-value pair as:

```
kvo: TKeyValue<TObject>;
```

In this version of the generic key-value pair class, we can add any object as value. However, we won't be able to do much on the extracted objects, unless we cast them to a more specific type. To find a good balance, you might want to go for something in between specific buttons and any object, requesting the value to be a component:

```
kvc: TKeyValue<TComponent>;
```

You can see corresponding code snippets in the same `KeyValueGeneric` application project. Finally, we can also create an instance of the generic key-value pair class that doesn't store object values, but rather plain integers:

```
var
  kvi: TKeyValue<Integer>;
begin
  kvi := TKeyValue<Integer>.Create;
  try
    kvi.Key := 'object';
    kvi.Value := 100;
    kvi.Value := Left;
    ShowMessage ('[' + kvi.Key + ',' +
```

```
    IntToStr (kvi.Value) + ']');
finally
  kvi.Free;
end;
```

Type Rules on Generics

When you declare an instance of a generic type, this type gets a specific version, which is enforced by the compiler in all subsequent operations. So if you have a generic class like:

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

as you declare a specific object with a given type, you cannot assign a different type to the `Value` field. Given the following two objects, some of the assignments below (part of the `TypeCompRules` application project) are incorrect:

```
var
  sg1: TSimpleGeneric<string>;
  sg2: TSimpleGeneric<Integer>;
begin
  sg1 := TSimpleGeneric<string>.Create;
  sg2 := TSimpleGeneric<Integer>.Create;

  sg1.Value := 'foo';
  sg1.Value := 10; // Error
  // E2010 Incompatible types: 'string' and 'Integer'

  sg2.Value := 'foo';  // Error
  // E2010 Incompatible types: 'Integer' and 'string'
  sg2.Value := 10;
```

Once you define a specific type in the generic declaration, this is enforced by the compiler, as you should expect from a strongly-typed language like Object Pascal. Type checking is also in place for generic objects as a whole. As you specify the generic parameter for an object, you cannot assign to it a similar generic type based on a different and incompatible type instance. If this seems confusing, an example should help clarifying:

```
sg1 := TSimpleGeneric<Integer>.Create; // Error
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

As we'll see in the section “Generic Types Compatibility Rules” in this peculiar case the type compatibility rule is by structure and not by type name. You cannot assign a different and incompatible type to a generic type once it has been declared.

Generics in Object Pascal

In the previous example we have seen how you can define and use a generic class in Object Pascal. I decided to introduce this feature with an example before delving into the technicalities, which are quite complex and very important at the same time. After covering generics from a language perspective we'll get back to more examples, including the use and definition of generic container classes, one of the main uses of this technique in the language.

We have seen that when you define a class you can add in an extra “parameter” within angle brackets to hold the place of a type to be provided later:

```
type
  TMyClass<T> = class
    ...
  end;
```

The generic type can be used as the type of a field (as I did in the previous example), as the type of a property, as the type of a parameter or return value of a function, and more. Notice that it is not compulsory to use the type for a local field (or array), as there are cases in which the generic type is used only as a result, a parameter, or is not used in the declaration of the class, but only in the definition of some of its methods.

This form of extended or *generic* type declaration is not only available for classes but also for records (that as I covered in Chapter 5, can also have methods, properties, and overloaded operators). A generic class can also have multiple parameterized types, as in following case in which you can specify an input parameter and a return value of a different type for a method:

```
type
  TPWGeneric<TInput,TReturn> = class
  public
    function AnyFunction (Value: TInput): TReturn;
  end;
```

The implementation of generics in Object Pascal, like in other static languages is not based on runtime support. It is handled by the compiler and the linker, leaving almost nothing to the runtime mechanism. Unlike virtual function calls, which are bound at runtime, generic class methods are generated once for each generic type you instantiate, and are generated at compile time! We'll see the possible drawbacks of this approach, but on the positive side it implies that generic classes are as efficient as plain classes, or even more efficient as the need for runtime checks is reduced. Before we look at some of the internals, though, let me focus on some very significant rules which break the traditional Pascal language type compatibility rules.

Generic Types Compatibility Rules

In traditional Pascal and in Object Pascal the core type compatibility rules are based on type name equivalence. In other words, two variables are type compatible only if their type name is the same, regardless of the actual data structure to which they refer.

This is a classic example of type incompatibility with static arrays (part of the `TypeCompRules` application project):

```
type
  TArrayOf10 = array [1..10] of Integer;

procedure TForm30.Button1Click(Sender: TObject);
var
  array1: TArrayOf10;
  array2: TArrayOf10
  array3, array4: array [1..10] of Integer;
begin
  array1 := array2;
  array2 := array3; // Error
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'

  array3 := array4;
  array4 := array1; // Error
  // E2010 Incompatible types: 'Array' and 'TArrayOf10'
end;
```

As you can see in the code above, all four arrays are structurally identical. However, the compiler will let you assign only those that are type compatible, either because their type has the same explicit name (like `TArrayOf10`) or because they have the same implicit (or compiler generated, type name, as the two arrays declared in a single statement.

This type compatibility rule has very limited exceptions, like those related to derived classes. Another exception to the rule, and a very significant one, is type compatibility for generic types, which is probably also used internally by the compiler to determine when to *generate* a new type from the generic one, with all of its methods.

The new rule states that generic types are compatible when they share the same generic class definition and instance type, regardless of the type name associated with this definition. In other words, the full name of the generic type instance is a combination of the generic type and the instance type.

In the following example the four variables are all type compatible:

```
type
  TGenericArray<T> = class
    anArray: array [1..10] of T;
```

```
  end;

  TIntGenericArray = TGenericArray<Integer>;

procedure TForm30.Button2Click(Sender: TObject);
var
  array1: TIntGenericArray;
  array2: TIntGenericArray;
  array3, array4: TGenericArray<Integer>;
begin
  array1 := TIntGenericArray.Create;
  array2 := array1;
  array3 := array2;
  array4 := array3;
  array1 := array4;
end;
```

Generic Methods for Standard Classes

While the use of generics types to define classes is likely the most common scenario, generic types can also be used in non-generic classes. In other words, a regular class can have a generic method. In this case, you don't specific a specific type for the generic placeholder when you create an instance of the class, but also when you invoke the method. Here is an example class with a generic method from the `GenericMethod` application project:

```
type
  TGenericFunction = class
  public
    function WithParam <T> (t1: T): string;
  end;
```

note When I first wrote this code, probably with a reminiscence of my C++ days, I wrote the parameter as `(t: T)`. Needless to say in a case insensitive language like Object Pascal, this is not a great idea. The compiler will actually let it go but issue errors every time you refer to the generic type T.

There isn't much you can do inside a similar class method (at least unless you use constraints, covered later in this chapter), so I wrote some code using special generic type functions (again covered later) and a special function to convert the type to a string, which it is not relevant to discuss here:

```
function TGenericFunction.WithParam<T>(t1: T): string;
begin
  Result := GetTypeName (TypeInfo (T));
end;
```

As you can see this method doesn't even use the actual value passed as parameter, but only grabs some type information. Again, not knowing at all `t1` type makes it fairly complex to use it in code.

You can call various versions of this “global generic function” as follows:

```
var
  gf: TGenericFunction;
begin
  gf := TGenericFunction.Create;
  try
    Show (gf.WithParam<string>('foo'));
    Show (gf.WithParam<Integer> (122));
    Show (gf.WithParam('hello'));
    Show (gf.WithParam (122));
    Show (gf.WithParam(Button1));
    Show (gf.WithParam<TObject>(Button1));
  finally
    gf.Free;
  end;
```

All of the calls above are correct, as the parametric type can be implicit in these calls. Notice the generic type is displayed (as specified or inferred) and not the actual type of the parameter, which explains this output:

```
string
Integer
string
ShortInt
TButton
TObject
```

If you call the method without indicating the type between angle brackets, the actual type is inferred from the parameter's type. If you call the method with a type and a parameter, the parameter's type must match the generic type declaration. So the three lines below won't compile:

```
    Show (gf.WithParam<Integer>('foo'));
    Show (gf.WithParam<string> (122));
    Show (gf.WithParam<TButton>(self));
```

Generic Type Instantiation

> *Notice this is a rather advanced section focusing on some of the internals of generics and their potential optimization. Good for a second read, not if this is the first time you are looking into generics.*

With the exception of some optimizations, every time you instantiate a generic type, whether in a method or in a class, a new type is generated by the compiler. This new

type shares no code with different instances of the same generic type (or different versions of the same method).

Let's look at an example (which is part of the `GenericCodeGen` application project). The program has a generic class defined as:

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (value: T);
  end;
```

The three methods are implemented as follows (notice that the `One` method is absolutely independent from the generic type):

```
procedure TSampleClass<T>.One;
begin
  Form30.Show ('OneT');
end;

function TSampleClass<T>.ReadT: T;
begin
  Result := data;
end;

procedure TSampleClass<T>.SetT(value: T);
begin
  data := value;
end;
```

Now the main program uses the generic type mostly to figure out the in-memory address of its methods once an instance is generated (by the compiler). This is the code

```
procedure TForm30.Button1Click(Sender: TObject);
var
  t1: TSampleClass<Integer>;
  t2: TSampleClass<string>;
begin
  t1 := TSampleClass<Integer>.Create;
  t1.SetT (10);
  t1.One;

  t2 := TSampleClass<string>.Create;
  t2.SetT ('hello');
  t2.One;

  Show ('t1.SetT: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.SetT)^, 8));
  Show ('t2.SetT: ' +
```

```
    IntToHex (PInteger(@TSampleClass<string>.SetT)^, 8));

  Show ('t1.One: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.One)^, 8));
  Show ('t2.One: ' +
    IntToHex (PInteger(@TSampleClass<string>.One)^, 8));
end;
```

The result is something like this (the actual values will vary):

```
t1.SetT: C3045089
t2.SetT: 51EC8B55
t1.One: 4657F0BA
t2.One: 46581CBA
```

As I anticipated, not only does the `SetT` method get a different version in memory generated by the compiler for each data type used, but even the `One` method does, despite the fact they are all identical.

Moreover, if you redeclare an identical generic type, you'll get a new set of implementation functions. Similarly, the same instance of a generic type used in different units forces the compiler to generate the same code over and over, possibly causing significant code bloat. For this reason if you have a generic class with many methods that don't depend on the generic type, it is recommended to define a base non-generic class with those common methods and an inherited generic class with the generic methods: this way the base class methods are only compiled and included in the executable once.

---

note There is currently compiler, linker, and low-level RTL work being done to reduce the size increase caused by generics in scenarios like those outlined in this section. See for example the considera - tions in http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html.

Generic Type Functions

The biggest problem with the generic type definitions we have seen so far is that there is very little you can do with elements of the generic class type. There are two techniques you can use to overcome this limitation. The first is to make use of the few special functions of the runtime library that specifically support generic types; the second (and much more powerful) is to define generic classes with constraints on the types you can use.

I'll focus on the first technique in this section and on constraints in the next section. As I mentioned, there are some RTL functions that work on the parametric type (`T`) of generic type definition:

`Default (T)` is a actually a new function introduced along with generics that returns the empty or "zero value" or null value for the current type; this can be zero, an empty string, `nil`, and so on; the zero-initialized memory has the same value of a global variable of the same type (differently from local variables, in fact, global ones are initialized to "zero" by the compiler);

`TypeInfo (T)` returns the pointer to the runtime information for the current version of the generic type; you'll find a lot more information about type information in Chapter 16;

`SizeOf (T)` returns memory size of the type in bytes (which in case of a reference type like a string or an object would be the size of the reference, that is 4 bytes for a 32-bit compiler and 8 bytes for a 64-bit compiler).

`IsManagedType(T)` indicates is the type is managed in memory, as happens for strings and dynamic arrays

`HasWeakRef(T)` is tied to ARC-enabled compilers, and indicates whether the target type has weak references, requiring specific memory management support

`GetTypeKind(T)` is a shortcut for accessing the type kind from the type information; which is a slightly higher level type definition than the one returned by `TypeInfo`.

note All of these methods return compiler evaluated constants rather than calling actual functions at runtime. The importance of this is not in the fact these operations are very fast, but that this makes it possible for the compiler and the linker to optimize the generated code, removing unused branches. If you have a case or an if statement based on the return value of one of these functions, the compiler can figure out that for a given type only one of the branches is going to be executed, removing the useless code. When the same generic method is compiled for a different type, it might end up using a different branch, but again the compiler can figure out up front and optimize the size of the method.

The `GenericTypeFunc` application project has a generic class showing the three generic type functions in action:

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
end;
```

```
function TSampleClass<T>.GetDataName: string;
begin
  Result := GetTypeName (TypeInfo (T));
end;

procedure TSampleClass<T>.Zero;
begin
  data := Default (T);
end;
```

In the `GetDataName` method I used the `GetTypeName` function (of the `TypInfo` unit) rather than directly accessing the data structure because it performs the proper conversion from the encoded string value holding the type name.

Given the declaration above, you can compile the following test code, that repeats itself three times on three different generic type instances. I've omitted the repeated code, but show the statements used to access the `data` field, as they change depending on the actual type:

```
var
  t1: TSampleClass<Integer>;
  t2: TSampleClass<string>;
  t3: TSampleClass<double>;
begin
  t1 := TSampleClass<Integer>.Create;
  t1.Zero;
  Show ('TSampleClass<Integer>');
  Show ('data: ' + IntToStr (t1.data));
  Show ('type: ' + t1.GetDataName);
  Show ('size: ' + IntToStr (t1.GetDataSize));

  t2 := TSampleClass<string>.Create;
  ...
  Show ('data: ' + t2.data);

  t3 := TSampleClass<double>.Create;
  ...
  Show ('data: ' + FloatToStr (t3.data));
```

Running this code (from the `GenericTypeFunc` application project) produces the following output:

```
TSampleClass<Integer>
data: 0
type: Integer
size: 4
TSampleClass<string>
data:
type: string
size: 4
TSampleClass<double>
data: 0
type: Double
size: 8
```

Notice that you can use the generic type functions also on specific types, outside of the context of generic classes. For example, you can write:

```
var
  I: Integer;
  s: string;
begin
  I := Default (Integer);
  Show ('Default Integer': + IntToStr (I));

  s := Default (string);
  Show ('Default String': + s);

  Show ('TypeInfo String': +
    GetTypeName (TypeInfo (string));
```

This is the trivial output:

```
Default Integer: 0
Default String:
TypeInfo String: string
```

note You cannot apply the `TypeInfo` call to a variable, like `TypeInfo(s)` in the code above, but only to a type.

Class Constructors for Generic Classes

A very interesting case arises when you define a class constructor for a generic class. In fact, one such constructor is generated by the compiler and called for each generic class instance, that is, for each actual type defined using the generic template. This is quite interesting, because it would be very complex to execute initialization code for each actual instance of the generic class you are going to create in your program without a class constructor.

As an example, consider a generic class with some class data. You'll get an instance of this class data for each generic class instance. If you need to assign an initial value to this class data, you cannot use the unit initialization code, as in the unit defining the generic class you don't know which actual classes you are going to need.

The following is a bare bones example of a generic class with a class constructor used to initialize the `DataSize` class field, taken from the `GenericClassCtor` application project:

```
type
  TGenericWithClassCtor <T> = class
  private
    FData: T;
    procedure SetData(const Value: T);
```

```
  public
    class constructor Create;
    property Data: T read FData write SetData;
    class var
      DataSize: Integer;
  end;
```

This is the code of the generic class constructor, which uses an internal string list (see the full source code for implementation details) for keeping track of which class constructors are actually called:

```
class constructor TGenericWithClassCtor<T>.Create;
begin
  DataSize := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

The demo program creates and uses a couple of instances of the generic class, and also declares the data type for a third, which is removed by the linker:

```
var
  genInt: TGenericWithClassCtor <SmallInt>;
  genStr: TGenericWithClassCtor <string>;
type
  TGenDouble = TGenericWithClassCtor <Double>;
```

If you ask the program to show the contents of the `ListSequence` string list, you'll see only the types that have actually been initialized:

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

However, if you create generic instances based on the same data type in different units, the linker might not work as expected and you'll have multiple calls to the same generic class constructor (or, to be more precise, two generic class constructors for the same type).

---

note It is not easy to address a similar problem. To avoid a repeated initialization, you might want to check if the class constructor has already been executed. In general, though, this problem is part of a more comprehensive limitation of generic classes and the linkers inability to optimize them.

I've added a procedure called `Useless` in the secondary unit of this example that, when uncommented, will highlight the problem, with an initialization sequence like:

```
TGenericWithClassCtor<System.string>
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

Generic Constraints

As we have seen, there is very little you can do in the methods of your generic class over the generic type value. You can pass it around (that is, assign it) and perform the limited operations allowed by the generic type functions I've just covered.

To be able to perform some actual operations of the generic type of class, you generally have to place a constraint on it. For example, if you limit the generic type to be a class, the compiler will let you call all of the `TObject` methods on it. You can also further constrain the class to be part of a given hierarchy or to implement a specific interface, making it possible to call the class or interface method on an instance of the generic type.

Class Constraints

The simplest constraint you can adopt is a class constraint. To use it, you can declare generic type as:

```
type
  TSampleClass <T: class> = class
```

By specifying a class constraint you indicate that you can use only object types as generic types. With the following declaration (taken from the `ClassConstraint` application project):

```
type
  TSampleClass <T: class> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (t: T);
  end;
```

you can create the first two instances but not the third:

```
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TStrings>;
  sample3: TSampleClass<Integer>; // Error
```

The compiler error caused by this last declaration would be:

```
E2511 Type parameter 'T' must be a class type
```

What's the advantage of indicating this constraint? In the generic class methods you can now call any `TObject` method, including virtual ones! This is the `One` method of the `TSampleClass` generic class:

```
procedure TSampleClass<T>.One;
begin
  if Assigned (data) then
  begin
    Form30.Show ('ClassName: ' + data.ClassName);
    Form30.Show ('Size: ' + IntToStr (data.InstanceSize));
    Form30.Show ('ToString: ' + data.ToString);
  end;
end;
```

note Two comments here. The first is that `InstanceSize` returns the actual size of the object, unlike the generic `SizeOf` function we used earlier, which returns the size of the reference type. Second, notice the use of the `ToString` method of the `TObject` class.

You can play with the program to see its actual effect, as it defines and uses a few instances of the generic type, as in the following code snippet:

```
var
  sample1: TSampleClass<TButton>;
begin
  sample1 := TSampleClass<TButton>.Create;
  try
    sample1.SetT (Sender as TButton);
    sample1.One;
  finally
    sample1.Free;
  end;
```

Notice that by declaring a class with a customized `ToString` method, this version will get called when the data object is of the specific type, regardless of the actual type provided to the generic type. In other words, if you have a `TButton` descendant like:

```
type
  TMyButton = class (TButton)
  public
    function ToString: string; override;
  end;
```

You can pass this object as value of a `TSampleClass<TButton>` or define a specific instance of the generic type, and in both cases calling `One` ends up executing the specific version of `ToString`:

```
var
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TMyButton>;
  mb: TMyButton;
begin
  ...
  sample1.SetT (mb);
  sample1.One;
  sample2.SetT (mb);
```

```
  sample2.One;
```

Similarly to a class constraint, you can have a record constraint, declared as:

```
type
  TSampleRec <T: record> = class
```

However, there is very little that different records have in common (there is no common ancestor), so this declaration is somewhat limited.

Specific Class Constraints

If your generic class needs to work with a specific subset of classes (a specific hierarchy), you might want to resort to specifying a constraint based on a given base class. For example, if you declare:

```
type
  TCompClass <T: TComponent> = class
```

instances of this generic class can be applied only to component classes, that is, any `TComponent` descendant class. This let's you have a very specific generic type (yes, it sounds odd, but that's what it really is) and the compiler will let you use all of the methods of the `TComponent` class while working on the generic type.

If this seems extremely powerful, think twice. If you consider what you can achieve with inheritance and type compatibly rules, you might be able to address the same problem using traditional object-oriented techniques rather than having to use generic classes. I'm not saying that a specific class constraint is never useful, but it is certainly not as powerful as a higher-level class constraint or (something I find very interesting) an interface-based constraint.

Interface Constraints

Rather than constraining a generic class to a given class, it is generally more flexible to accept as type parameter only classes implementing a given interface. This makes it possible to call the interface on instances of the generic type. This use of interface constraints for generics is also very common in the C# language. Let me start by showing you an example (from the `IntfConstraint` application project). First, we need to declare an interface:

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
```

```
    property Value: Integer
      read GetValue write SetValue;
  end;
```

Next, we can define a class implementing it:

```
type
  TGetValue = class (TSingletonImplementation, IGetValue)
  private
    fValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  end;
```

Things start to get interesting in the definition of a generic class limited to types that implement the given interface:

```
type
  TInftClass <T: IGetValue> = class
  private
    val1, val2: T; // or IGetValue
  public
    procedure Set1 (val: T);
    procedure Set2 (val: T);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

Notice that in the code of the generic methods of this class we can write, for example:

```
function TInftClass<T>.GetMin: Integer;
begin
  Result := min (val1.GetValue, val2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
  val1.SetValue (val1.GetValue + 10);
  val2.Value := val2.Value + 10;
end;
```

With all these definitions, we can now use the generic class as follows:

```
procedure TFormIntfConstraint.btnValueClick(
  Sender: TObject);
var
  iClass: TInftClass<TGetValue>;
begin
  iClass := TInftClass<TGetValue>.Create;
  try
    iClass.Set1 (TGetValue.Create (5));
    iClass.Set2 (TGetValue.Create (25));
```

```
    Show ('Average: ' + IntToStr (iClass.GetAverage));
    iClass.IncreaseByTen;
    Show ('Min: ' + IntToStr (iClass.GetMin));
  finally
    iClass.val1.Free;
    iClass.val2.Free;
    iClass.Free;
  end;
end;
```

To show the flexibility of this generic class, I've created another totally different implementation for the interface:

```
  TButtonValue = class (TButton, IGetValue)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;

function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
  Left := Value;
end;
```

The class function (not listed in the book) creates a button within a `Parent` control in a random position and is used in the following sample code:

```
procedure TFormIntfConstraint.btnValueButtonClick(
  Sender: TObject);
var
  iClass: TInftClass<TButtonValue>;
begin
  iClass := TInftClass<TButtonValue>.Create;
  try
    iClass.Set1 (TButtonValue.MakeTButtonValue (
      self, ScrollBox1));
    iClass.Set2 (TButtonValue.MakeTButtonValue (
      self, ScrollBox1));
    Show ('Average: ' + IntToStr (iClass.GetAverage));
    Show ('Min: ' + IntToStr (iClass.GetMin));
    iClass.IncreaseByTen;
    Show ('New Average: ' + IntToStr (iClass.GetAverage));
  finally
    iClass.Free;
  end;
end;
```

Interface References vs. Generic Interface Constraints

In the last example I have defined a generic class that works with any object implementing a given interface. I could have obtained a similar effect by creating a standard (non-generic) class based on interface references. In fact, I could have defined a class like (again part of the `IntfConstraint` application project):

```
type
  TPlainInftClass = class
  private
    val1, val2: IGetValue;
  public
    procedure Set1 (val: IGetValue);
    procedure Set2 (val: IGetValue);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

What is different between these two approaches? A first difference is that in the class above you can pass two objects of different types to the setter methods, provided their classes both implement the given interface, while in the generic version you can pass only objects of the given type (to any given instance of the generic class). So the generic version is more *conservative* and strict in terms of type checking.

In my opinion, the key difference is that using the interface-based version means having Object Pascal's reference counting mechanism in action, while using the generic version the class is dealing with plain objects of a given type and reference counting is not involved. Moreover, the generic version could have multiple constraints, like a constructor constraint and lets you use the various generic-functions (like asking for the actual type of the generic type), something you cannot do when using an interface. (When you are working with an interface, in fact, you have no access to the base `TObject` methods).

In other words, using a generic class with an interface constraint makes it possible to have the benefits of interfaces without their nuisances. Still, it is worth noticing that in most cases the two approaches would be equivalent, and in others the interface-based solution would be more flexible.

Default Constructor Constraint

There is another possible generic type constraint, called default constructor or parameterless constructor. If you need to invoke the default constructor to create a

new object of the generic type (for example for filling a list) you can use this constraint. In theory (and according to the documentation), the compiler should let you use it only for those types with a default constructor. In practice, if a default constructor doesn't exist, the compiler will let it go and call the default constructor of `TObject`.

A generic class with a constructor constraint can be written as follows (this one is extracted by the `IntfConstraint` application project):

```
type
  TConstrClass <T: class, constructor> = class
  private
    val: T;
  public
    constructor Create;
    function Get: T;
  end;
```

note You can also specify the constructor constraint without the class constraint, as the former probably implies the latter. Listing both of them makes the code more readable.

Given this declaration, you can use the constructor to create a generic internal object, without knowing its actual type up front, and write:

```
constructor TConstrClass<T>.Create;
begin
  val := T.Create;
end;
```

How can we use this generic class and what are the actual rules? In the next example I have defined two classes, one with a default (parameterless) constructor, the second with a single constructor having one parameter:

```
type
  TSimpleConst = class
  public
    Value: Integer;
    constructor Create;  // set Value to 10
  end;

  TParamConst = class
  public
    Value: Integer;
    constructor Create (I: Integer); // set Value to I
  end;
```

As I mentioned earlier, in theory you should only be able to use the first class, while in practice you can use both:

```
var
  constructObj: TConstrClass<TSimpleCost>;
  paramCostObj: TConstrClass<TParamCost>;
```

```
begin
  constructObj := TConstrClass<TSimpleCost>.Create;
  Show ('Value 1: ' + IntToStr (constructObj.Get.Value));

  paramCostObj := TConstrClass<TParamCost>.Create;
  Show ('Value 2: ' + IntToStr (paramCostObj.Get.Value));
```

The output of this code is:

```
Value 1: 10
Value 2: 0
```

In fact, the second object is never initialized. If you debug the application trace into the code you'll see a call to `TObject.Create` (which I consider wrong). Notice that if you try calling directly:

```
  with TParamConst.Create do
```

the compiler will (correctly) raise the error:

```
[DCC Error] E2035 Not enough actual parameters
```

note Even if a direct call to `TParamConst.Create` will fail at compile time (as explained here), a similar call using a class reference or any other form of indirection will succeed, which probably explains the behavior of the effect of the constructor constraint.

Generic Constraints Summary and Combining Them

As there are so many different constraints you can put on a generic type, let me provide a short summary here, in code terms:

```
type
  TSampleClass <T: class> = class
  TSampleRec <T: record> = class
  TCompClass <T: TButton> = class
  TInftClass <T: IGetValue> = class
  TConstrClass <T: constructor> = class
```

What you might not immediately realize after looking at constraints (and this certainly took me some time to get used to) is that you can combine them. For example, you can define a generic class limited to a sub-hierarchy and requiring also a given interface, like in:

```
type
  TInftComp <T: TComponent, IGetValue> = class
    ...
  end;
```

Not all combinations make sense: for example you cannot specify both a class and a record, while using a class constraint combined with a specific class constraint would be redundant. Finally, notice that there is nothing like a method constraint, something that can be achieved with a single-method interface constraint (much more complex to express, though).

Predefined Generic Containers

Since the early days of templates in the C++ Language, one of the most obvious uses of template classes has been the definition of template containers or lists, up to the point that the C++ language defined a Standard Template Library (or STL).

When you define a list of objects, like Object Pascal's own `TObjectList`, you have a list that can potentially hold objects of any kind. Using either inheritance or composition you can indeed define custom containers for specific a type, but this is a tedious (and potentially error-prone) approach.

Object Pascal compilers come with a small set of generic container classes you can find in the `Generics.Collections` unit. The four core container classes are all implemented in an independent way (the is no inheritance among these classes), all implemented in a similar fashion (using a dynamic array), and are all mapped to the corresponding non-generic container class of the older `Contnrs` unit:

```
type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey,TValue> = class
  TObjectList<T: class>  = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
  TObjectDictionary<TKey,TValue> = class(TDictionary<TKey,TValue>)
```

The logical difference among these classes should be quite obvious considering their names. A good way to test them, is to figure out how many changes you have to perform on existing code that uses a non-generic container class.

---

note The program uses only a few methods, so it is not a great test for interface compatibility between generic and non-generic lists, but I decided to take an existing program rather than fabricating one. Another reason for showing this demo, is that you might also have existing programs that don't use generic collection classes and will be encouraged to enhance them by taking advantage of this language feature.

Using TList<T>

The program, called `ListDemoMd2005`, has a unit defining a `TDate` class, and the main form used to refer to a `TList` of dates. As a starting point, I added a uses clause referring to `Generics.Collections`, then I changed the declaration of the main form field to:

```
  private
    ListDate: TList <TDate>;
```

Of course, the main form `OnCreate` event handler that does create the list needed to be updated as well, becoming:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListDate := TList<TDate>.Create;
end;
```

Now we can try to compile the rest of the code as it is. The program has a "wanted" bug, trying to add a `TButton` object to the list. The corresponding code used to compile and now fails:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add (Sender);  // Error:
    // E2010 Incompatible types: 'TDate' and 'TObject'
end;
```

The new list of dates is more robust in terms of type-checking than the original generic list pointers. Having removed that line, the program compiles and works. Still, it can be improved.

This is the original code used to display all of the dates of the list in a ListBox control:

```
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (
      (TObject(ListDate [I]) as TDate).Text);
```

Notice the type cast, due to the fact that the program was using a list of pointers (`TList`), and not a list of objects (`TObjectList`). We can easily improve the program by writing:

```
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (ListDate [I].Text);
```

Another improvement to this snippet can come from using an enumeration (something the predefined generic lists fully support) rather than a plain `for` loop:

```
var
  aDate: TDate;
begin
  for aDate in ListDate do
  begin
    Listbox1.Items.Add (aDate.Text);
  end;
```

Finally, the program can be improved by using a generic `TObjectList` owning the `TDate` objects, but that's a topic for the next section.

As I mentioned earlier, the `TList<T>` generic class has a high degree of compatibility. It has all the classic methods, like `Add`, `Insert`, `Remove`, and `IndexOf`. The `Capacity` and `Count` properties are there as well. Oddly, `Items` become `Item`, but being the default property (accessed by using the square brackets without the property name) you seldom explicitly refer to it anyway.

Sorting a TList<T>

What is interesting to understand is how sorting works (my goal here is to add sorting support to the `ListDemoMd2005` example). The `Sort` method is defined as:

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

where the `IComparer<T>` interface is declared in the `Generics.Defaults` unit. If you call the first version the program, it will use the default comparer, initialized by the default constructor of `TList<T>`. In our case this will be useless.

What we need to do, instead, is to define a proper implementation of the `IComparer<T>` interface. For type compatibility, we need to define an implementation that works on the specific `TDate` class. There are multiple ways to accomplish this, including using anonymous methods (covered in the next section even though that's a topic introduced in the next chapter). An interesting technique, also because it gives me the opportunity to show several usage patterns of generics, is to take advantage of a *structural* class that is part of the unit `Generics.Defaults` and is called `TComparer`.

note I'm calling this class *structural* because it helps defining the structure of the code, but doesn't add a lot in terms of actual implementation. There might be a better name, though.

The class is defined as an abstract and generic implementation of the interface, as follows:

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>;
    class function Construct(
      const Comparison: TComparison<T>): IComparer<T>;
    function Compare(
      const Left, Right: T): Integer; virtual; abstract;
  end;
```

What we have to do is instantiate this generic class for the specific data type (`TDate`, in the example) and also inherit a concrete class that implements the `Compare` method for the specific type. The two operations can be done at once, using a coding idiom that may take a while to digest:

```
type
  TDateComparer = class (TComparer<TDate>)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

If you think this code looks very unusual, you're not alone. The new class inherits from a specific instance of the generic class, something you could express in two separate steps as:

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class (TAnyDateComparer)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

note Having the two separate declarations might help reduce the generated code where you are reusing the base `TAnyDateComparer` type in the same unit.

You can find the actual implementation of the `Compare` function in the source code, as that's not the key point I want to stress here. Keep in mind, though, that even if you sort the list its `IndexOf` method won't take advantage of it (unlike the `TStringList` class).

Sorting with an Anonymous Method

The sorting code presented in the previous section looks quite complicated and it really is. It would be much easier and cleaner to pass the sorting function to the `Sort` method directly. In the past this was generally achieved by passing a function pointer. In Object Pascal this can be done by passing an anonymous method (a kind of method pointer, with several extra features, covered in detail in the next chapter).

note I suggest you have a look at this section even if you don't know much about anonymous methods, and then read it again after going through the next chapter.

The `IComparer<T>` parameter of the `Sort` method of the `TList<T>` class, in fact, can be used by calling the `Construct` method of `TComparer<T>`, passing an anonymous method as a parameter defined as:

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

In practice you can write a type-compatible function and pass it as parameter:

```
function DoCompare (const Left, Right: TDate): Integer;
var
  ldate, rDate: TDateTime;
begin
  lDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  rDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if lDate = rDate then
    Result := 0
  else if lDate < rDate then
    Result := -1
  else
    Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

note The `DoCompare` method above works like an anonymous method even if it does have a name. We'll see in a later code snippet that this is not required, though. Have patience until the next chapter for more information about this Object Pascal language construct. Notice also that with a `TDate` record I could have defined less than and greater then operators, making this code simpler, but even with a class I could have placed the comparison code in a method of the class.

If this looks quite traditional, consider you could have avoided the declaration of a separate function and pass it (its source code) as parameter to the `Construct` method, as follows:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (
    function (const Left, Right: TDate): Integer
    var
      ldate, rDate: TDateTime;
    begin
      lDate := EncodeDate(Left.Year,
        Left.Month, Left.Day);
```

```
      rDate := EncodeDate(Right.Year,
        Right.Month, Right.Day);
      if lDate = rDate then
        Result := 0
      else if lDate < rDate then
        Result := -1
      else
        Result := 1;
    end));
end;
```

This example should have whet your appetite for learning more about anonymous methods! For sure, this last version is much simpler to write than the original comparison covered in the previous section, although for many Object Pascal developers having a derived class might look cleaner and be easier to understand (the inherited version separates the logic better, making potential code reuse easier, but many times you won't make use of it anyway).

Object Containers

Beside the generic classes covered at the beginning of this section, there are also four inherited generic classes that are derived from the base classes defined in the `Generics.Collections` unit, mimicking existing classes of the `Contnrs` unit:

```
type
  TObjectList<T: class> = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
```

Compared to their base classes, there are two key differences. One is that these generic types can be used only for objects; the second is that they define a customized `Notification` method, that in the case when an object is removed from the list (beside optionally calling the `OnNotify` event handler), will `Free` the object.

In other words, the `TObjectList<T>` class behaves like its non-generic counterpart when the `OwnsObjects` property is set. If you are wondering why this is not an option any more, consider that `TList<T>` can now be used directly to work with object types, unlike its non-generic counterpart.

There is also a fourth class, again, called `TObjectDictionary<TKey, TValue>`, which is defined in a different way, as it can own the key object, the value objects, or both of them. See the `TDictionaryOwnerships` set and the class constructor for more details.

Using a Generic Dictionary

Of all the predefined generic container classes, the one probably worth more detailed study is the generic dictionary, `TObjectDictionary<TKey, Tvalue>`.

note *Dictionary* in this case means a collection of elements each with a (unique) key value referring to it. (It is also known as an associative array.) In a classic dictionary you have words acting as keys for their definitions, but in programming terms the key doesn't have to be a string (even if this is a rather frequent case).

Other classes are just as important, but they seem to be easier to use and understand. As an example of using a dictionary, I've written an application that fetches data from a database table, creates an object for each record, and uses a composite index with a customer ID and a description as key. The reason for this separation is that a similar architecture can easily be used to create a proxy, in which the key takes the place of a light version of the actual object *loaded* from the database.

These are the two classes used by the `CustomerDictionary` application project for the key and the actual value. The first has only two relevant fields of the corresponding database table, while the second has the complete data structure (I've omitted the private fields, getter methods, and setter methods):

```
type
  TCustomerKey = class
  private
    ...
  published
    property CustNo: Double
      read FCustNo write SetCustNo;
    property Company: string
      read FCompany write SetCompany;
  end;

  TCustomer = class
  private
    ..
    procedure Init;
    procedure EnforceInit;
  public
    constructor Create (aCustKey: TCustomerKey);
    property CustKey: TCustomerKey
      read FCustKey write SetCustKey;
  published
    property CustNo: Double
      read GetCustNo write SetCustNo;
    property Company: string
      read GetCompany write SetCompany;
    property Addr1: string
```

```
      read GetAddr1 write SetAddr1;
    property City: string
      read GetCity write SetCity;
    property State: string
      read GetState write SetState;
    property Zip: string
      read GetZip write SetZip;
    property Country: string
      read GetCountry write SetCountry;
    property Phone: string
      read GetPhone write SetPhone;
    property FAX: string
      read GetFAX write SetFAX;
    property Contact: string
      read GetContact write SetContact;
  class var
    RefDataSet: TDataSet;
  end;
```

While the first class is very simple (each object is initialized when it is created), the `TCustomer` class uses a *lazy initialization* (or *proxy*) model and keeps around a reference to the source database shared (`class var`) by all objects. When an object is created it is assigned a reference to the corresponding `TCustomerKey`, while a class data field refers to the source dataset. In each getter method, the class checks if the object has indeed been initialized before returning the data, as in the following case:

```
function TCustomer.GetCompany: string;
begin
  EnforceInit;
  Result := FCompany;
end;
```

The `EnforceInit` method checks a local flag, eventually calling `Init` to load data from the database to the in-memory object:

```
procedure TCustomer.EnforceInit;
begin
  if not fInitDone then
    Init;
end;

procedure TCustomer.Init;
begin
  RefDataSet.Locate('custno', CustKey.CustNo, []);

  // could also load each published field via RTTI
  FCustNo := RefDataSet.FieldByName ('CustNo').AsFloat;
  FCompany := RefDataSet.FieldByName ('Company').AsString;
  FCountry := RefDataSet.FieldByName ('Country').AsString;
  ...
  fInitDone := True;
end;
```

Given these two classes, I've added a special purpose dictionary to the application. This custom dictionary class inherits from a generic class instantiated with the proper types and adds to it a specific method:

```
type
  TCustomerDictionary = class (
    TObjectDictionary <TCustomerKey, TCustomer>)
  public
    procedure LoadFromDataSet (dataset: TDataSet);
  end;
```

The loading method populates the dictionary, copying data in memory for only the key objects:

```
procedure TCustomerDictionary.LoadFromDataSet(
  dataset: TDataSet);
var
  custKey: TCustomerKey;
begin
  TCustomer.RefDataSet := dataset;
  dataset.First;
  while not dataset.EOF do
  begin
    custKey := TCustomerKey.Create;
    custKey.CustNo := dataset ['CustNo'];
    custKey.Company := dataset ['Company'];
    self.Add(custKey, TCustomer.Create (custKey));
    dataset.Next;
  end;
end;
```

The demo program has a main form and a data module hosting a ClientDataSet component. The main form has a ListView control that is filled when a user presses the only button.

---

note You might want to replace the ClientDataSet component with a real dataset, expanding the example considerably in terms of usefulness, as you could run a query for the keys and a separate one for the actual data of each single `TCustomer` object. I have similar code, but adding it here would have distracted us too much from the goal of the example, which is experimenting with a generic dictionary class.

After loading the data in the dictionary, the `btnPopulateClick` method uses an enumerator on the dictionary's keys:

```
procedure TFormCustomerDictionary.btnPopulateClick(
  Sender: TObject);
var
  custkey: TCustomerKey;
  listItem: TListItem;
begin
  DataModule1.ClientDataSet1.Active := True;
  CustDict.LoadFromDataSet(DataModule1.ClientDataSet1);
```

```
  for custkey in CustDict.Keys do
  begin
    listItem := ListView1.Items.Add;
    listItem.Caption := custkey.Company;
    listItem.SubItems.Add(FloatTOStr (custkey.CustNo));
    listItem.Data := custkey;
  end;
end;
```

This fills the first two columns of the ListView control, with the data available in the key objects. Whenever a user selects an item of the ListView control, though, the program will fill a third column:

```
procedure TFormCustomerDictionary.ListView1SelectItem(
  Sender: TObject; Item: TListItem; Selected: Boolean);
var
  aCustomer: TCustomer;
begin
  aCustomer := CustDict.Items [Item.data];
  Item.SubItems.Add(IfThen (
    aCustomer.State <> '',
    aCustomer.State + ', ' + aCustomer.Country,
    aCustomer.Country));
end;
```

The method above gets the object mapped to the given key, and uses its data. Behind the scenes, the first time a specific object is used, the property access method triggers the loading of the entire data for the `TCustomer` object.

Dictionaries vs. String Lists

Over the years many Object Pascal developers, myself included, have overused the `TStringList` class. Not only you can use it for a plain list of strings and for a list of name/value pairs, but you can also use it to have a list objects associated with strings and search these objects.

Since the introduction of It is much better to use generics instead of this use a favorite tool as swiss-army knife kind of approach. Specific and focused container classes are a much better option. For example, a generic `TDictionary` with a string key and an object-value will generally be better than a string list on two counts: cleaner and safer code, as there will be fewer type casts involved, and faster execution, given that dictionaries use hash tables.

To demonstrate these differences I've written a rather simple application project, called `StringListVsDictionary`. Its main form stores two identical lists, declared as:

```
private
  sList: TStringList;
  sDict: TDictionary<string,TMyObject>;
```

The lists are filled with random but identical entries by a cycle, which repeats this code:

```
  sList.AddObject (aName, anObject);
  sDict.Add (aName, anObject);
```

Two buttons retrieve each element of the list doing a search by name on each of them. Both methods scan the string list for the values, but the first locates the objects in the string list, while the second uses the dictionary. Notice that in the first case you need an as cast to get back the given type, while the dictionary is tied to that class already. Here is the main loop of the two methods:

```
theTotal := 0;
for I := 0 to sList.Count -1  do
begin
  aName := sList[I];
  // now search for it
  anIndex := sList.IndexOf (aName);
  // get the object
  anObject := sList.Objects [anIndex] as TMyObject;
  Inc (theTotal, anObject.Value);
end;

theTotal := 0;
for I := 0 to sList.Count -1  do
begin
  aName := sList[I];
  // get the object
  anObject := sDict.Items [aName];
  Inc (theTotal, anObject.Value);
end;
```

I don't want to access the strings in sequence, but figure out how much time it takes to search in the sorted string list (which does a binary search) compared to the hashed keys of the dictionary. Not surprisingly the dictionary is faster, here are numbers in milliseconds for a test:

```
Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686
```

The result is the same, given the initial values weere identical, but the time is quite different, with the dictionary taking about *one fourth of the time* for a million entries.

Generic Interfaces

In the section “Sorting a TList<T>” you might have noticed a rather strange use of a predefined interface, which had a generic declaration. It is worth looking into this technique in detail, as it opens up significant opportunities.

The first technical element to notice is that it is perfectly legal to define a generic interface, as I've done in the `GenericInterface` example:

```
type
  IGetValue<T> = interface
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

note This is the generic version of the `IGetValue` interface of the `IntfContraints` application project, covered in the earlier section “Interface Constraints” of this chapter. In that case the interface had an Integer value, now it has a generic one.

Notice that differently from a standard interface, in case of a generic interface you don't need to specify a GUID to be used as Interface ID (or IID). The compiler will generate an IID for you for each instance of the generic interface, even if implicitly declared. In fact, you don't have to create a specific instance of the generic interface to implement it, but can define a generic class that implements the generic interface:

```
type
  TGetValue<T> = class (TInterfacedObject, IGetValue<T>)
  private
    fValue: T;
  public
    constructor Create (Value: T);
    destructor Destroy; override;
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

While the constructor assigns the initial value of the object, the destructor's only purpose is to log that an object was destroyed. We can create an instance of this generic class (thus generating a specific instance of the interface type behind the scenes) by writing:

```
procedure TFormGenericInterface.btnValueClick(
  Sender: TObject);
var
  aVal: TGetValue<string>;
begin
  aVal := TGetValue<string>.Create (Caption);
  try
```

```
    Show ('TGetValue value: ' + aVal.GetValue);
  finally
    aVal.Free;
  end;
end;
```

An alternative approach, as we saw in the past for the `IntfConstraint` application project, is to use an interface variable of the corresponding type, making the specific interface type definition explicit (and not implicit as in the previous code snippet):

```
procedure TFormGenericInterface.btnIValueClick(
  Sender: TObject);
var
  aVal: IGetValue<string>;
begin
  aVal := TGetValue<string>.Create (Caption);
  Show ('IGetValue value: ' + aVal.GetValue);
  // freed automatically, as it is reference counted
end;
```

Of course, we can also define a specific class that implements the generic interface, as in the following scenario (from the `GenericInterface` application project):

```
type
  TButtonValue = class (TButton, IGetValue<Integer>)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;
```

Notice that while the `TGetValue<T>` generic class implements the generic `IGetValue<T>` interface, the `TButtonValue` specific class implements the `IGetValue<Integer>` specific interface. Specifically, as in a previous example, the interface is remapped to the `Left` property of the control:

```
function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;
```

In the class above, the `MakeTButtonValue` class function is a ready-to-use method to create an object of the class. This method is used by the third button of the main form, as follows:

```
procedure TFormGenericInterface.btnValueButtonClick(
  Sender: TObject);
var
  iVal: IGetValue<Integer>;
begin
  iVal := TButtonValue.MakeTButtonValue (self, ScrollBox1);
  Show ('Button value: ' + IntToStr (iVal.GetValue));
end;
```

Although it is totally unrelated to generic classes, here is the implementation of the `MakeTButtonValue` class function:

```
class function TButtonValue.MakeTButtonValue(
  Owner: TComponent; Parent: TWinControl): TButtonValue;
begin
  Result := TButtonValue.Create(Owner);
  Result.Parent := Parent;
  Result.SetBounds(Random (Parent.Width),
    Random (Parent.Height), Result.Width, Result.Height);
  Result.Caption := 'btnv';
end;
```

Predefined Generic Interfaces

Now that we have explored how to define generic interfaces and combine them with the use of generic and specific classes, we can get back to having a second look at the `Generics.Default` unit. This unit defines two generic comparison interfaces:

`IComparer<T>` has a `Compare` method

`IEqualityComparer<T>` has `Equals` and `GetHashCode` methods

These classes are implemented by some generic and specific classes, listed below (with no implementation details):

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  TEqualityComparer<T> = class(
    TInterfacedObject, IEqualityComparer<T>)
  TCustomComparer<T> = class(TSingletonImplementation,
    IComparer<T>, IEqualityComparer<T>)
  TStringComparer = class(TCustomComparer<string>)
```

In the listing above you can see that the base class used by the generic implementations of the interfaces is either the classic reference-counted `TInterfacedObject` class or the new `TSingletonImplementation` class. This is an oddly named class that provides a basic implementation of `IInterface` with no reference counting.

---

note The term singleton is generally used to define a class of which you can create only one instance, and not one with no reference counting. I consider this quite a misnomer.

As we have already seen in the "Sorting a TList<T>" section earlier in this chapter, these comparison classes are used by the generic containers. To make things more complicated, though, the `Generics.Default` unit relies quite heavily on anonymous methods, so you should probably look at it only after reading the next chapter.

Smart Pointers in Object Pascal

When approaching generics, you might get the wrong first impression that this language construct is mostly used for collections. While this is the simplest case for using generic classes, and very often the first example in books and docs, generics are useful well beyond the realm of collection (or container) classes. In the last example of this chapter I'm going to show you a *non-collection* generic type, that is the definition of a smart pointer.

If you come from an Object Pascal background, you might not have heard of smart pointers, an idea that comes from the C++ language. In C++ you can have pointers to objects, for which you have to manage memory directly and manually, and local object variables that are managed automatically but have many other limitations (including the lack of polymorphism). The idea of a smart pointer is to use a locally managed object to take care of the lifetime of the pointer to the real object you want to use. If this sounds too complicated, I hope the Object Pascal version (and its code) will help clarify it.

note The term polymorphisms in OOP languages is used to denote the situation in which you assign to a variable of a base class an object of a derived class and call one of the base class virtual methods, potentially ending up calling the version of the virtual method of the specific subclass.

A Smart Pointer Generic Record

In Object Pascal objects are managed by reference, but records have a lifetime bound to the method in which they are declared. When the method ends, the memory area for the record is cleaned up. So what we can do is to use a record to manage the lifetime of an Object Pascal object. Of course, we want to write the code only once, so we can use a generic record. Here is a first version:

```
type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    function GetValue: T;
  public
    constructor Create(AValue: T);
    property Value: T read GetValue;
  end;
```

The `Create` and `GetValue` methods of the record could simply assign and read back the value. Using this code you can create an object, create a smart pointer wrapping it, and refer from one to the other:

```
var
  sl: TStringList;
  smartP: TSmartPointer<TStringList>;
begin
  sl := TStringList.Create;
  smartP.Create (sl);
  sl.Add('foo');
  smartP.Value.Add ('foo2');
```

As you may have worked out, this code causes a memory leak in the exact same way as without the smart pointer! In fact the record is destroyed as it goes out of scope, but it has no way of freeing the internal object. Considering a record has no destructor, how can we manage the object disposal? A trick is to use an interface inside the record itself, as the record will automatically free the interfaced object. Should we add an interface to the object we are wrapping? Probably not, as this imposes a significant limitation on the objects we'll be able to pass to the smart pointer.

Interfaces to the Rescue

A better alternative is probably to write a specific wrapper class, tied to an interface, and use the interface reference counting mechanism to the wrapped object. The internal class might look like the following:

```
type
  TFreeTheValue = class (TInterfacedObject)
  private
    fObjectToFree: TObject;
  public
    constructor Create(anObjectToFree: TObject);
    destructor Destroy; override;
  end;

constructor TFreeTheValue.Create(
  anObjectToFree: TObject);
begin
  fObjectToFree := anObjectToFree;
end;

destructor TFreeTheValue.Destroy;
begin
  fObjectToFree.Free;
  inherited;
end;
```

Even better, in the actual example I've declared this as a nested type of the generic smart pointer type. All we have to do in the smart pointer generic type, to enable this feature, is to add an interface reference and initialize it with a `TFreeTheValue` object referring to the contained object:

```
type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    FFreeTheValue: IInterface;
    function GetValue: T;
  public
    constructor Create(AValue: T); overload;
    property Value: T read GetValue;
  end;
```

The pseudo-constructor (records don't have real constructors) becomes:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
  FFreeTheValue := TFreeTheValue.Create(FValue);
end;
```

Using the Smart Pointer

With this code in place, we can now write the following code in a program without causing a memory leak:

```
procedure TFormSmartPointers.btnSmartClick(
  Sender: TObject);
var
  sl: TStringList;
  smartP: TSmartPointer<TStringList>;
begin
  sl := TStringList.Create;
  smartP.Create (sl);
  sl.Add('foo');
  Show ('Count: ' + IntToStr (sl.Count));
end;
```

At the end of the method the `smartP` record is disposed, which causes its internal interfaced object to be destroyed, freeing the `TStringList` object. Notice that this disposal takes place even when an exception is raised, so we don't need to protect our code with a `try-finally` block.

---

note In practice, implicit `try-finally` blocks are being added all over the places by the compiler to handle the interface within the record, but we don't have to write them (and the compiler is less likely to forget one).

In the program, I verify that all objects are actually destroyed and there is no memory leak by setting the global `ReportMemoryLeaksOnShutdown` to True in the initialization code. As a counter test, there is a button in the program that causes a leak, which is caught as the program terminates.

Adding Implicit Conversion

So using the smart pointer record we have been able to remove the need for the `Free` call, and hence the need for a `try-finally` block, but there is still quite some code to write (and to remember writing). An extension to the smart pointer class is the inclusion of an `Implicit` conversion operator, providing the capability to assign the target object to the smart pointer:

```
class operator TSmartPointer<T>.
  Implicit(AValue: T): TSmartPointer<T>;
begin
  Result := TSmartPointer<T>.Create(AValue);
end;
```

With this code (and taking advantage of the `Value` field) we can now write a more compact version of the code, like:

```
var
  smartP: TSmartPointer<TStringList>;
begin
  smartP := TStringList.Create;
  smartP.Value.Add('foo');
  Show ('Count: ' + IntToStr (smartP.Value.Count));
```

As an alternative, we can use a `TStringList` variable and use a complicated constructor to initialize the smart pointer record even without an explicit reference to it:

```
var
  sl: TStringList;
begin
  sl := TSmartPointer<TStringList>.
    Create(TStringList.Create).Value;
  sl.Add('foo');
  Show ('Count: ' + IntToStr (sl.Count));
```

As we've started down this road, we can also define the opposite conversion, and use the cast notation rather than the `Value` property:

```
class operator TSmartPointer<T>.
  Implicit(AValue: T): TSmartPointer<T>;
begin
  Result := TSmartPointer<T>.Create(AValue);
end;

var
```

```
  smartP: TSmartPointer<TStringList>;
begin
  smartP := TStringList.Create;
  TStringList(smartP).Add('foo2');
```

Now, you might also notice that I've always used a pseudo-constructor in the code above, but this is not needed on a record. All we need is a way to initialize the internal object, possibly calling its constructor, the first time we use it. We cannot test if the internal object is `Assigned`, because records (unlike classes) are not initialized to zero. However we can perform that test on the interface variable, which is initialized.

Auto-Creation

The extra code of the smart pointer record type is an overloaded `Create` procedure (it cannot be a constructor, as parameterless constructors are not legal for records) and a lazy initialization of the `Value` property:

```
procedure TSmartPointer<T>.Create;
begin
  TSmartPointer<T>.Create (T.Create);
end;

function TSmartPointer<T>.GetValue: T;
begin
  if not Assigned(FFreeTheValue) then
    self := TSmartPointer<T>.Create (T.Create);
  Result := FValue;
end;
```

With this code we now have many ways to use the smart pointer, including not freeing and not even creating it explicitly:

```
var
  smartP: TSmartPointer<TStringList>;
begin
  smartP.Value.Add('foo');
  Show ('Count: ' + IntToStr (smartP.Value.Count));
end;
```

The fact that the method above creates a string list and frees it at the end sounds certainly a big departure from the standard coding model Object Pascal developers are used to. And this is only one specific case of using generics for non collections code.

The Complete Smart Pointer Code

To end this section, though. Let me list the complete source code of the smart pointer generic record I've build in several iterations:

```
type
  TSmartPointer<T: class, constructor> = record
  strict private
    FValue: T;
    FFreeTheValue: IInterface;
    function GetValue: T;
  private
    type
      TFreeTheValue = class (TInterfacedObject)
      private
        fObjectToFree: TObject;
      public
        constructor Create(anObjectToFree: TObject);
        destructor Destroy; override;
      end;
  public
    constructor Create(AValue: T); overload;
    procedure Create; overload;
    class operator Implicit(AValue: T): TSmartPointer<T>;
    class operator Implicit(smart: TSmartPointer <T>): T;
    property Value: T read GetValue;
  end;
```

The complete code and some of the usage patterns mentioned in this section are in the `SmartPointers` project. Now, I'm not advocating using this type of code regularly, rather than more standard memory management techniques. The reason for this section of the book is to highlight the depth of Object Pascal, which makes it possible to write some very complex and powerful code like the implementation of smart pointers explained in this section.

Covariant Return Types with Generics

In general in Object Pascal (and most other static object-oriented languages) a method can return an object of a class but you cannot override it in a derived class to return a derived class object. This is a rather common practice called “Covariant Return Type” and explicitly supported by some languages like C++.

Of Animals, Dogs, and Cats

In coding terms, if `TDog` inherits from `TAnimal`, I'd want to have the methods:

```
function TAnimal.Get (name: string): TAnimal;
function TDog.Get (name: string): TDog;
```

However, in Object Pascal you cannot have virtual functions with a different return value, nor you can overload on the return type, but only when using different parameters. Let me show you the complete code of a simple demo. Here are the three classes involved:

```
type
  TAnimal = class
  private
    FName: string;
    procedure SetName(const Value: string);
  public
    property Name: string read FName write SetName;
  public
    class function Get (const aName: string): TAnimal;
    function ToString: string; override;
  end;

  TDog = class (TAnimal)

  end;

  TCat = class (TAnimal)

  end;
```

The implementation of the two methods is quite simple, once you notice that the class function is actually used to create new objects, internally calling a constructor. The reason is I don't want to create a constructor directly is that this is a more general technique, in which a method of a class can create objects of other classed (or class hierarchies). This is the code:

```
class function TAnimal.Get(const aName: string): TAnimal;
begin
  Result := Create;
  Result.fName := aName;
end;

function TAnimal.ToString: string;
begin
  Result := 'This ' + Copy (ClassName, 2, maxint) +
    ' is called ' + FName;
end;
```

Now we can use the class by writing the following code, which is what I don't terribly like, given we have to cast back the result to the proper type:

```
var
  aCat: TCat;
begin
  aCat := TCat.Get('Matisse') as TCat;
  Memo1.Lines.Add (aCat.ToString);
  aCat.Free;
```

Again, what I'd like to do is to be able to assigned the value returned by `TCat.Get` to a reference of the `TCat` class without an explicit cast. How can we do that?

A Method with a Generic Result

It turns out generics can help us solve the problem. Not generic types, which is the most commonly used form of generics. But generic methods for non-generic types, discussed earlier in this chapter. What I can add to the `TAnimal` class is a method with a *generic type* parameter, like:

```
  class function GetAs<T: class> (const aName: string): T;
```

This method requires a generic type parameter, which needs to be a class (or instance type) and returns an object of that type. A sample implementation is here:

```
class function TAnimal.GetAs<T>(const aName: string): T;
var
  res: TAnimal;
begin
  res := Get (aName);
  if res.inheritsFrom (T) then
    Result := T(res)
  else
    Result := nil;
end;
```

Now we can create an instance and using it omitting the *as* cast, although we still have to pass the type as parameter:

```
var
  aDog: TDog;
begin
  aDog := TDog.GetAs<TDog>('Pluto');
  Memo1.Lines.Add (aDog.ToString);
  aDog.Free;
```

Returning a Derived Object of a Different Class

When you return an object of the same class, you can replace this code with a proper use of constructors. But the use of generics to obtain covariant return types is actually more flexible. In fact we can use it to return objects of a different class, or hierarchy of classes:

```
type
  TAnimalShop = class
    class function GetAs<T: TAnimal, constructor> (
      const aName: string): T;
  end;
```

note A class like this, used to create objects of a different class (or more than one, depending on the parameters os status) is generally called a "*class factory*".

We can now use the specific class constraint (something impossible in the class itself) and we have to specify the constructor constraint to be able to create an object of the given class from within the generic method:

```
class function TAnimalShop.GetAs<T>(const aName: string): T;
var
  res: TAnimal;
begin
  res := T.Create;
  res.Name := aName;
  if res.inheritsFrom (T) then
    Result := T(res)
  else
    Result := nil;
end;
```

Notice that now in the call we don't have to repeat the class type twice:

```
  aDog := TAnimalShop.GetAs<TDog>('Pluto');
```

15: anonymous methods

The Object Pascal language includes both procedural types (that is, types declaring pointers to procedures and functions) and method pointers (that is, types declaring pointers to methods).

note In case you want more information, procedural types were covered in Chapter 4 while events and method pointer types were described Chapter 10.

Although you might seldom use them directly, these are key features of Object Pascal that every developer works with. In fact, method pointers types are the foundation for event handlers in components and visual controls: every time you declare an event handler, even a simple `Button1Click` you are in fact declaring a method that will be connected to an event (the `OnClick` event, in this case) using a method pointer.

Anonymous methods extend this feature by letting you pass the actual code of a method as a parameter, rather than the name of a method defined elsewhere. This is not the only difference, though. What makes anonymous methods very different from other techniques is the way they manage the lifetime of local variables.

The definition above matches with a feature called closures in many other languages, for example JavaScript. If Object Pascal anonymous methods are in fact closures, how come the language refers to them using a different term? The reason lies in the fact both terms are used by different languages and that the C++ compiler produced by Embarcadero uses the term closures for what Object Pascal calls event handlers (so having a different feature with the same name would have been confusing).

Anonymous methods have been around in different forms and with different names for many years in quite a few programming languages, most notably dynamic languages. I've had extensive experience with closures in JavaScript, particularly with the jQuery library and AJAX calls. The corresponding feature in C# is called an anonymous delegate.

But here I don't want to devote too much time comparing closures and related techniques in the various programming languages, but rather describe in detail how they work in Object Pascal.

---

note From a very high perspective, generics allows code to be code parametrized for a type, anonymous methods allows code to be parametrized for a method.

Syntax and Semantics of Anonymous Methods

An anonymous method in Object Pascal is a mechanism to *create a method value in an expression context*. A rather cryptic definition, but a rather precise one given it underlines the key difference from method pointers, the *expression context*. Before we get to this, though, let me start from the beginning with a very simple code example (included in the `AnonymFirst` application project along with most other code snippets in this section).

This is the declaration of an anonymous method type, something you need to provide given that Object Pascal is a strongly typed language:

```
type
  TIntProc = reference to procedure (n: Integer);
```

This is different from a method reference type only in the keywords being used for the declaration:

```
type
  TIntMethod = procedure (n: Integer) of object;
```

An Anonymous Method Variable

Once you have an anonymous method type you can, in the simplest cases, declare a variable of this type, assign a type-compatible anonymous method, and call the method through the variable:

```
procedure TFormAnonymFirst.btnSimpleVarClick(
  Sender: TObject);
var
  anIntProc: TIntProc;
begin
  anIntProc :=
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (IntToStr (n));
    end;
  anIntProc (22);
end;
```

Notice the syntax used to assign an actual procedure, with in-place code, to the variable.

An Anonymous Method Parameter

As a more interesting example (with even more surprising syntax), we can pass an anonymous method as parameter to a function. Suppose you have a function taking an anonymous method parameter:

```
procedure CallTwice (value: Integer;
  anIntProc: TIntProc);
begin
  anIntProc (value);
  Inc (value);
  anIntProc (value);
end;
```

The function calls the method passed as parameter twice with two consecutive integers values, the one passed as parameter and the following one. You call the function by passing an actual anonymous method to it, with directly in-place code that looks surprising:

```
procedure TFormAnonymFirst.btnProcParamClick(
  Sender: TObject);
begin
  CallTwice (48,
    procedure (n: Integer)
    begin
      Show (IntToHex (n, 4));
    end);
```

```
  CallTwice (100,
    procedure (n: Integer)
    begin
      Show (FloatToStr(Sqrt(n)));
    end);
end;
```

From the syntax point of view notice the procedure passed as parameter within parentheses and not terminated by a semicolon. The actual effect of the code is to call the `IntToHex` with 48 and 49 and the `FloatToStr` on the square root of 100 and 101, producing the following output:

```
0030
0031
10
10.0498756211209
```

Using Local Variables

We could have achieved the same effect using method pointers albeit with a different and less readable syntax. What makes anonymous method clearly different is the way they can refer to local variables of the calling method. Consider the following code:

```
procedure TFormAnonymFirst.btnLocalValClick(
  Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 0;
  CallTwice (10,
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
    end);
  Show (IntToStr (aNumber));
end;
```

Here the method, still passed to the `CallTwice` procedure, uses the local parameter `n`, but also a local variable from the context from which it was called, `aNumber`. What's the effect? The two calls of the anonymous method will modify the local variable, adding the parameter to it, 10 the first time and 11 the second. The final value of `aNumber` will be 21.

Extending the Lifetime of Local Variables

The previous example shows an interesting effect, but with a sequence of nested function calls, the fact you can use the local variable isn't that surprising. The power of anonymous methods, however, lies in the fact they can use a local variable and also extend its lifetime until needed. An example will prove the point more than a lengthly explanation.

note In slightly more technical details, anonymous methods copy the variables and parameters they use to the heap when they are created, and keep them alive as long as the specific instance of the anonymous method.

I've added (using class completion) to the `TFormAnonymFirst` form class of the `AnonymFirst` example a property of an anonymous method pointer type (well, actually the same anonymous method pointer type I've used in all of the code of the project):

```
  private
    FAnonMeth: TIntProc;
    procedure SetAnonMeth(const Value: TIntProc);
  public
    property AnonMeth: TIntProc
      read FAnonMeth write SetAnonMeth;
```

Then I've added two more buttons to the form of the program. The first saves the property an anonymous method that uses a local variable (more or less like in the previous `btnLocalValClick` method):

```
procedure TFormAnonymFirst.btnStoreClick(
  Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 3;
  AnonMeth :=
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
      Show (IntToStr (aNumber));
    end;
end;
```

When this method executes the anonymous method is not executed, only stored. The local variable `aNumber` is initialized to three, is not modified, goes out of local scope (as the method terminates), and is displaced. At least, that is what you'd expect from standard Object Pascal code.

The second button I added to the form for this specific step calls the anonymous method stored in the `AnonMeth` property:

```
procedure TFormAnonymFirst.btnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
  end;
end;
```

When this code is executed, it calls an anonymous method that uses the local variable `aNumber` of a method that's not on the stack any more. However, since anonymous methods *capture* their execution context the variable is still there and can be used as long as that given instance of the anonymous method (that is, a reference to the method) is around.

As a further proof, do the following. Press the `Store` button once, the `Call` button two times and you'll see that the same *captured* variable is being used:

```
5
8
10
13
```

note The reason for this sequence is that the value starts at 3, each call to `CallTwice` passed its parameter to the anonymous methods a first time (that is 2) and then a second time after incrementing it (that is, the second time it passes 3).

Now press `Store` once more and press `Call` again. What happens, why is the value of the local variable reset? By assigning a new anonymous method instance, the old anonymous method is deleted (along with its own execution context) and a new execution context is capture, including a new instance of the local variable. The full sequence *Store – Call – Call – Store – Call* produces:

```
5
8
10
13
5
8
```

It is the implication of this behavior, resembling what some other languages do, that makes anonymous methods an extremely powerful language feature, which you can use to implement something that simply wasn't possible in the past.

Anonymous Methods Behind the Scenes

If the variable capture feature is one of the most relevant for anonymous methods, there are a few more techniques that are worth looking at, before we focus on some real world examples.

The (Potentially) Missing Parenthesis

Notice that in the code above I used the `AnonMeth` symbol to refer to the anonymous method, not to invoke it. For invoking it, I should have typed:

```
AnonMeth (2)
```

The difference is clear; I need to pass a proper parameter to invoke the method. Things are slightly more confusing with parameterless anonymous methods. If you declare:

```
type
  TAnyProc = reference to procedure;
var
  AnyProc: TAnyProc;
```

The call to `AnyProc` must be followed by the empty parentheses, otherwise the compiler thinks you are trying to get the method (its address) rather than call it:

```
AnyProc ();
```

Something similar happens when you call a function that returns an anonymous method, as in the following case taken from the usual `AnonymFirst` application project:

```
function GetShowMethod: TIntProc;
var
  x: Integer;
begin
  x := Random (100);
  ShowMessage ('New x is ' + IntToStr (x));
  Result :=
    procedure (n: Integer)
    begin
      x := x + n;
      ShowMessage (IntToStr (x));
    end;
end;
```

Now the question is, how do you call it? If you simply call

```
GetShowMethod;
```

It compiles and executes, but all it does is call the anonymous method assignment code, throwing away the anonymous method returned by the function.

How do you call the actual anonymous method passing a parameter to it? One option is to use a temporary anonymous method variable:

```
var
  ip: TIntProc;
begin
  ip := GetShowMethod();
  ip (3);
```

Notice in this case the parentheses after the `GetShowMethod` call. If you omit them (a standard Pascal practice) you'll get the following error:

```
E2010 Incompatible types: 'TIntProc' and 'Procedure'
```

Without the parentheses the compiler thinks you want to assign the `GetShowMethod` function itself, and not its result to the `ip` method pointer. Still, using a temporary variable might not be the best option in this case, as is makes the code unnaturally complex. A simple call

```
GetShowMethod(3);
```

won't compile, as you cannot pass a parameter to the method. You need to add the empty parenthesis to the first call, and the Integer parameter to the resulting anonymous method. Oddly enough, you can write:

```
GetShowMethod()(3);
```

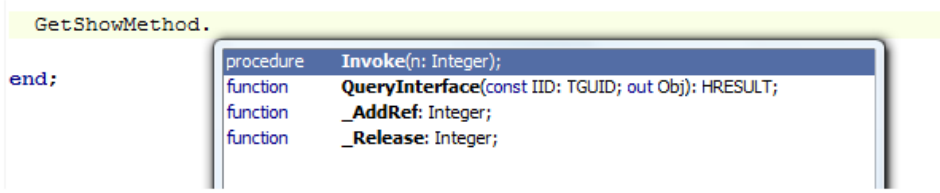
An alternative solution is to use the internal implementation of anonymous methods, and call the low-level `Invoke` method that gets added by the compiler (in which case you can omit the empty parenthesis):

```
GetShowMethod.Invoke (3);
```

Anonymous Methods Implementation

What is this `Invoke` method? What happens behind the scenes in the implementation of anonymous methods? The actual code generated by the compiler for anonymous methods is based on interfaces, with a single invocation method called `Invoke`, plus the usual reference counting support (that's useful to determine the lifetime of anonymous methods and the context they capture).

You can see those interface methods in the editor if you use code completion, in the following way:



Getting details of the internals is probably very complicated and of limited worth. Suffice to say that the implementation is very efficient, in terms of speed, and requires about 500 extra bytes for each anonymous method.

In other words, a method reference in Object Pascal is implemented with a *special* single method interface, with a compiler-generated method having the same signature as the method reference it is implementing. The interface takes advantage of reference counting for its automatic disposal.

---

note Although practically the interface used for an anonymous method looks like any other interface, the compiler distinguishes between these *special* interfaces so you cannot mix them in code.

Beside this hidden interface, for each invocation of an anonymous method the compiler creates a hidden object that has the method implementation and the data required to *capture* the invocation context. That's how you get a new set of captured variables for each call of the method.

Ready To Use Reference Types

Every time you use an anonymous method as a parameter you need to define a corresponding reference pointer data type. To avoid the proliferation of local types, Object Pascal provides a number of ready-to-use reference pointer types in the `SysUtils` unit. As you can see in the code snippet below, most of these type definitions use parameterized types, so that with a single generic declaration you have a different reference pointer type for each possible data type:

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure (Arg1: T);
  TProc<T1,T2> = reference to procedure (
    Arg1: T1; Arg2: T2);
  TProc<T1,T2,T3> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3);
  TProc<T1,T2,T3,T4> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);
```

Using these declarations, you can define procedures that take anonymous method parameters like in the following:

```
procedure UseCode (proc: TProc);
function DoThis (proc: TProc): string;
function DoThat (procInt: TProc<Integer>): string;
```

In the first and second case you pass a parameterless anonymous method, in the third you pass a method with a single Integer parameter:

```
UseCode (
  procedure
  begin
    ...
  end);
strRes := DoThat (
  procedure (I: Integer)
  begin
    ...
  end);
```

Similarly the `SysUtils` unit defines a set of anonymous method types with a generic return value:

```
type
  TFunc<TResult> = reference to function: TResult;
  TFunc<T,TResult> = reference to function (
    Arg1: T): TResult;
  TFunc<T1,T2,TResult> = reference to function (
    Arg1: T1; Arg2: T2): TResult;
  TFunc<T1,T2,T3,TResult> = reference to function (
    Arg1: T1; Arg2: T2; Arg3: T3): TResult;
  TFunc<T1,T2,T3,T4,TResult> = reference to function (
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4): TResult;
  TPredicate<T> = reference to function (
    Arg1: T): Boolean;
```

These definitions are very broad, as you can use countless combinations of data types for up to four parameters and a return type. The last definition is very similar to the second, but corresponds to a specific case that is very frequent, a function taking a generic parameter and returning a Boolean.

Anonymous Methods in the Real World

At first sight, it is not easy to fully understand the power of anonymous methods and the scenarios that can benefit from using them. That's why rather than coming out

with more convoluted examples covering the language, I decided to focus on some that have a practical impact and provide starting points for further exploration.

Anonymous Event Handlers

One of the distinguishing features of Object Pascal has been its implementation of event handlers using method pointers. Anonymous methods can be used to attach a new behavior to an event without having to declare a separate method and capturing the method's execution context. This avoids having to add extra fields to a form to pass parameters from one method to another.

As an example (called `AnonButton`), I've added an *anonymous click* event to a button, declaring a proper method pointer type and adding a new event handler to a custom button class (defined using an interceptor class):

```
type
  TAnonNotif = reference to procedure (Sender: TObject);

  // interceptor class
  TButton = class (FMX.StdCtrls.TButton)
  private
    FAnonClick: TAnonNotif;
    procedure SetAnonClick(const Value: TAnonNotif);
  public
    procedure Click; override;
  public
    property AnonClick: TAnonNotif
      read FAnonClick write SetAnonClick;
  end;
```

note An *interceptor* class is a derived class having the same name as its base class. Having two classes with the same name is possible because the two classes are in different units, so their full name (*unitname.classname*) is different. Declaring an interceptor class can be handy as you can simply place a Button control on the form and attach extra behavior to it, without having to install a new component in the IDE and replace the controls on your form with the new type. The only trick you have to remember is that if the definition of the interceptor class is in a separate unit (not the form unit as in this simple example), that unit has to be listed in the uses statement after the unit defining the base class.

The code of this class is fairly simple, as the setter method saves the new pointer and the `Click` method calls it before doing the standard processing (that is, calling the `OnClick` event handler if available):

```
procedure TButton.SetAnonClick(const Value: TAnonNotif);
begin
  FAnonClick := Value;
end;
```

```
procedure TButton.Click;
begin
  if Assigned (FAnonClick) then
    FAnonClick (self)
  inherited;
end;
```

How can you use this new event handler? Basically you can assign an anonymous method to it

```
procedure TFormAnonButton.btnAssignClick(
  Sender: TObject);
begin
  btnInvoke.AnonClick :=
    procedure (Sender: TObject)
    begin
      Show ((Sender as TButton).Text);
    end;
end;
```

Now this looks rather pointless, as the same effect could easily be achieved using a standard event handler method. The following, instead, starts making a difference, as the anonymous method captures a reference to the component that assigned the event handler, by referencing the `Sender` parameter.

This can be done after temporarily assigning it to a local variable, as the `Sender` parameter of the anonymous method hides the `btnKeepRefClick` method's `Sender` parameter:

```
procedure TFormAnonButton.btnKeepRefClick(
  Sender: TObject);
var
  aCompRef: TComponent;
begin
  aCompRef := Sender as TComponent;
  btnInvoke.AnonClick :=
    procedure (Sender: TObject)
    begin
      Show ((Sender as TButton).Text +
         ' assigned by ' + aCompRef.Name);
    end;
end;
```

As you press the `btnInvoke` button, you'll see its caption along with the name of the component that assigned the anonymous method handler.

Timing Anonymous Methods

Developers frequently add timing code to existing routines to compare their relative speed. Supposing you have two code fragments and you want to compare their speed by executing them a few million times, you could write the following which is taken from the `LargeString` application project of Chapter 6:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  str1, str2: string;
  I: Integer;
  t1: TStopwatch;
begin
  str1 := 'Marco ';
  str2 := 'Cantu ';

  t1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    str1 := str1 + str2;

  t1.Stop;
  Show('Length: ' + str1.Length.ToString);
  Show('Concatenation: ' + t1.ElapsedMilliseconds.ToString);
end;
```

A second method has similar code but used the `TStringBuilder` class rather than plain concatenation. Now we can take advantage of anonymous methods to create a timing skeleton and pass the specific code as parameter, as I've done in an updated version of the code, in the `AnonLargeStrings` application project.

Rather than repeating the timing code over and over, you can write a function with the timing code that would invoke the code snippet through a parameterless anonymous method:

```
function TimeCode (nLoops: Integer; proc: TProc): string;
var
  t1: TStopwatch;
  I: Integer;
begin
  t1 := TStopwatch.StartNew;
  for I := 1 to nLoops do
    proc;
  t1.Stop;
  Result := t1.ElapsedMilliseconds.toString;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  str1, str2: string;
begin
  str1 := 'Marco ';
```

```
  str2 := 'Cantu ';
  Show ('Concatenation: ' +
    TimeCode (MaxLoop,
      procedure ()
      begin
        str1 := str1 + str2;
      end));
  Show('Length: ' + str1.Length.ToString);
end;
```

Notice, though, that if you execute the standard version and the one based on anonymous methods you'll get a slightly different output, the anonymous method version sees a penalty of roughly 10%. The reason is that rather than directly executing the local code, the program has to make a virtual call to the anonymous method implementation. As this difference is consistent, the testing code makes perfect sense anyway. However, if you need to squeeze performance from your code, using anonymous methods won't be as fast as directly writing the code, with using a direct function. Using a non-virtual method pointer would probably be somewhere in between the two in terms of performance.

Thread Synchronization

In multi-threaded applications that need to update the user interface, you cannot access properties of visual components (or in memory-objects) that are part of the global thread without a synchronization mechanism. The visual component libraries for Object Pascal, in fact, aren't thread-safe (as is true for most user-interface libraries). Two threads accessing an object at the same time could compromise its state.

The classic solution offered by the `TThread` class in Object Pascal is to call a special method, `Synchronize`, passing as a parameter the reference to another method, the one to be executed safely. This second method cannot have parameters, so it is common practice to add extra fields to the thread class to pass the information from one method to another.

As a practical example, in the book Mastering Delphi 2005 I wrote a `WebFind` application (a program that runs searches on Google via HTTP and extracts the resulting links from the HTML of the page), with the following thread class:

```
type
  TFindWebThread = class(TThread)
  protected
    Addr, Text, Status: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
```

```
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject;
      AWorkMode: TWorkMode; AWorkCount: Int64);
  public
    strUrl: string;
    strRead: string;
  end;
```

The three protected string fields and some of the extra methods have been introduced to support synchronization with the user interface. For example, the `HttpWork` event handler hooked to an event of an internal `IdHttp` object (an Indy component supporting the client side of the HTTP protocol), used to have the following code, that called the `ShowStatus` method:

```
procedure TFindWebThread.HttpWork(Sender: TObject;
  AWorkMode: TWorkMode; AWorkCount: Int64);
begin
  Status := 'Received ' + IntToStr (AWorkCount) +
    ' for ' + strUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := Status;
end;
```

The `Synchronize` method of the Object Pascal RTL has two different overloaded definitions:

```
type
  TThreadMethod = procedure of object;
  TThreadProcedure = reference to procedure;

  TThread = class
    ...
    procedure Synchronize(
      AMethod: TThreadMethod); overload;
    procedure Synchronize(
      AThreadProc: TThreadProcedure); overload;
```

For this reason we can remove the `Status` text field and the `ShowStatus` function, and rewrite the `HttpWork` event handler using the new version of `Synchronize` and an anonymous method:

```
procedure TFindWebThreadAnon.HttpWork(Sender: TObject;
  AWorkMode: TWorkMode; AWorkCount: Int64);
begin
  Synchronize (
    procedure
    begin
      Form1.StatusBar1.SimpleText :=
        'Received ' + IntToStr (AWorkCount) +
```

```
        ' for ' + strUrl;
    end);
end;
```

Using the same approach throughout the code of the class, the thread class becomes the following (you can find both thread classes in the version of the `WebFind` application project that comes with the source code of this book):

```
type
  TFindWebThreadAnon = class(TThread)
  protected
    procedure Execute; override;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (Sender: TObject;
      AWorkMode: TWorkMode; AWorkCount: Int64);
  public
    strUrl: string;
    strRead: string;
  end;
```

Using anonymous methods simplifies the code needed for thread synchronization.

note Anonymous methods have a lot of relationships with threading, because a thread is used to run some code and anonymous method represent code. This is why there is support in the `TThread` class to use them, but also in the Parallel Programming library (in `TParallel.For` and to define a `TTask`). Given examining multi-threading goes well beyond this chapter, I won't add more examples in this direction. Still, I'm going to use another thread in the next example, because this is most often a requirement when making an HTTP call.

AJAX in Object Pascal

The last example in this section, the `AnonAjax` application demo, is one of my favorite examples of anonymous methods (even if a bit extreme). The reason is that I learned using closures (or anonymous methods) in JavaScript, while writing AJAX applications with the jQuery library.

note The AJAX acronym stands for Asynchronous JavaScript XML, as this was originally the format used in web services calls done from the browser. As this technology became more popular and widespread, and web services moved to the REST architecture and the JSON format, the term AJAX has faded away a bit, in favor of REST.

The `AjaxCall` global function spawns a thread, passing to the thread an anonymous method to execute on completion. The function is just a wrapper around the thread constructor:

```
type
```

```
  TAjaxCallback = reference to procedure (
    ResponseContent: TStringStream);

procedure AjaxCall (const strUrl: string;
  ajaxCallback: TAjaxCallback);
begin
  TAjaxThread.Create (strUrl, ajaxCallback);
end;
```

All of the code is in the `TAjaxThread` class, a thread class with an internal Indy HTTP client component used to access to a given URL, asynchronously:

```
type
  TAjaxThread = class (TThread)
  private
    fIdHttp: TIdHttp;
    fURL: string;
    fAjaxCallback: TAjaxCallback;
  protected
    procedure Execute; override;
  public
    constructor Create (const strUrl: string;
      ajaxCallback: TAjaxCallback);
    destructor Destroy; override;
  end;
```

The constructor does some initialization, copying its parameters to the corresponding local fields of the thread class and creating the `fIdHttp` object. The real meat of the class is in its `Execute` method, which does the HTTP request, saving the result in a stream that is later reset and passed to the callback function – the anonymous method:

```
procedure TAjaxThread.Execute;
var
  aResponseContent: TStringStream;
begin
  aResponseContent := TStringStream.Create;
  try
    fIdHttp.Get (fURL, aResponseContent);
    aResponseContent.Position := 0;
    fAjaxCallback (aResponseContent);
  finally
    aResponseContent.Free;
  end;
end;
```

As an example of its usage, the `AnonAjax` application demo has a button used to copy the content of a Web page to a Memo control (adding the requested URL at the beginning):

```
procedure TFormAnonAjax.btnReadClick(Sender: TObject);
begin
  AjaxCall (edUrl.Text,
    procedure (aResponseContent: TStringStream)
```

```
    begin
      Memo1.Lines.Text := aResponseContent.DataString;
      Memo1.Lines.Insert (
        0, 'From URL: ' + edUrl.Text);
    end);
end;
```

After the HTTP request has finished, you can do any sort of processing you want on it.

Another example would be to extract links from the HTML file (in a way that resembles the `WebFind` application covered earlier). Again, to make this function flexible, it takes as a parameter the anonymous method to execute for each link:

```
type
  TLinkCallback = reference to procedure (
    const strLink: string);

procedure ExtractLinks (strData: string;
  procLink: TLinkCallback);
var
  strAddr: string;
  nBegin, nEnd: Integer;
begin
  strData := LowerCase (strData);
  nBegin := 1;
  repeat
    nBegin := PosEx ('href="http', strData, nBegin);
    if nBegin <> 0 then
    begin
      // find the end of the HTTP reference
      nBegin := nBegin + 6;
      nEnd := PosEx ('"', strData, nBegin);
      strAddr := Copy (strData, nBegin, nEnd - nBegin);
      // move on
      nBegin := nEnd + 1;
      // execute anon method
      procLink (strAddr)
    end;
  until nBegin = 0;
end;
```

If you apply this function to the result of an AJAX call and provide a further method for processing, you end up with two nested anonymous method calls, like in the second button of the `AnonAjax` application demo:

```
procedure TFormAnonAjax.btnLinksClick(Sender: TObject);
begin
  AjaxCall (edUrl.Text,
    procedure (aResponseContent: TStringStream)
    begin
      ExtractLinks(aResponseContent.DataString,
        procedure (const aUrl: string)
        begin
```

```
          Memo1.Lines.Add (aUrl + ' in ' + edUrl.Text);
        end);
    end);
end;
```

In this case the Memo control will receive a collection of links, instead of the HTML of the returned page. A variation to the link extraction routine above would be an image extraction routine. The `ExtractImages` function grabs the source (`src`) of the `img` tags of the HTML file returned, and calls another `TLinkCallback`-compatible anonymous method (see the source code for the function details).

Now you can envision opening an HTML page (with the `AjaxCall` function), extract the image links, and use `AjaxCall` again to grab the actual images. This means using a triple-nested closure, in a coding structure that some Object Pascal programmers might find unreadable (it takes a while to get used to it!), but is certainly very powerful and expressive:

```
procedure TFormAnonAjax.btnImagesClick(Sender: TObject);
var
  nHit: Integer;
begin
  nHit := 0;
  AjaxCall (edUrl.Text,
    procedure (aResponseContent: TStringStream)
    begin
      ExtractImages(aResponseContent.DataString,
        procedure (const aUrl: string)
        begin
          Inc (nHit);
          Memo1.Lines.Add (IntToStr (nHit) + '.' +
            aUrl + ' in ' + edUrl.Text);
          if nHit = 1 then // load the first
          begin
            AjaxCall (aUrl,
              procedure (aResponseContent: TStringStream)
              begin
                // load image of the current type only
                Image1.Picture.Graphic.
                  LoadFromStream(aResponseContent);
              end);
          end;
        end);
    end);
end;
```

note This code snippet was the topic of a blog post of mine, "Anonymous, Anonymous, Anonymous" of September 2008, which attracted some comments, as you can see on: http://blog.marcocantu.com/blog/anonymous_3.html.

Beside the fact that the graphic only works in the case where you are loading a file with the same format as the one already in the Image component, the code and its result are both impressive.

Notice in particular the numbering sequence, based on the capture of the `nHit` local variable. What happens if you press the button twice, in a fast sequence? Each of the anonymous methods will get a different copy of the `nHit` counter, and they might potentially be displayed out of sequence in the list, with the second thread starting to produce its output before the first.

end.

This final section of the book has a few appendices, that focus on specific side issues worth considering, but out of the flow of the text. There is a short history of the Pascal and Object Pascal languages, a glossary, and a short section on getting started with the Delphi and Appmethod development environments.

Appendix Summary

Appendix A: The Evolution of Object Pascal

Appendix B: Glossary of Terms

Appendix C: Getting Started with the IDE for Building the Demos

Appendix D: An introduction to Object-Oriented Programming Concepts

a: the evolution of object pascal

Object Pascal is a language built for the growing range of today's computing devices, from smartphones and tablets to desktops and servers. It didn't just appear out of thin air. It has been carefully designed on a solid foundation to be the tool of choice for modern programmers. It provides an almost ideal balance between the speed of programming and the speed of the resulting programs, clarity of syntax and power of expression.

The solid foundation that Object Pascal is built upon is the Pascal family of programming languages. In the same way that Google's go language or Apple's Objective-C language are rooted in C, Object Pascal is rooted in Pascal. No doubt you would have guessed that from the name.

This short appendix includes a brief history of the family of languages and actual tools around Pascal, Turbo pascal, Delphi's Pascal, and Object Pascal. While it is not really necessary to read this to learn the language, it is certainly worth understanding the language's evolution and where it is today.

The Object Pascal programming language we use today in Embarcadero development tools was invented in 1995 when Borland introduced Delphi, which at the time

was its new visual development environment. The first Object Pascal language was extended from the language already in use in the Turbo Pascal products, where the language was generally referenced as Turbo Pascal. Borland didn't invent Pascal, it only helped make it very popular, and extend its foundations to overcome some of its limitations compared to the C language.

The following sections cover the history of the language from Wirth's Pascal to the most recent LLVM-based Delphi's Object Pascal compiler for ARM chips and mobile devices.

Wirth's Pascal

The Pascal language was originally designed in 1971 by Niklaus Wirth, professor at the Polytechnic of Zurich, Switzerland. The most complete biography of Wirth is available at `http://www.cs.inf.ethz.ch/~wirth`.

Pascal was designed as a simplified version of the Algol language for educational purposes. Algol itself was created in 1960. When Pascal was invented, many programming languages existed, but only a few were in widespread use: FORTRAN, Assembler, COBOL, and BASIC. The key idea of the new language was order, managed through a strong concept of data types, declaration of variables, and structured program controls. The language was also designed to be a teaching tool, that is to teach programming using best practices.

Needless to say that the core tenets of Wirth's Pascal have had a huge influence on the history of all programming languages, well beyond and above those still based on the Pascal syntax. As for teaching languages, too often schools and universities have followed other criteria (like job requests or donations from tool vendors) rather than looking at which language helps learning the key concepts of programming better. But that is another story.

Turbo Pascal

Borland's world-famous Pascal compiler, called Turbo Pascal, was introduced in 1983, implementing "Pascal User Manual and Report" by Jensen and Wirth. The Turbo Pascal compiler has been one of the best-selling series of compilers of all time, and made the language particularly popular on the PC platform, thanks to its

balance of simplicity and power. The original author was Anders Hejlsberg, later father of the C# language at Microsoft.

Turbo Pascal introduced an Integrated Development Environment (IDE) where you could edit the code (in a WordStar compatible editor), run the compiler, see the errors, and jump back to the lines containing those errors. It sounds trivial now, but previously you had to quit the editor, return to DOS; run the command-line compiler, write down the error lines, open the editor and jump to the error lines.

Moreover Borland sold Turbo Pascal for 49 dollars, where Microsoft's Pascal compiler was sold for a few hundred. Turbo Pascal's many years of success contributed to Microsoft eventual dropping its Pascal compiler product.

You can actually download a copy of the original version of Borland's Turbo Pascal from the *Museum* section of the Embarcadero Developer Network:

> http://edn.embarcadero.com/museum

note After the original Pascal language, Nicklaus Wirth designed the Modula-2 language, an extension of Pascal syntax now almost forgotten, which introduced a concept of modularization very similar to the concept of units in early Turbo Pascal and today's Object Pascal.

A further extension of Modula-2 was Modula-3, which had object-oriented features similar to Object Pascal. Modula-3 was even less used than Modula-2, with most commercial Pascal language development moving towards Borland and Apple compilers, until Apple abandoned Object Pascal for Objective-C, leaving Borland with almost a monopoly on the language.

The early days of Delphi's Object Pascal

After 9 versions of Turbo and Borland Pascal compilers, which gradually extended the language into the Object Oriented Programming (OOP) realm, Borland released Delphi in 1995, turning Pascal into a visual programming language. Delphi extended the Pascal language in a number of ways, including many object-oriented extensions which are different from other flavors of Object Pascal, including those in the *Borland Pascal with Objects* compiler (the last incarnation of Turbo Pascal).

note Year 1995 was really a special year for programming languages, as it saw the debut of Delphi's Object Pascal, Java, JavaScript, and PHP. These are some of the most popular programming languages still in use today. In fact, most other popular languages (C, C++, ObjectiveC, and COBOL) are much older, while the only newer popular language is C#. For a history of programming languages you can see `http://en.wikipedia.org/wiki/History_of_programming_languages`.

With Delphi 2, Borland brought the Pascal compiler to the 32-bit world, actually re-engineering it to provide a code generator common with the C++ compiler. This brought many optimizations previously found only in C/C++ compilers to the Pascal language. In Delphi 3 Borland added to the language the concept of interfaces, making a leap forward in the expressiveness of classes and their relationships.

With the release of version 7 of Delphi, Borland formally started to call the Object Pascal language the Delphi language, but nothing really changed in the language at that time. At that time Borland also created Kylix, a Delphi version for Linux, and later created a Delphi compiler for Microsoft .NET framework (the product was Delphi 8). Both projects were later abandoned, but Delphi 8 (released at the end of 2003) marked a very extensive set of changes to the language, changes that were later adopted in the Win32 Delphi compiler and all other following compilers.

Object Pascal From CodeGear to Embarcadero

With Borland unsure about its investments in development tools, later versions like Delphi 2007, were produced by CodeGear, a subsidiary of the main company. This subsidiary (or business unit) was later sold to Embarcadero Technologies, the current owner of the Delphi and C++Builder product lines (including the combined RAD Studio offerings). After that release, the company re-focused on growing and extending the Object Pascal language, adding long-awaited features like Unicode support (in Delphi 2009), generics, anonymous methods or closures, extended runtime type information or reflection, and many other significant language features (mostly covered in Part III of this book).

At the same time, along side the Win32 compiler the company introduced a Win64 compiler (in Delphi XE2) and a Mac OS X compiler, getting back to a multi-platform strategy after the attempt done earlier on Linux with the short-lived Kylix product. This time however the idea was to have a single Windows development environment and cross-compile to other platforms. The Mac support was only the beginning of

the company's multi-device strategy, embracing desktop and mobile platforms, like iOS and Android.

Going Mobile

The shift to mobile and the first Object Pascal compiler for ARM chips (as all previous platforms Delphi supported were only on Intel x86 chips) have been tied to an overall re-architecture of the compilers and the related tools (or “compiler toolchain”) based on the open LLVM compiler architecture. The ARM compiler for iOS released in Delphi XE4 was the first Object Pascal compiler based on LLVM, but also the first to introduce some new features like Automatic Reference Counting (or ARC) and a substantial “cleanup” of the string data types.

Later in the same year (2013), Delphi XE5 added support for the Android platform, with a second ARM compiler based on LLVM. To summarize, Delphi XE5, shipped with 6 compilers for the Object Pascal language (for the Win32, Win64, Mac OS X, iOS Simulator on Mac, iOS ARM, and Android ARM support). All these compilers support a largely common language definition, with a few significant differences I'll cover in details throughout the book.

In the first few months of 2014, Embarcadero released a new development tool based on the same core mobile technologies and called Appmethod. Appmethod uses the same Object Pascal compiler previously found only in Delphi. In April 2014, the company also released the XE6 version of Delphi, while September 2014 saw the third release of Appmethod and Delphi XE7.