Mastering™

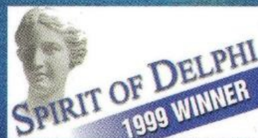
Marco Cantù

2025 Annotated Edition

Delphi™ 5

The Ultimate Delphi Resource—
for New and Experienced
Developers Alike

Written by the Winner of Borland's
1999 *Spirit of Delphi Award*



Marco Cantù

Mastering Delphi 5
2025 Annotated Edition

Original Edition: Sybex, 1995

2025 Annotated Edition: Marco Cantu, 2025

Release 0.2 – February 17th, 2025

Author: Marco Cantù

Publisher: Sybex (original edition), Marco Cantù (2025 edition)

Copyright 1995-2025 Marco Cantù, Piacenza, Italy. World rights reserved.

The author created example code in this publication expressly for the free use by its readers. Source code for this book is copyrighted freeware, distributed via a GitHub project listed in the book and on the book's web site. The copyright prevents you from republishing the code in print or electronic media without permission. Readers are granted limited permission to use this code in their applications, as long at the code itself is not distributed, sold, or commercially exploited as a stand-alone product.

Aside from the above exception concerning the source code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, either in the original or in a translated language, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Delphi is a trademark of Embarcadero Technologies (a division of Idera, Inc.). Other trademarks are of the respective owners, as referenced in the text. Whilst the author and publisher have made their best efforts to prepare this book, they make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Mastering Delphi 5 2025 Annotated Edition

This PDF version is a draft dated February 17th, 205

The electronic edition of this book is freely distributed by the author, but doesn't further distribution. *Do not distribute the PDF version of this book without permission from the author.*

More information at http://www.marcocantu.com/md52025

To my wife, Lella, the love of my life[1]

1 I've kept the dedication of the book, as it was originally. In fact, that is still true today!

Preface To The 2025 Commented Edition

As you know I wrote several Mastering Delphi books over the course of the years. I thoughts a few times about writing a new one... but the task is fairly daunting, given Delphi (as an IDE and considering the libraries and target platforms it now supports) has dramatically grown in size and complexity, and you'd now need several thousand pages to cover the product adequately, and not even in depth. While I have several draft of my older books, it turns out Mastering Delphi 5 is the oldest one I have in an electronic version with images and proper formatting. A few years back, I acquired the rights of this edition from my original publisher (Sybex, now part of Wiley) and considering a new edition I asked a person to reformat the text, import the images, and turn this into a complete volume.

That was a few years back. More recently, I found this edited and formatted manuscript, and decided to make it public rather than keeping it on my hard drive. The

text of the book is, with minor and limited changes, the original text covering version 5 of Delphi, released in 1999.

This is not a book on recent versions of Delphi: A few of the sections are clearly dated, but most of the core content covering the key features of the product is still actual today. However, publishing it as is would have been of very limited use and possibly confusing. Therefore I've made two primary changes to the book. First I've captured some updated images of the IDE and of the running applications. I've kept some of the original images alongside, though, mixing the old and the new. The different is so striking I don't even need to call them out. Second I've added a large number of footnotes to underline new features, significant changes, code I'd write differently, assertions that are no longer true. I haven't rewritten the text, as this would have been way more time consuming, but I've pointed out many facts, giving ideas and suggestions for further study, or just providing some tidbits and facts, along with many links to additional information available online. I've used footnotes to reduce the impact on the existing text, compared to adding notes in the text flow or doing direct edits.

But you might still wonder, who is this book for? Although it might appeal them, *this is not only for the nostalgic*, although some of the old timers might find it interesting to read it. *It is for anyone who wants to understand Delphi*. Even covering the product how it was many years ago, **this book helps understanding all of Delphi's core concepts**.

You might be wondering if this is possible because Delphi is an old product. This is certainly *not* the case. It underlines the fact the product has a great history, but also that it has kept and keeps evolving in a fantastic way while maintaining its core tenets and offering an *unparalleled degree of compatibility in the development tools space*. The fact that most of the code in this old book can be compiled and run today, producing modern looking Windows 11 applications is a testament of the power of Delphi.

This preface is the only new section of the book. From now on, this is the old book with my comments and annotation. Have a good reading!

Acknowledgments

This incarnation of *Mastering Delphi* marks the fifth year of the Delphi era[2]. As it has for many other programmers, Delphi has been my primary interest throughout these years; and writing, consulting, teaching, and speaking at conferences about Delphi have absorbed more and more of my time, leaving other languages and programming tools in the dust of my office[3]. Because my work and my life are quite intertwined, many people have been involved in both, and I wish I had enough space and time to thank them all as they deserve. Instead, I'll just mention a few particular people and say a warm "Thank You" to the entire Delphi community (also for the Spirit of Delphi 1999 Award I've been happy to share with Bob Swart).

The first official thanks are for the Borland programmers and managers[4] who made Delphi possible and continue to improve it: Chuck Jazdzewski, Danny Thorpe[5], Eddie Churchill, Allen Bauer, Steve Todd, Mark Edington, Jim Tierney, Ravi

---

2 I made further editions of Mastering Delphi for Delphi 6, Delphi 7, and Delphi 2005 with the same publisher. Later I moved to self publishing and started the "Delphi 20xx Handbook" series, focused on specific new features of the given version of Delphi, rather than providing the broad overview of the Mastering Delphi volumes. That's one of the reasons for this new "reprint" of Mastering Delphi 5. You can find more on my web site www.marcocantu.com.

3 A few years ago I ended up accepting a Product Manager position at Embarcadero (now part of Idera Inc.), the company who owns Delphi. So my focus on Delphi continues to be a full time focus even today, although with a different perspective. I have used Delphi for 30 years and continue to do so. My knowledge of the technologies behind the product has grown during the years I've been working for Embarcadero.

Kumar, Jörg Weingarten, Anders Ohlsson, and all the others I have not had a chance to meet. I'd also like to give particular mention to my friends Ben Riga (the current Delphi product manager), Charlie Calvert, John Kaster, and David I (all three at Borland Developer's Relations). I cannot forget the help I received from Zack Urlocker and Nan Borreson.

The next thanks are for the Sybex editorial and production crew, many of whom I don't even know. Special thanks go to Denise Santoro, Jim Compton, and Diane Lowery for their editorial acumen; I'd also like to thank Richard Mills, Kristine O'Callaghan, Maureen Forys, Teresa Trego, Jennifer Campbell, Carol Iverson, and Tony Jonick.

This edition of *Mastering Delphi* has had an incredibly picky and detailed review from Delphi team member Danny Thorpe. His highlights and comments have improved the book in all areas: technical content, accuracy, examples, and even readability. Thanks a lot. Past editions of the book also had special contributions: Tim Gooch worked on Part V for Mastering Delphi 4 and Giuseppe Madaffari contributed a lot of database material for this and the last edition. Many improvements to the text and sample programs were suggested by technical reviewers of the past editions (Juancarlo Añez, Ralph Friedman, Tim Gooch, and Alain Tadros) and from other reviews done over the years by Bob Swart, Giuseppe Madaffari, and Steve Tendon.

Special thanks go to my friends Bruce Eckel, Andrea Provaglio, Norm McIntosh, Johanna and Phil of the BUG-UK, Ray Konopka, Mark Miller, Cary Jensen, Chris Frizelle of *The Delphi Magazine,* Foo Say How, John Howe, Mike Orriss, Chad "Kudzu" Hower, Dan Miser, and Marco Miotti. Also, a very big "Thank You" to all the attendees of my Delphi programming courses, seminars, and conferences in Italy, the United States, France, the United Kingdom, Singapore, the Netherlands, Germany, and Sweden.

Aside from the people involved with Delphi, my biggest thanks go to my patient wife, Lella, who (while carrying a child[6]) had to spend another summer with little vacation, as the book always took more time than I expected. Many of our friends provided healthy breaks in the work: Sandro and Monica with Luca, Stefano and

---

4 Needless to say none of this developers and managers work at Embarcadero any more, after 25 years. I've kept this Acknowledgments section as it was, despite the changes to my like and that of all of the people mentioned here.

5 Unfortunately we lost Danny a few years back. Danny was the technical reviewer of the original edition of this book, Mastering Delphi 5, and I've long been in touch with him after he left the company.

6 That child is now a young adult. We also have a second one, who's also grown up.

Elena, Marco and Laura with Matteo, Bianca, Chiara, Luca and Elena, Chiara and Daniele with Leonardo, Laura, Vito and Marika with Sofia. Our parents, brothers, sisters, and their families were very supportive, too. It was nice to spend some of our free time with them and our six nephews, Matteo, Andrea, Giacomo, Stefano, Andrea, and Pietro.

Finally, I would like to thank all of the people, many of them unknown, who enjoy life and help to build a better world. If I never stop believing in the future and in peace, it is also because of them.

Introduction

The first time Zack Urlocker[7] showed me a yet-to-be-released product code-named Delphi, I realized that it would change my work—and the work of many other software developers. I used to struggle with C++ libraries for Windows, and Delphi was and still is the best combination of object-oriented programming and visual programming for Windows.

Delphi 5 simply builds on this tradition and on the solid foundations of the VCL to deliver another astonishing and all-encompassing software development tool. Looking for database, client/server, multi tier, intranet, or Internet solutions? Looking for control and power? Looking for fast productivity? With Delphi 5 and the plethora of techniques and tips presented in this book, you'll be able to accomplish all this[8].

---

7 Zack was the first Delphi Product Manager, and made a career in many other management positions including at MySQL and, more recently, several startups.

8 Most of the features discussed here are still valid in the latest versions of Delphi, even if they represent a subset of the available features. A few have been discontinued or are not recommended any more, and this will all be covered in footnotes.

Five Versions and Counting

Some of the original Delphi features that attracted me were its form-based and object-oriented approach, its extremely fast compiler, its great database support, its close integration with Windows programming, and its component technology. But the most important element was the Object Pascal language, which is the foundation of everything else.

Delphi 2 was even better! Among its most important additions were these: the Multi-Record Object and the improved database grid, OLE Automation support and the variant data type, full Windows 95 support and integration, the long string data type, and Visual Form Inheritance. Delphi 3 added to this the Code Insight technology, DLL debugging support, component templates, the TeeChart, the Decision Cube[9], the Web Broker technology, component packages, ActiveForms, and an astonishing integration with COM, thanks to interfaces.

Delphi 4 gave us the AppBrowser editor, new Windows 98 features, improved OLE and COM support, extended database components, and many additions to the core VCL classes, including support for docking, constraining, and anchoring controls. There are a great many new features in Delphi 4, as you can still discover by reading this book if you missed the last edition.

Delphi 5 adds to the picture many more improvements of the IDE (too many to list here), extended database support (with specific ADO and InterBase datasets), an improved version of MIDAS[10] with Internet support, the TeamSource version-control tool[11], translation capabilities, the concept of frames, many new components, and much more, as you'll see in the following pages.

Delphi is a great tool, but it is also a complex programming environment that involves many elements. This book will help you master Delphi programming, including the Object Pascal language, Delphi components (both using the existing ones and developing your own), database and client/server support, the key elements of Windows and COM programming, and Internet and Web development.

---

9 The Decision Cube is a feature that was later dropped form the product.

10 MIDAS was later turned into DataSnap and some of the related technologies are still around, even if the world of multi-tier development and web services has changed a bit since the early days. Today's multi-tier solutions tend to use the REST architecture, which is true for RAD Server, the current multi-tier technology in Delphi.

11 More recent versions of Delphi have added support for modern version control systems, including Subversion and Git.

You do not need an in-depth knowledge of any of these topics to read this book, but you do need to know the basics of Pascal programming. Having some familiarity with Delphi will help you considerably, particularly after the introductory chapters. The book starts covering its topics in depth immediately; much of the introductory material from previous editions has been removed. Some of this left-out material and an introduction to Pascal[12] is available on the author's Web site and can be a starting point if you are not confident with Delphi basics. Each new Delphi 5 feature is covered in the relevant chapters throughout the book.

The Structure of the Book

The book is divided into five parts:

- Part I, "Delphi and Object Pascal," introduces new features of the Delphi 5 Integrated Development Environment (IDE) in Chapter 1 and then moves to the Object Pascal language and the Visual Component Library (VCL), providing both foundations and advanced tips[13].
- Part II, "Using Components," covers standard components, Windows common controls, graphics, menus, dialog, scrolling, docking, multiple-page controls, Multiple Document Interface, and many other topics[14].
- Part III, "Writing Database Applications," covers plain database access, advanced Paradox topics, in-depth coverage of the data-aware controls, client/server programming, InterBase Express, and ADO[15].

---

12 This material on the Pascal language later turned into the Essential Pascal e-book, but it is also included in my "Object Pascal Handbook", a book I'm maintaining up to date over time. It is available as a PDF via Embarcadero and as a printed book on Amazon, see www.marcocantu.com/objectpascalhandbook/ for more information.

13 While the IDE changed considerably a fair number of the techniques and tips still applied today. Also the core of the language remains the same, even if additions have been relevant. So most of the content of Part I is quite relevant.

14 The core techniques related with using components have not changed at all. So these foundation chapters provide a very good introduction to Delphi, even after many years. I'll mention changes in notes, like on all other chapters, of course.

15 The database part of the product has seen many significant changes, with the demise of Paradox and the introduction of the dbExpress library and later the migration to FireDAC. ADO components are still available and the core classes in the DB.pas unit did not change that much.

- Part IV, "Components and Libraries," covers Delphi component and Dynamic Link Library (DLL) development; it then looks at COM and OLE, covering Windows shell extensions, OLE Automation, and ActiveX development[16].
- Part V, "Real-World Delphi Programming," discusses many common programming techniques, such as multithreading, memory handling, debugging, using resources, printing support, file handling, programming TCP/IP sockets, Internet development, Web server-side extensions, three-tier architectures, and distributed database applications build upon the MIDAS technology[17].

As this brief summary suggests, the book covers topics of interest to Delphi users at nearly all levels of programming expertise, from "advanced beginners" to component developers.

In this book, I've tried to skip reference material almost completely and focus instead on techniques for using Delphi effectively. Because Delphi provides extensive online documentation, to include lists of methods and properties of components in the book would not only be superfluous, it would also make it obsolete as soon as the software changes slightly. I suggest that you read this book with the Delphi help files at hand, to have reference material readily available. You can find some more Delphi reference material on my Web site, as described later.

However, I've done my best to allow you to read the book away from a computer if you prefer. Screen images and the key portions of the listings should help in this direction. The book uses just a few conventions to make it more readable. All the source code elements, such as the keywords, the names of properties, classes, and functions appear in `this font`, and listings are formatted as they appear in the Delphi editor, with boldfaced keywords and italic comments and strings.

---

16 The section on components and libraries is still surprisingly up-to-date, as the COM layer in Windows is still there and Delphi's support saw limited improvements (as it was already very good and Microsoft didn't touch COM for many years, focusing on the newer .NET framework).

17 While some core techniques like multi-threading are still based on the same foundations, most of what relates to Web development saw significant improvements. Still WebBroker is still an architecture heavily used today.

Free Source Code on the Web

This book focuses on examples. After the presentation of each concept or Delphi component, you'll find a working program example (sometimes more than one) that demonstrates how the feature can be used. All told, there are more than 200 examples presented in the book[18]. Most of the examples are quite simple and focus on a single feature. More complex examples are often built step-by-step, with intermediate steps including partial solutions and incremental improvements.

note Some of the database examples also require you to have the Delphi sample database DBDEMOS installed; it is part of the default Delphi installation.

Besides the archive with the minimal source code files required to build the programs, a second archive includes an HTML version of the source code, with full syntax highlighting, along with a complete cross-reference of keywords and identifiers (class, function, method, and property names, among others). The cross-reference is an HTML file, so you'll be able to use your browser to easily find all the programs that use a Delphi keyword or identifier you're looking for.

The directory structure of the downloaded files is quite simple. Basically, each part of the book has its own folder, with a subfolder for each chapter, and a further subfolder for each example (e.g., `Part2\06\Borders`). In the text, the examples are simply referenced by name (e.g., Borders).

note Be sure to read the source code archive's Readme file, which contains important information about using the software legally and effectively.

How to Reach the Author

If you find any problems in the text or examples in this book, I would be happy to hear from you. Besides reporting errors and problems, please give us your unbiased

18 I have not updated or modified the source code demos, although I might do it in the future. I'll occasionally point out to code that needs an update. The source code demos are available on more modern repositories, like github.com/MarcoDelphiBooks/MasteringDelphi5.

opinion of the book and tell us which examples you found most useful and which you liked least. There are several ways you can provide this feedback[19]:

- My own Web page (`www.marcocantu.com`) hosts news and tips, technical articles, the free online book “Essential Pascal,” Delphi 5 reference information we could not fit in this book, Delphi links, and my collection of Delphi components and tools.
- Finally, you can reach me via e-mail at `marco@marcocantu.com`. My mailbox is usually quite full and, regretfully, I cannot reply promptly to every request. Please write to me in English or Italian.

19 Here I've made an exception to the “no edits” rule and removed referenced to the original publisher, Sybex. My contact information is still valid, but some is missing like my blog (blog.marcocantu.com).

Chapter I: Delphi And Object Pascal

In a visual programming tool such as Delphi, the role of the environment is at times even more important than the programming language. Delphi 5 provides many new features in its visual development environment, and this chapter covers them in detail. This chapter isn't a complete tutorial but mainly a collection of tips and suggestions aimed at the average Delphi user. In other words, it's not for newcomers. I'll be covering the new features of the Delphi 5 Integrated Development Environment (IDE) and some of the advanced and/or little-known features of previous versions as well, but in this chapter I won't provide a step-by-step introduction. Throughout this book I'll assume you already know how to carry out the basic hands-on operations of the IDE, and all the chapters after this one focus on programming issues and techniques.

If you *are* a beginning programmer, don't be afraid. The Delphi Integrated Development Environment is quite intuitive to use. Delphi itself includes a manual (available in Acrobat format on the Delphi CD[20]) with a tutorial that introduces the

20 There is no Delphi CD any more, but you can find tutorials and documentation at docwiki.embarcadero.com/RADStudio/.

development of Delphi applications. You can also find a step-by-step introduction to the Delphi 5 IDE on my Web site, `www.marcocantu.com`. The short online book *Essential Pascal*[21] is based on material from the first chapters of earlier editions of *Mastering Delphi*.

Editions of Delphi 5

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single edition of Delphi 5; there are many of them. Second, any Delphi environment can be customized. For these reasons, Delphi screens you see illustrated in this chapter may differ from those on your own computer. Here are the current editions of Delphi:

- The basic version (the "Standard" edition) is aimed at Delphi newcomers and casual programmers.[22]
- The second level (the "Professional" edition) is aimed at professional developers. It includes all the basic features, plus database programming support, extensive Web server support (WebBroker), and some of the external tools. This book generally assumes you are working with at least the Professional edition.
- The full-blown Delphi (the "Enterprise" edition, previously called the "Client/ Server Suite") is aimed at developers building enterprise applications. It includes SQL Links for native Client/Server BDE connections, ADO and InterBase Express components, support for multiuser applications, internationalization, and three-tier architecture, and many other tools, including the SQL Monitor. Some chapters cover features included only in Delphi Enterprise; these sections are specifically identified.

---

21 See www.marcocantu.com/epascal/ for the latest information about this e-book.

22 The "Standard" edition of Delphi has been long discontinued. It was temporarily replaced by a Turbo edition (now discontinued as well). Later the company introduced a new low cost version called Starter edition. Today you can use the free Delphi Community Edition (if you quality in terms of use case and earnings) or buy the "Professional" and "Enterprise" versions, which continue to be the core offerings, with differences not radically changed since Delphi 5. Check the latest product description and Feature Matrix at www.embarcadero.com/products/delphi for information on differences between the versions, so you can download or buy the one that better serves your needs.

Some of the features of Delphi Enterprise are available as an "upsell" to owners of Delphi Professional. Although this is a marketing decision that may change in the future, you should be able to buy ADO components and TeamSource (for cooperation among programmers). If you can't justify the cost of the full Enterprise edition for your work, you may be able to buy Delphi Professional plus the specific subsystems you want separately from the Borland Online Store[23].

Besides the different editions available, there are a number of ways to customize the Delphi environment. In the screen illustrations throughout the book, I've tried to use a standard user interface (as it comes out of the box); however, I have my preferences, of course, and I generally install many add-ons, which might be reflected in some of the screen shots.

The Delphi 5 IDE

The Delphi 5 IDE includes some of the broadest changes Borland has introduced since it upgraded Delphi 1 to Delphi 2. Among the new features are a redesigned Object Inspector, a new Project Manager, the ability to save the position of the desktop windows, the to-do list, and much more. Most of the features are quite easy to grasp, but it's worth examining them with some care so that you can start using Delphi 5 at its full potential.

Command-Line Options

The first thing to notice is that there are changes even before you start Delphi. In fact, the delphi32.exe program[24], which starts the IDE, has many new command-line options. Most of these options (listed in the Help topic "IDE command-line options"[25]) are aimed at advanced users and allow you to track the status of the Delphi IDE itself.

---

23 Extra add-ins are currently not sold separately any more.

24 The executable file that starts the IDE is now called "*bds.exe*" (which originally was a short version of Borland Developer Studio), some of the command lines parameters mentioned here still work and little known by developers.

25 See docwiki.embarcadero.com/RADStudio/en/IDE_Command_Line_Switches_and_Options

For example, you can load a program into the debugger or attach the system to a process that's already running (topics I'll discuss along with other debugging features in Chapter 18).

Other features might be useful even to the casual programmer. The `-ns` ("no splash") flag skips the splash screen, and the `-np` ("no project") flag tells Delphi not to open an empty project on startup. (This allows for a fast boot because it prevents the loading of any package of components, which are attached to projects.)

Probably the most commonly used feature isn't strictly a command-line option, or even a startup option. You can easily specify a project, project group, or Pascal source code file to open. When Delphi is already running, double-clicking a filename or icon in Windows Explorer doesn't open a new copy of the IDE, it simply opens a PAS or DFM file in the current copy of Delphi. When you select a project file (.DPR), Delphi first closes the current project after asking you to save any changes[26].

From the command line you can load a project and let Delphi automatically build or make it (with the `-b` and `-m`) options, immediately closing the IDE after the operation is completed. This doesn't seem terribly useful; for compiling a series of large projects with a script or batch file, you should instead use the faster command-line compiler[27] (which doesn't need the IDE).

Saving the Desktop Settings

Building on past versions of Delphi and on the support for docking that was added in Win32, Delphi has since version 4 allowed programmers to customize the IDE in a number of ways, typically opening many windows and arranging them and docking them to each other. However, programmers often need to open one set of windows at design time and a different set at debug time. Similarly, programmers might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

With Delphi 5, every time you come up with an arrangement of IDE windows you like for a specific purpose, you can save it with a name and restore it easily. Also, you can make one of these groupings your default debugging setting, so that it will

26 This behavior has changed: As you activate a project in Explorer, the IDE by default adds the project to the current project group, rather than replacing the currently open project.

27 Starting from Delphi 2005 the command line compilation can also be invoked using a MS-Build script, which is what the IDE does anyway. Compiling outside of the IDE is also directly available as a compiler project option.

be restored automatically when you start the debugger. All these features are available in the new Desktops toolbar, shown in Figure 1.1. (It's the only toolbar with a combo box.) You can also work with desktop settings using the View ➢ Desktops menu. This has the features of the toolbar, and also allows you to delete one of the saved settings[28].

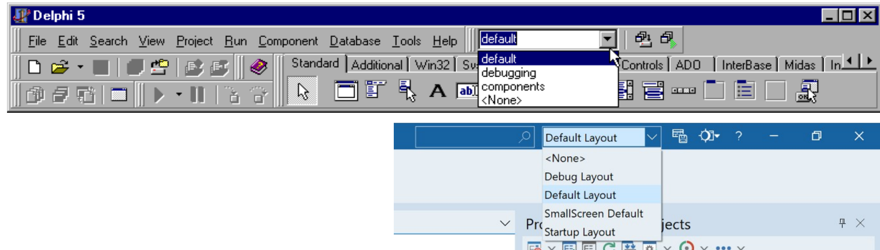


Figure 1.1: The main window of Delphi 5 includes the Desktops toolbar, which you can use to reload a configuration of the IDE windows. Images captured in Delphi 5 and Delphi 12.

Desktop setting information is saved in DST files[29], which are INI files in disguise. The saved settings include the position of the main window, the Project Manager, the Alignment Palette, the Object Inspector (including its new property category settings), the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here is a small excerpt from a DST file, which should be easily readable:

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78

[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1
```

28 While the UI has changed the same idea remains today, with the addition of a new default desktop settings called "Startup Layout", used when no project is open.

29 In recent versions, the DST files are saved in the folder with the version number under C:\Users\<username>\AppData\Roaming\Embarcadero\BDS\xxx. The file content remains largely the same.

```
[AlignmentPalette]
Create=1
Visible=0
...

[PropertyInspector]
Create=1
Visible=1
...
Dockable=1
SplitPos=85
ArrangeBy=Name
HiddenCategories=Legacy
ShowStatusBar=1
```

note Desktop settings override project settings. This helps eliminate the problem of moving a project between machines (or between developers) and having to rearrange the windows to your liking. Delphi 5 separates per-user and per-machine preferences from the project settings, to better support team development.

The To-Do List

Another brand-new feature of Delphi 5's IDE is the to-do list[30]. This is a list of tasks you still have to do to complete a project, a collection of notes for the programmer (or programmers, as this tool can be very handy in a team). While the idea is not new, the key concept of the to-do list in Delphi 5 is that it works as a two-way tool.

In fact, you can add or modify to-do items by adding special comments to the source code of any file of a project; you'll then see the corresponding entries in the list. But you can also visually edit the items in the list to modify the corresponding source code comment. For example, here is how a to-do list item might look like in the source code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // TODO -oMarco: Add creation code
end;
```

The same item can be visually edited in the window shown in Figure 1.2.

30 While superseded by modern developer collaboration tools for tracking changes and work, the To-Do list has still a nice role and I think it has been a bit neglected, while I find it handy to leave notes for myself and occasionally for others using this format rather than using a general comment, as the IDE makes it easier to find them.

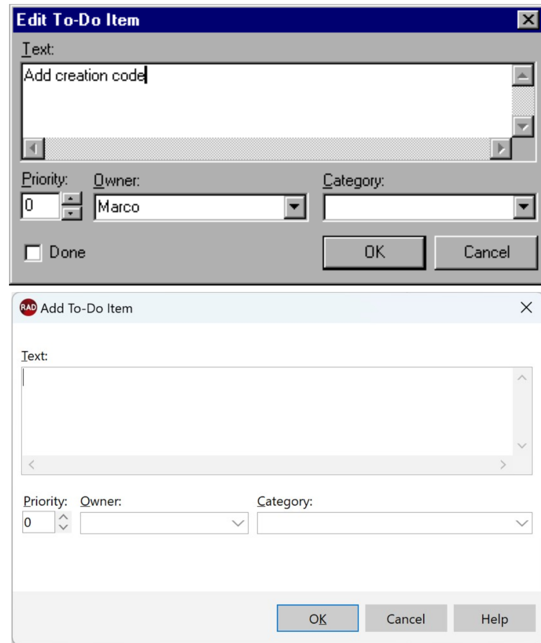


Figure 1.2: The Edit To-Do Item window can be used to modify a to-do item, an operation you can also do directly in the source code. Images captured in Delphi 5 and Delphi 12.

The exception to this two-way rule is the definition of project-wide to-do items. You must add these items directly to the list. To do that, you can either use the Ctrl+A key combination in the To-Do List window or right-click in the window and select Add from the shortcut menu. These items are saved in a special file with the .TODO extension.

There are multiple options you can use with a `TODO` comment. You can use `-o` (as in the code excerpt above) to indicate the owner, the programmer who entered the comment; the `-c` option to indicate a category; or simply a number from 1 to 5 to indicate the priority (0, or no number, indicates that no priority level is set). For example, using the Add To-Do Item command on the editor's shortcut menu (or the Ctrl+Shift+T shortcut[31]) generated this comment:

```
{ TODO 2 -oMarco : Button pressed }
```

Delphi treats everything after the colon, up to the end of line or the closing brace, depending on the type of comment, as the text of the to-do item.

---

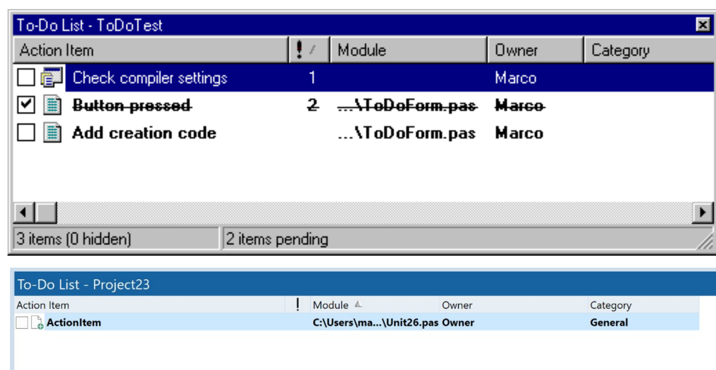
31 The shortcut still opens the Edit To-Do item dialog above, but you can also type "*todo*" in the editor and press space to trigger the generation of this line:

```
{TODO -oOwner -cGeneral : ActionItem}
```

Finally, in the To-Do List window you can check off an item to indicate that it has been done. The source code comment will change from `TODO` to `DONE`. You can also change the comment in the source code manually to see the check mark appear in the To-Do List window.

One of the most powerful elements of this architecture is the main To-Do List window, shown in Figure 1.3, which can automatically collect to-do information from the source code files as you type them. The items of this list are part of this chapter's ToDoTest example (which does nothing but has lots of things to do). The list items in this window show the various attributes I've just described, along with the source code files where they are defined. The initial check box is marked for Done items, which also have their text crossed out.

Figure 1.3: The To-Do List window for the ToDoTest example. Images captured in Delphi 5 and Delphi 12.



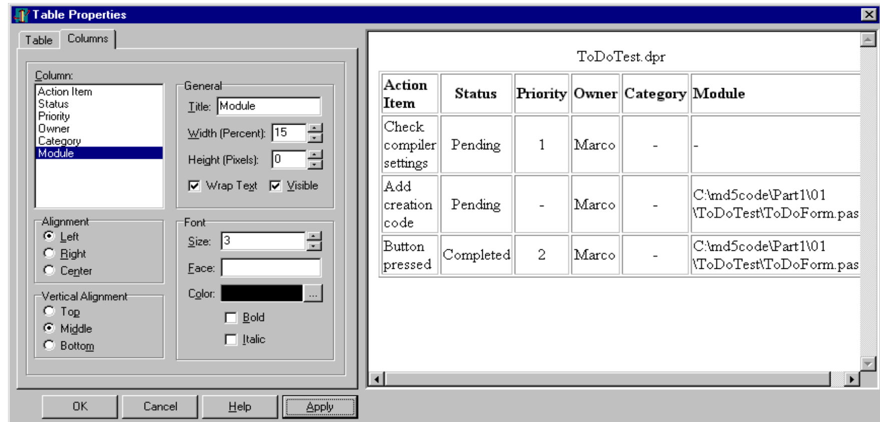
note To try out ToDoTest and all the program examples in this book, you need to download the source code[32]. Every reader should download the source code in order to get the full value of this book. Each time the text mentions a new program example by name, you should look for a folder of that name among the downloaded files and read the complete source code. For most examples you'll also want to compile the program and run it.

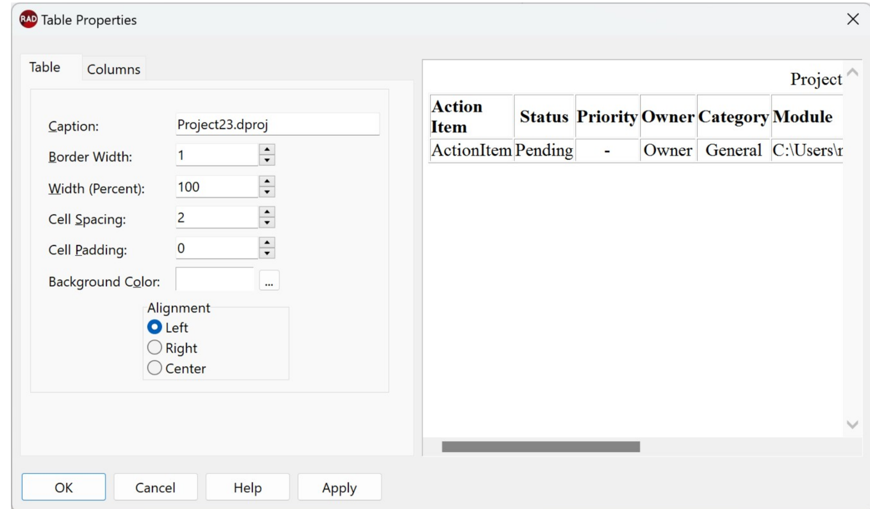
The To-Do List window has a shortcut menu that allows you to add, edit, or delete items, filter and sort them, and export them to the Clipboard. The command used to perform this last operation, Copy As, lets you export the items either as text or as an HTML table, which can be customized using the Table Properties command. The HTML table settings include a nice preview, as you can see in Figure 1.4. The information is not saved in an HTML file; it's just copied to the Clipboard. You have to

32 In this case, I've deleted portions of the text as the old locations don't exist any more. Again, the correct link is github.com/MarcoDelphiBooks/MasteringDelphi5.

open your favorite HTML editor (or Notepad or a text window in the Delphi editor) to save it to a file.[33]

Figure 1.4: The HTML table preview of the to-do list. Images captured in Delphi 5 and Delphi 12.





33 I realized I had not seen that HTML preview in so many years, I thought the feature had been dropped, but – as Figure 1.4 shows – it's still in the most recent versions of Delphi.

The AppBrowser Editor

The editor included with Delphi 5 hasn't changed much from Delphi 4. However, Delphi 4 had many new features, so it's worth briefly examining this tool. Delphi 4 introduced three fundamental innovations: a Code Explorer window (which lists all the definitions of a unit), support for navigation (similar to that of a Web browser), and Class Completion (a code-generation technology).

Delphi 5 adds to the editor a new keyboard mapping for Visual Studio emulation and the ability to extend the editor with custom key mapping modules. These last settings are defined in the new Key Bindings tab of the Editor Properties dialog box, which you can activate with the Tools ➢ Editor Options[34] command. This new dialog box shows the environment settings related to the editor.

---

note The custom key mapping modules can be written using new Tools API features added to Delphi 5. You can write a completely new key mapping module or simply add a few extra shortcut keys to the existing one. This advanced topic is not covered in the book, but you can find examples in the Editor Keybinding folder of Delphi's Demos directory. One of these additional key bindings, called Buffer List, is installed by default and available by pressing the Ctrl+B key combination.

The Delphi editor allows you to work on several files at once, using a "notebook with tabs" metaphor, and you can also open multiple editor windows[35]. You can jump from one page of the editor to the next by pressing Ctrl+Tab (or Shift+Ctrl+Tab to move in the opposite direction). There are a number of options that affect the editor, located in the new Editor Properties dialog box. You have to go to the Preferences page of the Environment Options[36] dialog box, however, to set the editor's AutoSave feature, which saves the source code files each time you run the program (preventing data loss in case the program crashes badly).

I won't discuss the various settings of the editor, as they are quite intuitive and are described in the online Help. What is not officially documented is that you can use two entries of the Windows Registry to set the initial width and height of the editor[37]

34 This is now found in the Editor section of the Tools | Options dialog box.

35 Starting with very recent versions of Delphi, you can also use "split views" which is the ability to slit an editor horizontally or vertically to see more than one file, but also see two different locations of the same file side by side. *I like this "split views" new feature a lot!*

36 Now under Tools | Options. I won't keep adding footnotes for each occurrence, it's a general changes how options are now surfaced I a single all-encompassing dialog box.

37 This entire concept doesn't exist any more, given the editor is now docked to the main IDE window.

(to make it as large as your screen, for example). Go to the Delphi section in the Registry[38], `HKEY_CURRENT_USER/Software/Borland/Delphi/5.0`, and add under the `Editor` key two new `DWORD` items, called `DefaultHeight` and `DefaultWidth`, indicating the height and width of the editor in pixels. To modify the Windows Registry you can use the RegEdit.EXE application under Windows 95 and 98 or RegEdt32.EXE under NT[39].

Another tip to remember is that beginning with Delphi 4, using Cut and Paste commands is not the only way to move source code. You can also select and drag words, expressions, or entire lines of code. You can also copy text instead of moving it, by pressing the Ctrl key while dragging.

The Code Explorer

The *Code Explorer* window[40], which is generally most useful when it's docked on the side of the editor, simply lists all of the types, variables, and routines, defined in a unit, plus other units appearing in `uses` statements. For complex types, such as classes, the Code Explorer can list detailed information including a list of fields, properties, and methods. All the information is updated as soon as you start typing in the editor. You can use the Code Explorer to navigate in the editor. If you double-click one of the entries in the Code Explorer, the editor jumps to the corresponding declaration.

While all that is quite obvious after you've used Delphi for a few minutes, there are some features of the Code Explorer that are not so intuitive. One important point is that you have full control of the layout of the information, and you can reduce the depth of the tree usually displayed in this window by customizing the Code Explorer. Collapsing the tree can help you make your selections more quickly. You can configure the Code Explorer by using the corresponding page of the Environment Options[41], as shown in Figure 1.5.

---

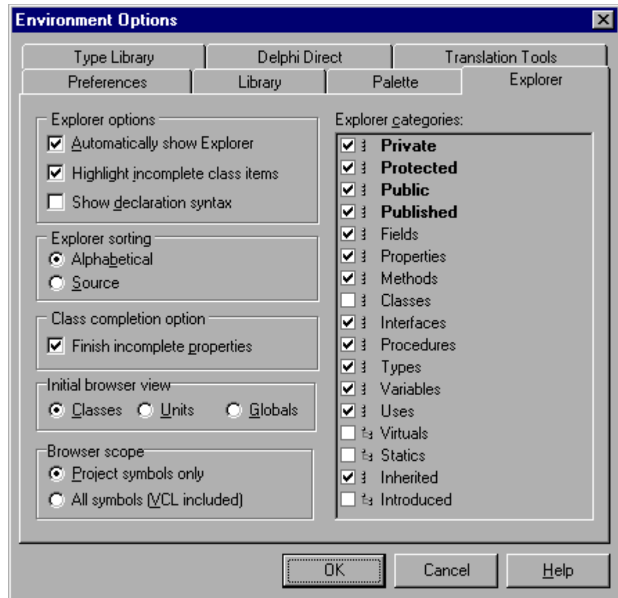
38 In recent releases, that's a key under Computer\HKEY_CURRENT_USER\Software\Embarcadero\BDS\23.0 or similar (depending on the internal product version number).

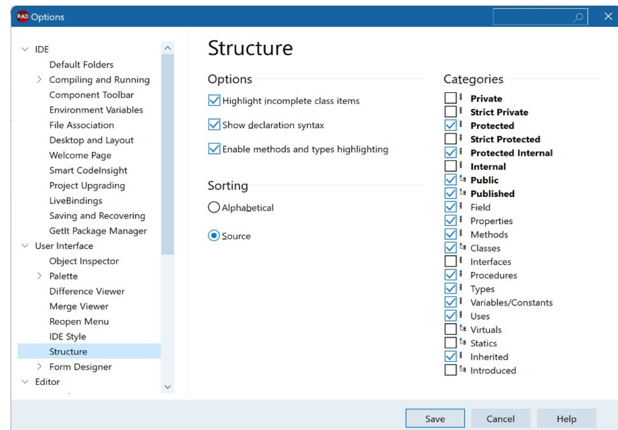
39 Today, it's just called *regedit.exe*.

40 The content of what was the Code Explorer windows now displayed in the Structure view in case a source code file is open in the editor (while the same pane doubles as a form layout view when a designer is selected). The Code Explorer pane has now some more information, including Error Insight, the list of errors in the given unit.

41 In recent versions these settings are available under Tools | Options, User Interface, Structure.

Figure 1.5:
You can configure the Code Explorer in the Environment Options dialog box. Images captured in Delphi 5 and Delphi 12: The content it surprisingly similar.





Notice that when you deselect one of the Explorer Categories items on the right side of this page of the dialog box, the Explorer doesn't remove the corresponding elements from view, it simply adds the node in the tree. For example, if you deselect the Uses check box, Delphi doesn't hide the list of the used units from the Code Explorer. On the contrary, the used units are listed as main nodes instead of being kept in the Uses folder. As another example, by disabling the Types, Classes, and Variables selections, you obtain the output shown in Figure 1.6.

The most important settings are probably those related to classes. The definitions related to a class can be arranged in three ways:

- According to the private, protected, public, and published categories
- According to the methods and fields categories
- All together in a single group

As each item of the Code Explorer tree has an icon marking its type, arranging by field and method seems less important than arranging by access specifier. My preference is to show all items in a single group, as this requires the fewest mouse clicks to reach each item. Selecting items in the Code Explorer, in fact, provides a very handy way of navigating the source code of a large unit. When you double-click on a method in the Code Explorer, the focus moves to the definition in the class declaration (in the interface portion of the unit). You can use the Ctrl+Shift combination with the up or down arrow keys to jump from the definition of a method or procedure in the interface portion of a unit to its complete definition in the implementation portion (or back again).

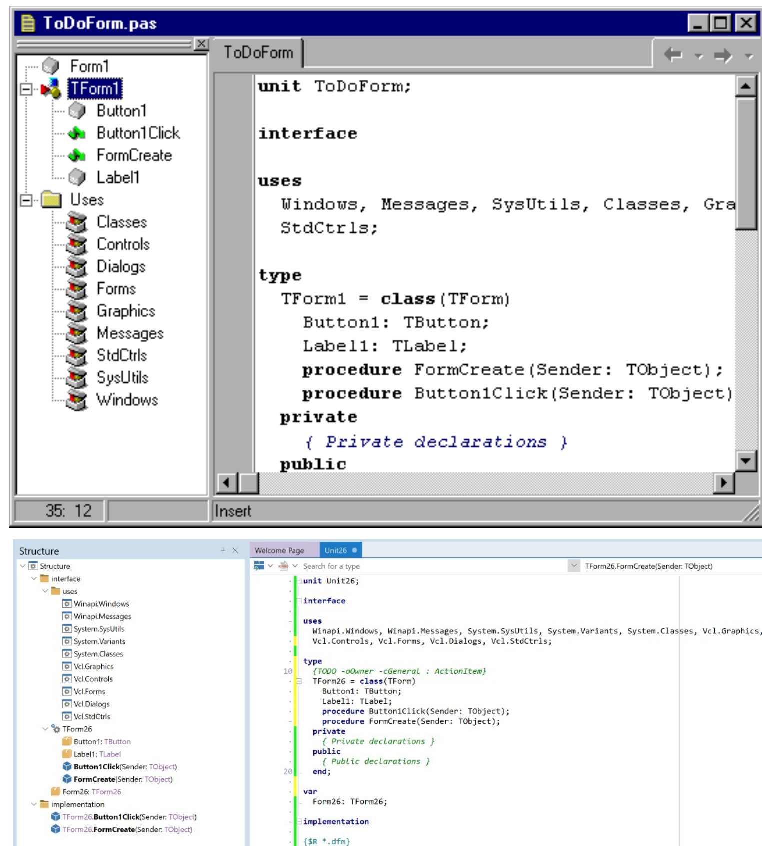


Figure 1.6: Some of the folders of the Code Explorer can be removed by removing the items from the corresponding settings. Images captured in Delphi 5 and Delphi 12.

note Some of the Explorer Categories shown in Figure 1.5 are used by the new Project Explorer (or Browser) introduced in Delphi 5, rather than by the Code Explorer. These include, among others, the Virtuals, Statics, Inherited, and Introduced grouping options.

The Code Explorer is not only an output and browsing tool. In fact, you can use it for entering new items in each category. Actually, the type of the new item generally depends on what you type. A name that starts with the `procedure` or `function` keywords is automatically considered a method, while a name followed by a semicolon and a data type is considered a field. The editing capabilities of the Code Explorer are too limited to provide any real advantage compared to editing in the source-code window. It would be nice to have dragging capabilities, for example, to move a field or method to a different visibility section or copy it to another class.

note *Field*, *methods*, *public*, *private*…? If you're not familiar with the terminology of the Object Pascal language, you'll find good coverage of these terms in Chapter 2. I've used them here without explaining them simply because most readers of this book probably have at least some exposure to Delphi and its programming language.

Browsing in the Editor

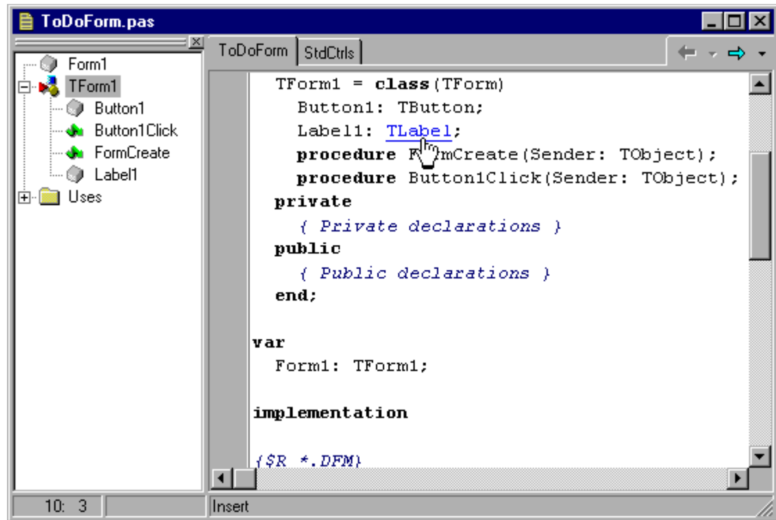
Another feature of the AppBrowser editor is the *Tooltip Symbol Insight*. Move the mouse over a symbol in the editor, and a Tooltip will show you where the identifier is declared. This feature can be particularly important for tracking identifiers, classes, and functions within an application you are writing, and also for referring to the source code of the Visual Component Library (VCL).

note Although it may seem a good idea at first, you cannot use Tooltip Symbol Insight to find out which unit declares an identifier you want to use. If the corresponding unit is not already included, in fact, the Tooltip won't appear.

The real bonus of this feature, however, is that you can turn it into a navigational aid. When you hold down the Ctrl key and move the mouse over the identifier Delphi creates an active link to the definition instead of showing the Tooltip. These links are displayed with the blue color and underline style that are typical of Web browsers, and the pointer changes to a hand whenever it's positioned on the link, as shown in Figure 1.7.

For example, you can Ctrl-click on the `TLabel` identifier to open its definition in the VCL source code. As you select references, the editor keeps track of the various positions you've jumped to, and you can move backward and forward among them—again as in a Web browser. You can also click on the drop-down arrows near the Back and Forward buttons to view a detailed list of the lines of the source code files you've already jumped to, for more control over the backward and forward movement.

Figure 1.7: Delphi's browsing capability is activated by keeping the Ctrl key pressed and moving the mouse over an identifier. Images captured in Delphi 5 and Delphi 12.




```
type
  {TODO -oOwner -cGeneral : ActionItem}
  TForm26 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

How can you jump directly to the VCL source code if it is not part of your project? The AppBrowser editor can find not only the units in the Search path (which are compiled as part of the project), but also those in Delphi's Debug Source, Browsing, and Library paths. These directories are searched in the order I've just listed, and you can set them in the Directories/Conditionals page[42] of the Project Options dialog box and in the Library page of the Environment Options dialog box. By default, Delphi adds the VCL source code directories in the Browsing path of the environment, which has the following declaration:

```
$(DELPHI)\source\vcl;$(DELPHI)\source\rtl\Corba;
$(DELPHI)\source\rtl\Sys;$(DELPHI)\source\rtl\Win;
$(DELPHI)\source\Internet
```

42 The Browsing Path is now configured the Language | Delphi | Library section of the Tools | Options dialog box. The default path is very long, and not worth listing here.

In this series of paths, the declaration `$(DELPHI)` stands for the directory where Delphi is installed[43].

Class Completion

The third important feature of Delphi's AppBrowser editor is *Class Completion*, activated by pressing the Ctrl+Shift+C key combination. Adding an event handler to an application is a fast operation, as Delphi automatically adds the declaration of a new method to handle the event in the class and provides you with the skeleton of the method in the implementation portion of the unit. This is part of Delphi's support for visual programming.

Newer versions of Delphi also simplify life in a similar way for programmers who write a little extra code behind event handlers. The new code-generation feature, in fact, applies to general methods, message-handling methods, and properties. For example, if you type the following code in the class declaration:

```
public
  procedure Hello (MessageText: string);
```

and then press Ctrl+Shift+C, Delphi will provide you with the definition of the method in the implementation section of the unit, generating the following lines of code:

```
{ TForm1 }
procedure TForm1.Hello(MessageText: string);
begin
end;
```

This is really handy, compared with the traditional approach of many Delphi programmers, which is to copy and paste one or more declarations, add the class names, and finally duplicate the `begin .. end` code for every method copied.

Class Completion can also work the other way around. You can write the implementation of the method with its code directly, and then press Ctrl+Shift+C to generate the required entry in the class declaration.

Glancing back at the Explorer settings shown in Figure 1.5, you'll see one option for Class Completion—you can use it to complete the definition of a property. If you simply type in a brand-new form class,

```
property X: Integer;
```

---

43 This is now replaced by the $(BDS) symbolic reference.

and activate Class Completion, Delphi generates a SetX method for the property and adds the FX field to the class. The resulting code looks like this:

```
type
  TForm1 = class(TForm)
  private
    FX: Integer;
    procedure SetX(const Value: Integer);
  public
    property X: Integer read FX write SetX;
  end;

implementation

procedure TForm1.SetX(const Value: Integer);
begin
  FX := Value;
end;
```

This really saves a lot of typing. In fact, you can even partially control how Class Completion generates Set and Get methods for the property, as discussed in Chapter 3 in the section devoted to properties.

Code Insight

Besides the Code Explorer, Code Completion, and the navigational features, the Delphi editor still supports the *Code Insight*[44] technology originally introduced in Delphi 3. Collectively, the Code Insight techniques are based on a constant background parsing, both of the source code you write and of the source code of the system units your source code refers to. Code Insight comprises five capabilities:

- **Code Completion** allows you to choose the property or method of an object simply by looking it up on a list, or by typing its initial letters. To activate it you can simply type the name of an object, such as Button1, then add the dot, and wait. To force the display of the list, press Ctrl+Spacebar; to remove it when you don't want it, press Esc. Code Completion also lets you look for a proper value in an assignment statement. As you type := after a variable or property, Delphi will list all the other variables or objects of the same type, plus the objects having properties of that type. While the list is visible, you can right-click on it to change the order of the items, sorting either by scope or by name.

44 Most of the Code Insight features are now based on a DelphiLSP engine, a Delphi implementation of the Language Server Ptotocol defined by Microsoft. The behavior in the IDE remains almost unchanged.

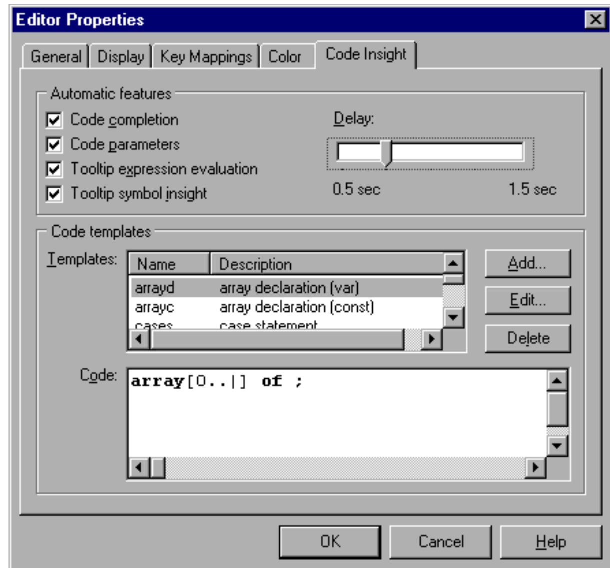
- **Code Templates** allow you to insert one of the predefined code templates, such as a complex statement with an inner `begin..end` block. Code Templates must be activated manually, by typing Ctrl+J to show a list of all of the templates[45]. If you type a few letters (such as a keyword) before pressing Ctrl+J, Delphi will list only the templates starting with those letters.
- **Code Parameters** display, in a hint or Tooltip window, the data type of a function's or method's parameters while you are typing it. Simply type the function or method name and the open (left) parenthesis, and the parameter names and types appear immediately in a popup hint window. To force the display of Code Parameters, you can press Ctrl+Shift+spacebar. As a further help, the current parameter appears in boldface type.
- **Tooltip Expression Evaluation** is a debug-time feature. It shows you the value of the identifier, property, or expression that is under the mouse cursor.
- **Tooltip Symbol Insight** lets you see the definition of an identifier in a Tooltip, as discussed earlier, in the section "Browsing in the Editor."

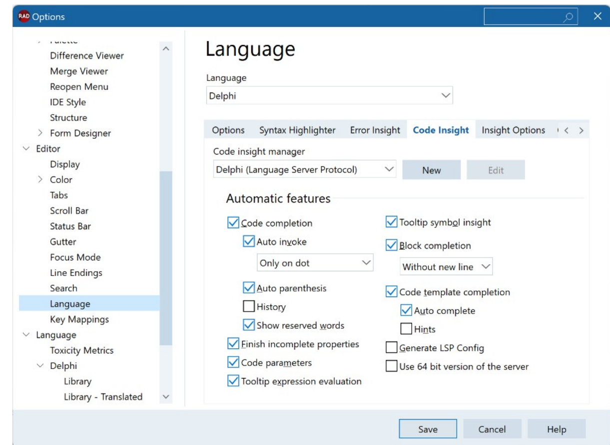
You can enable and disable or configure each of these features in the Code Insight page of the Editor Options dialog box[46], shown in Figure 1.8.

45 Since Delphi 2006, Code Templates have been replaced and superseded by the more powerful Live Templates, which are invoked either by the Tab key or the plain Space key, but are still listed if you press the original Ctrl+J shortcut key. Live Templates are covered in my "Delphi 2007 Handbook".

46 The configuration is now under the Editor | Language page of the Tools | Options dialog box. The page has multiple tabs including a "Code Insigth" one, as shown in Figure 1.8.

Figure 1.8:
The Code Insight page of the Editor Options dialog box allows you to activate or disable each of these technologies and to set the delay time. Images captured in Delphi 5 and Delphi 12.





note When the code you've written is not correct, Code Insight won't work, and you may see just a generic error message indicating the situation. It is possible to display specific Code Insight errors in the Message pane (which must already be open; it doesn't open automatically to display compilation errors). To activate this feature you need to set another undocumented registry entry, setting the string key `Delphi\5.0\Compiling\ShowCodeInsiteErrors` to the value "1".[47]

---

47 This feature is now active by default and it can be configured in the same page of the Tools Options dialog box, in the "Error Insight" tab.

More Editor Shortcut Keys

The editor has many more shortcut keys, which depend on the editor style you've selected. Here are a few of the less-known shortcuts, most of which are useful:

- Ctrl+Shift plus a number key from 0 to 9 activates a bookmark, indicated in a "gutter" margin on the side of the editor. To jump back to the bookmark you can press the Ctrl key plus the number key. The usefulness of bookmarks in the editor is limited by the fact that a new bookmark can override an existing one and that bookmarks are not persistent[48]; they are lost when you close the file.
- Ctrl+E activates the incremental search. You can press Ctrl+E and then directly type the word you want to search for, without the need to go through a special dialog box and click the Enter key to do the actual search.
- Ctrl+Shift+I indents multiple lines of code at once. The number of spaces used is the one that is set by the Block Indent option in the Editor page of the Environment Options dialog box. Ctrl+Shift+U is the corresponding key for unindenting the code.
- Ctrl+O+U toggles the case of the selected code; you can also use Ctrl+K+E to switch to lowercase and Ctrl+K+F to switch to uppercase.
- Ctrl+Shift+R starts recording a macro, which you can later play by using the Ctrl+Shift+P shortcut. The macro records all the typing, moving, and deleting operations done in the source code file. Playing the macro simply repeats the sequence—an operation that has little meaning once you've moved on to a different source code file. I have yet to find a use for this technique, although I guess Borland uses it for testing purposes[49].
- Holding down the Alt key, you can drag the mouse to select rectangular areas of the editor, not just consecutive lines and words.

48 This is not true any more: Editor bookmarks are saved along with other local project settings.

49 I started using this feature (which is now surfaced with specific buttons at the bottom of the editor pane) when I need to perform repeated editing, like deleting or adding the same text to multiple lines. It can be very effective.

The Form Designer

Another Delphi window you'll interact with very often is the Form Designer, a visual tool for placing components on forms. In the Form Designer you can select a component directly with the mouse or through the Object Inspector, a handy feature when a control is behind another one or is very small. If a control covers another one completely, you can use the Esc key to select the parent control of the current one. You can press Esc one or more times to select the form, or press and hold Shift while you click on the selected component. This will deselect the current component and select the form by default.

note What if you need to move a control at design time by dragging it, but its area is covered by a child control? Just drag the child control and then press the Esc key (while holding down the mouse button) to switch the dragging operation to the parent control.

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down Ctrl. Using arrow keys is particularly useful for fine-tuning an element's position. (The Snap to Grid option works only for mouse operations.[50]) Similarly, by pressing the arrow keys while you hold down Shift, you can fine-tune the size of a component. (If you press Shift+Ctrl along with an arrow key, the component will be moved only at grid intervals.) Unfortunately, during these fine-tuning operations the component hints with the position and size are not displayed.

To align multiple components or make them the same size, you can select several components and set the `Top`, `Left`, `Width`, or `Height` property for all of them at the same time. To select several components, you can click on them with the mouse while holding down the Shift key, or, if all the components fall into a rectangular area, you can drag the mouse to "draw" a rectangle surrounding them. When you've selected multiple components, you can also set their relative position using the Alignment dialog box (with the Align command of the form's shortcut menu) or the Alignment palette (accessible through the View ➢ Alignment Palette[51] menu command).

---

50 The design time guidelines now available in Delphi offer you a lot of power for aligning components to the sides or the text baseline and effectively replace some of the techniques described here and later. Notice also that you now get some of the hints that were missing when I wrote the text.

51 Now available with the menu View | Toolbars | Align.

When you've finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component in a form. This is particularly helpful, as there is no real Undo operation on forms (only an Undelete one), but the setting is not persistent.

Among its other features, the Form Designer offers a number of Tooltip hints:

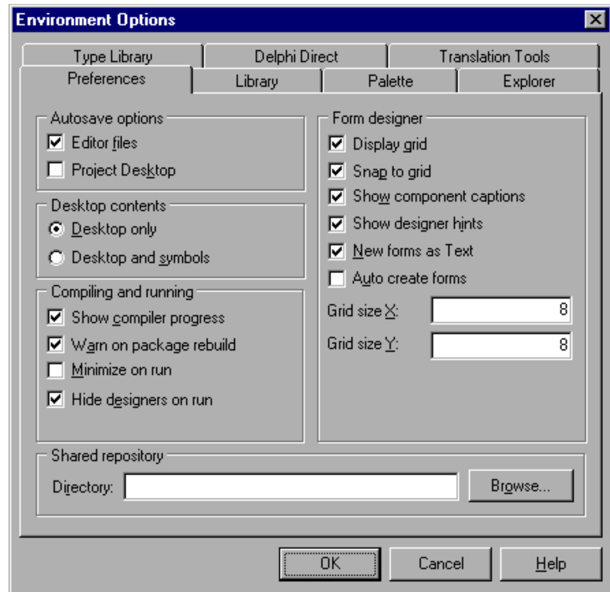
- As you move the pointer over a component, the hint shows you the name and type of the component. This is an alternative to the Show Component Captions environment setting, which I tend to keep always active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for nonvisual components (which are indicated in the Form Designer by icons).
- As you move a component, the hint indicates the current position (the `Left` and `Top` properties).

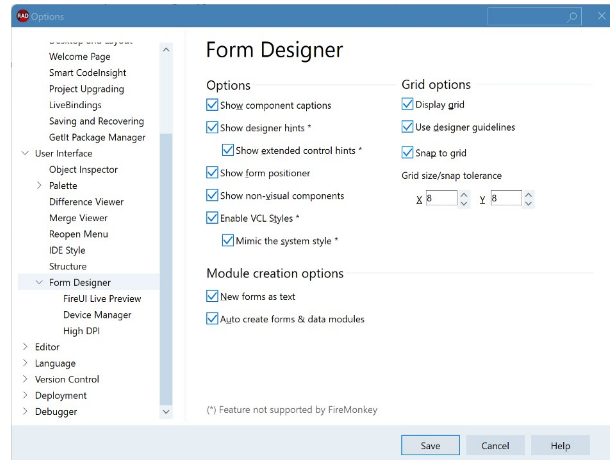
Finally, what may be the most important new Delphi 5 feature of the Form Designer is that you can save DFM (Delphi Form Module) files in plain text instead of the traditional binary resource format[52]. You can toggle this option for an individual form with the Form Designer's shortcut menu, or you can set a default value for newly created forms in the Preferences page of the Environment Options dialog box (see Figure 1.9).

---

52 Using textual DFM files has now long been the default in Delphi.

Figure 1.9:
The Preferences page of the Environment Options dialog box in Delphi 5 allows you to determine whether forms will be created by default and whether the DFM files will hold plain text. Images captured in Delphi 5 and Delphi 12.





In the same page you can also specify whether the secondary forms of a program will be automatically created at startup, a decision you can always reverse for each individual form (using the Forms page of the Project Options dialog box). But the most obvious difference between Delphi 5 and past versions, when working with forms, is the Object Inspector.

Having DFM files stored as text is a welcome addition; it lets you better operate with version-control systems. Programmers won't get a real advantage from this

feature, as you could already open the binary DFM files in the Delphi editor with a specific common of the shortcut menu of the designer. Version control systems, on the other hand, need to store the textual version of the DFM files to be able to compare them and capture the differences between two versions of the same file. This was probably introduced in Delphi 5 in conjunction with the new TeamSource version control system interface, discussed in Chapter 19[53].

In any case, note that if you use DFM files as text, Delphi will still convert them into a binary resource format before including them in the executable file of your programs. DFM are linked into your executable in binary format to reduce the executable size (although they are not really compressed) and to improve run-time performance (they can be loaded faster).

note Earlier versions of the Delphi IDE won't recognize text DFM files. When you open a textual DFM in Delphi 4 (or past versions), you'll get an error. To fix it, you should manually first use Delphi 5 to convert the DFM file to the binary format, using the shortcut menu of the Form Designer. (On a computer that doesn't have Delphi 5, you can use the Delphi 4 command-line tool `CONVERT`.[54]) When you open an existing DFM in the Delphi 5 IDE, the original DFM format will be preserved (unless you explicitly change it using the Text DFM shortcut menu item), thus allowing you to reopen the same form in past version of Delphi.

The Object Inspector in Delphi 5

If you have used Delphi in the past, you will immediately see that there is something new in the Object Inspector. The most important changes involve the graphical drop-down lists and the property categories.

The first element is the simplest to use. The drop-down list for a property in the Object Inspector can include graphical elements. Many of the relevant properties use this feature by default: `Color`, `Cursor` and its variations, generally the `ImageIndex` property of components connected with an ImageList (such as an action, a menu item, or a toolbar button), the Pen and Brush styles, and a few others. For example, Figure 1.10 shows the list of cursors (`Cursor` properties)[55]. Of course, developers of Delphi components and add-ins will be able to customize this feature, and you'll see more graphical drop-down elements in the future. See the fol-

53 Given this entire feature is no longer available (and it has been removed from the product for a long time), I'm going to remove that section of the book.

54 The *convert.exe* tools continues to exist and be available in the *bin* folder today

55 The list of cursors is still displayed today, even if I haven't included an updated image.

lowing section "Drop-Down Fonts in the Object Inspector" for a simple customization of this window.

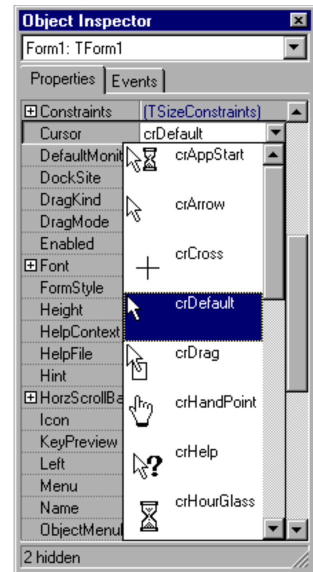
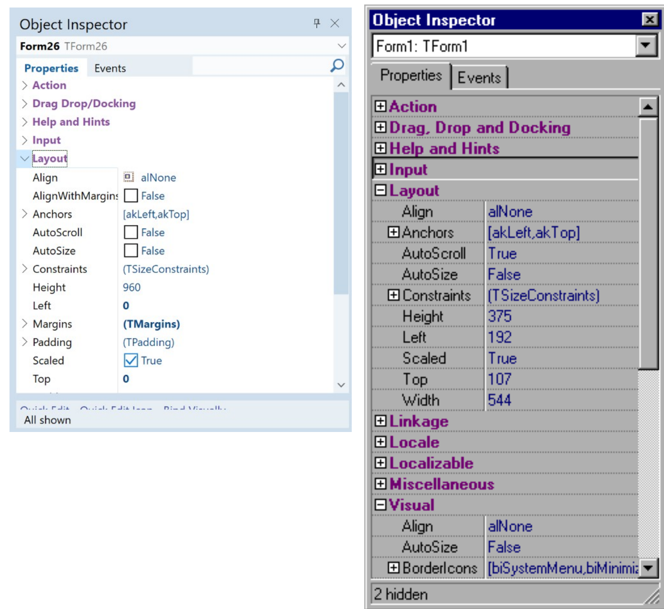


Figure 1.10: A graphical drop-down list in the Delphi 5 Object Inspector, showing available cursors.

It takes a little more time to get used to the property categories[56]. To understand this feature, you first need to make it visible. To display properties by category instead of by name, right-click in the Object Inspector and choose the proper Arrange option from the shortcut menu. You can see the effect of this choice in Figure 1.11. Looking carefully at this figure, you may notice something strange—the `Align` property is available in two different categories. This is a general rule; categories are not exclusive, and a property can register itself for multiple categories.

56 While the ability to group Object Inspector properties in categories still exists today (see Figure 1.11), this feature is not frequently used and generally not recommended. Because of this, I've skipped capturing new versions of some of the other figures.

Figure 1.11:
The effect of arranging properties by category. Images captured in Delphi 5 and Delphi 12.



Categories have the benefit of reducing the complexity of the Object Inspector. You can use the View submenu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). For example, in Figure 1.12 you can see the properties of a form arranged by name, but only the properties within the Visual and Input categories. In fact, as you can see in the status bar of the Object Inspector, 44 properties are hidden. The arrangement and the visibility you select will affect events, as well.

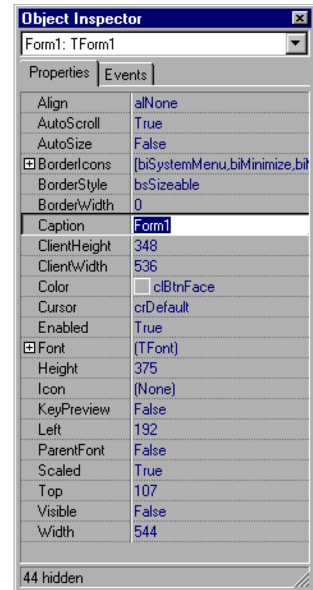


Figure 1.12: You can hide properties of some categories, even when they are arranged by name.

note Another new feature of the Delphi 5 Object Inspector is the ability to select the component referenced by a property. To do this, double-click with the left mouse button on the property value while keeping the Ctrl key pressed. For example, if you have a MainMenu component in a form and you are looking at the properties of the form in the Object Inspector, you can select the MainMenu component by moving to the MainMenu property of the form and Ctrl+double-clicking on the value of this property. This selects the main menu indicated as the value of the property in the Object Inspector. This feature can be very useful when you have many connected components; for example, when using multiple data-source and dataset components.[57]

Drop-Down Fonts in the Object Inspector[58]

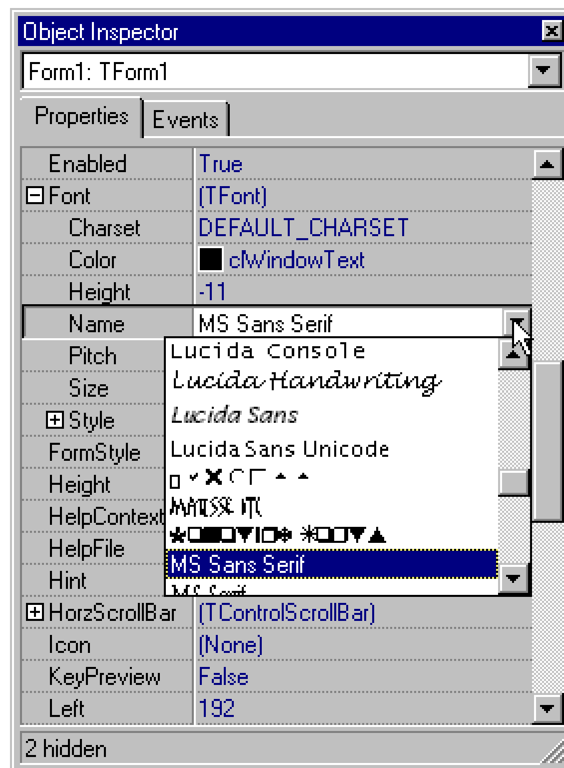
The Delphi 5 Object Inspector has graphical drop-down lists for several properties. You might want to add one showing the actual image of the font you are selecting,

57 The ability to jump to the connected component has later been extended with the ability to expand the properties of the connected component in place, as if it was a local property with subproperties.

58 This Object Inspector customization still works today, but it is rarely used as painting the drop down list of fonts with the actual fonts can be very slow, compared to showing the font names only.

corresponding to the `Name` subproperty of the `Font` property. This capability is actually built into Delphi 5, but it has been disabled because most computers have a large number of fonts installed and rendering them can really slow down the computer. If you want to enable this feature, you have to install in Delphi a package that enables the `FontNamePropertyDisplayFontNames` global variable of the DsgnIntf unit. I've done this in the OiFontPk package, which you can find among the program examples for this chapter[59].

Once this package is installed, you can move to the `Font` property of any component, and use the graphical Name drop-down menu, as displayed below:



There is a second, more complex customization of the Object Inspector I like and use frequently, a custom font for the entire Object Inspector, to make its text more visible. This feature is particularly useful for public presentations[60]. You can find the

59 Again, this is not recommended as this makes the display terribly slow. The feature can be enabled without the special add-in package.

60 I won't recommend using this old add-in package either, I doubt it's going to work smoothly.

package to install custom fonts in the Object Inspector on my Web site, `www.marcocantu.com`.

Secrets of the Component Palette

The Component Palette[61] is very simple to use, but there are a few things you might not know. There are four simple ways to place a component on a form:

- After selecting a control in the palette, click within the form to set the position for the control, and press-and-drag the mouse to size it.
- After selecting any component, simply click within the form to place it with the default height and width.
- Double-click the icon in the palette to add a component of that type in the center of the form.
- Shift-click on the component icon to place several components of the same kind in the form. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component Palette.

You can select the Properties command on the shortcut menu of the palette to completely rearrange the components in the various pages, possibly adding new elements or just moving them from page to page. In the resulting Properties page, you can simply drag a component from the Components list box to the Pages list box to move that component to a different page.

note When you have too many pages in the Component Palette, you'll need to scroll them to reach a component. There is a simple trick you can use in this case: Rename the pages with shorter names, so that all the pages will fit on the screen. Obvious—once you've thought about it.

The real undocumented feature of the Component Palette is the "hot-track" activation. By setting special keys of the Registry, you can simply select a page of the palette by moving over the tab, without any mouse click. The same feature can be

61 The Component Palette has been replaced by the Tools Palette, but some of the description in this section (like the ways to select components) still applies. Delphi still has also a Components Toolbar that acts, behaves, and can be customized much like the original Component palette, although it's not a stable and reliable feature and Embarcadero has hinted at deprecating and removing it.

applied to the component scrollers on both sides of the palette, which show up when a page has too many components.

To activate this hidden feature you have to add an `Extras` key under `HKEY_CURRENT_USER\Software\Borland\Delphi\5.0`. Under this key you have to enter two string values, `AutoPaletteSelect` and `AutoPaletteScroll`, and set each value to the string `'1'`.

Defining Event Handlers

There are several techniques you can use to define a handler for an event of a component:

- Select the component, move to the Events page, and either double-click in the white area on the right side of the event or type a name in that area and press the Enter key.
- For many controls, you can double-click on them to perform the default action, which is to add a handler for the `OnClick`, `OnChange`, or `OnCreate` events.

When you want to remove an event handler you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the `begin` and `end` keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event handler that is still empty, consider adding a comment to it (even simply the `//` characters), so that it will not be removed.

Copying and Pasting Components

An interesting feature of the Form Designer is the ability to copy and paste components from one form to another or to duplicate the component in the form. During this operation Delphi duplicates all the properties and keeps the connected event handlers, and, if necessary, changes the name of the control (which must be unique in each form).

It is also possible to copy components from the Form Designer to the editor and vice versa. When you copy a component to the Clipboard, Delphi also places the textual

description there. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component. For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Now, if you change the name of the object, its caption, or its position, for example, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'My Button'
  TabOrder = 0
  Font.Name = 'Arial'
end
```

Copying this description and pasting it into the form will create a button in the specified position with the caption *My Button* in an Arial font.

To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and it might change the position of the component so that it doesn't overlap a previous copy. Of course, if you write something completely wrong and try to paste it into a form, Delphi will display an error message indicating what has gone wrong.

You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form, by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

To create a component template, select one or more components and issue the Component ➢ Create Component Template menu command. This opens the Component Template Information dialog box (see Figure 1.13) where you enter the name of the template, the page of the Component palette where it should appear, and an icon.

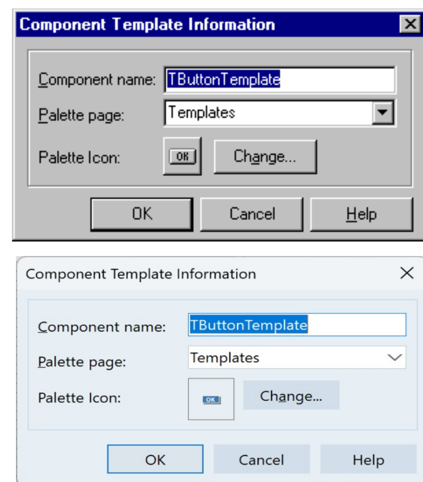


Figure 1.13: The Component Template Information dialog box. Images captured in Delphi 5 and Delphi 12.

By default, the template name is the name of the first component you've selected followed by the word *Template*. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Component Palette (when Delphi displays the pop-up hint).

All the information about component templates is stored in a single file, DELPHI32.DCT[62], but there is apparently no way to retrieve this information and edit a

62 The file is now *bds.dct,* stored in C:\Users\xxx\AppData\Roaming\Embarcadero\BDS\xxx.

template. What you can do, however, is place the component template in a brand-new form, edit it, and install it again as a component template *using the same name*. This way you can overwrite the previous definition.

---

note A group of Delphi programmers can share component templates by storing them in a common directory, adding to the Registry the entry `CCLibDir` under the key `Software\Borland\Delphi\5.0\Component Templates`.[63]

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the new *frames* technology in Delphi 5 does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each of the instances of the frame.

You can see a simple example, called Frames1, in Figure 1.14[64]. A screen snapshot doesn't really mean much; you should open the program or rebuild a similar one if you want to start playing with frames.

Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than Component Templates. Chapter 4 provides an in-depth look at the VCL and includes a more detailed description of frames. As you might imagine from this short introduction, frames are a powerful new technique.

---

63 The registry key is still exists, but I'm not sure if this undocumented configuration works today.

64 Frames work today, for both VCL and FireMonkey, even if I haven't captured a new image for Figure 1.14.

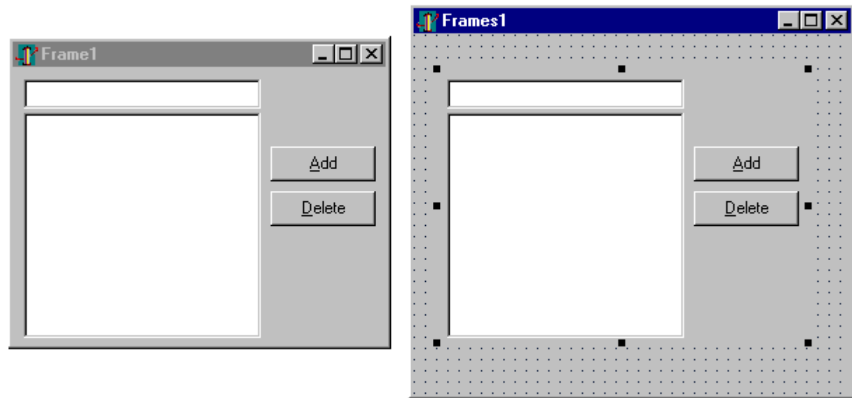


Figure 1.14: The Frames1 example demonstrates the use of frames. The frame (on the left) and its instance inside a form (on the right) are kept in synch.

Managing Projects

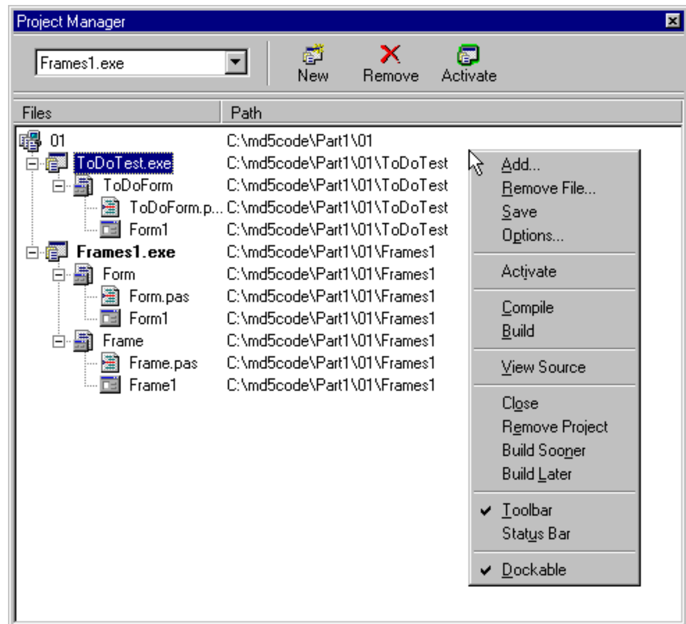
One of the new features of the Delphi 4 IDE was the multi-target Project Manager (View ➢ Project Manager). The Project Manager works on a project *group*, which can have one or more projects under it. For example, a project group can include a DLL and an executable file, or multiple executable files.

In Figure 1.15 you can see the Project Manager with the project group for the examples of the current chapter. As you can see, the Project Manager is based on a tree view, which shows the hierarchical structure of the project group, the projects, and all of the forms and units that make up each project. You can use the simple toolbar and the more complex shortcut menus of the Project Manager to operate on it. The shortcut menu is context-sensitive; its options depend on the selected item. There are menu items to add a new or existing project to a project group, to compile or build a specific project, or to open a unit.

Of all the projects in the group only one is active, and this is the project you operate upon when you select a command such as Project ➢ Compile. The Project pull-down of the main menu has two commands you can use to compile or build all the projects of the group. (Strangely enough, these commands are not available in the shortcut menu of the Project Manager for the project group.[65]) When you have mul-

65 They were not, now they've been added. The Project Manager has seen many extensions over the years, but its core behavior is still what I described here (see also Figure 1.15).

tiple projects to build, you can set a relative order by using the Build Sooner and Build Later commands. These two commands basically rearrange the projects in the list.



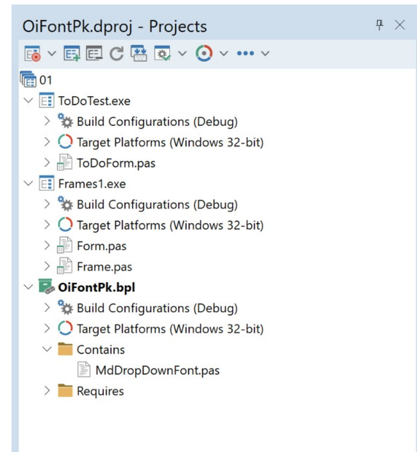


Figure 1.15: Delphi's multi-target Project Manager. Images captured in Delphi 5 and Delphi 12.

Delphi 5 adds some features to the Project Manager. You can now drag source code files from Windows folders or Windows Explorer onto a project in the Project Manager window to add them to that project. Unfortunately, you cannot drag an existing

project or package file to add it to the entire project group. You can also drag from one project to another of the same project group.

Another big advantage is that the Project Manager automatically selects as current project the one you are working with, for example, opening a file. You can easily see which project is selected and change it by using the combo box on the top of the form[66].

note Besides adding Pascal files and projects, you can add Windows resource files to the Project Manager; they are compiled along with the project. Simply move to a project, select the Add shortcut menu, and choose *Resource file (*.rc)* as the file type. This resource file will be automatically bound to the project, even without a corresponding `$R` directive.

Delphi saves the project groups with the new `.BPG` extension, which stands for Borland Project Group[67]. This feature comes from C++Builder and from past Borland C++ compilers, a history that is clearly visible as you open the source code of a project group, which is basically that of a makefile in a C/C++ development environment[68].

Project Options

The Project Manager doesn't provide a way to set the options of two different projects at one time. What you can do instead is invoke the Project Options dialog from the Project Manager for each project[69]. The first page of Project Options (Forms) lists the forms that should be created automatically at program startup and the forms that are created manually by the program. The next page (Application) is used to set the name of the application and the name of its Help file, and to choose its icon. Other Project Options choices relate to the Delphi compiler and linker, version information, and the use of run-time packages.

---

66 That combo box is still available in the form of a drop down split button, the first button of the Project Manager toolbar, with the symbol of a target superimposed.

67 This is not the case any more. Project and project groups are now XML files in the MSBuild format, as this is the tool for building applications since Delphi 2007, as detailed in my "Delphi 2007 Handbook". I took the freedom of removing the project group files listed in the original book, as they are totally useless in today's Delphi.

68 The format was later changed to the MSBUILD XML format. You can still open a Delphi 5 project group file today, although the IDE will ask you to save it in the current format.

69 The Project Options dialog still exists, but the sequence of pages has changed, with many more features available.

There are two ways to set compiler options. One is to use the Compiler page of the Project Options dialog. The other is to set or remove individual options in the source code with the `{$X+}` or `{$X-}` commands, where you'd replace `X` with the option you want to set. This second approach is more flexible, since it allows you to change an option only for a specific source-code file, or even for just a few lines of code. The source-level options override the compile-level options.

All of the Project Options are saved automatically with the project, but in a separate file with a .DOF extension[70]. This is a text file you can easily edit. You should not delete this file if you have changed any of the default options. Delphi also saves the compiler options in another format in a CFG file, for command line compilation.

Another alternative for saving compiler options is to press Ctrl+O+O (press the O key twice while keeping Ctrl pressed). This inserts, at the top of the current unit, compiler directives that correspond to the current project options, as in the following listing[71]:

```
{$A+,B-,C+,D+,E-,F-,G+,H+,I+,J+,K-,L+,M-,N+,O+,P+,Q-,
R-,S-,T-,U-,V+,W-,X+,Y+,Z1}

{$MINSTACKSIZE $00004000}

{$MAXSTACKSIZE $00100000}

{$IMAGEBASE $00400000}

{$APPTYPE GUI}
```

Compiling and Building Projects

There are several ways to compile a project. If you run it (by pressing F9 or clicking the Run toolbar icon), Delphi will compile it first. When Delphi compiles a project, it compiles only the files that have changed.

---

70 Project options files are gone as well, and so are their command line counterparts (as compilation now follows the same steps both from the command line and from the IDE. In current versions of Delphi, the project settings are saved in project files or in build configurations, can be shared among projects, have release and build variations, and much more. Still the core application settings haven't changed much. Coverage of project settings management is my Delphi Handbooks, as they were extended from version to version.

71 There are now many more lines inserted, but the keyboard shortcut and the overall concept remain the same.

If you select Compile ➢ Build All instead[72], every file is compiled, even if it has not changed. You should only need this second command infrequently, since Delphi can usually determine which files have changed and compile them as required. The only exception is when you change some project options. In this case you have to use the Build All command to put the new options into effect.

To build a project, Delphi first compiles each source code file, generating a Delphi compiled unit (DCU). (This step is performed only if the DCU file is not already up to date.) The second step, performed by the linker, is to merge all the DCU files into the executable file, optionally with compiled code from the VCL library (if you haven't decided to use packages at run time). The third step is binding into the executable file any optional resource files, such as the RES file of the project, which hosts its main icon, and the DFM files of the forms. You can better understand the compilation steps and follow what happens during this operation if you enable the Show Compiler Progress option (in the Preferences page of the Environment Options dialog box).

note Delphi doesn't always properly keep track of when to rebuild units based on other units you've modified. This is particularly true for the cases (and there are many) in which user intervention confuses the compiler logic. For example, renaming files, modifying source files outside the IDE, copying older source files or DCU files to disk, or having multiple copies of a unit source file in your search path can break the compilation. Every time the compiler shows some strange error message, the first thing you should try is the Build All command to resynchronize the make feature with the current files on disk.

The Compile command can be used only when you have loaded a project in the editor. If no project is active and you load a Pascal source file, you cannot compile it. However, if you load the source file *as if it were a project*, that will do the trick and you'll be able to compile the file. To do this, simply select the Open Project toolbar button and load a PAS file. Now you can check its syntax or compile it, building a DCU.[73]

I've mentioned before that Delphi allows you to use run-time packages, which affect the distribution of the program more than the compilation process. Delphi packages are dynamic link libraries (DLLs) containing Delphi components. By using packages, you can make an executable file much smaller. However, the program won't

72 The menu command is now Project | Build.

73 In recent versions of Delphi (probably since MSBuild was introduced) this trick doesn't work any more. There is apparently no way to compile an individual source code file outside of a project in the IDE. However, you can easily compile a single Pascal source code files with the command line compiler.

run unless the proper dynamic link libraries (such as vcl50.bpl, which is quite large) are available on the computer where you want to run the program.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the apparently smaller program built with run-time packages is much larger than the space required by the apparently bigger stand-alone executable file. Of course if you have multiple applications on a single system, you'll end up saving a lot, both in disk space and memory consumption at run time. The use of packages is often but not always recommended. I'll discuss all the implications of packages in detail in Chapter 13, where we'll build some packages, and in Chapter 14, which is devoted to DLLs and packages.

note You don't have to use the stock vcl50.bpl package if you only need a small set of VCL units. You can create your own mini-VCL package, as long as you don't call it vcl50.bpl.

In both cases, Delphi executables are extremely fast to compile, and the speed of the resulting application is comparable to that of a C or C++ program. Delphi compiled code runs at least five times faster than the equivalent code in interpreted or "semi-compiled" tools[74].

Conditional Compilation for Different versions of Delphi

You can test the `VER130` define to check whether you are compiling with Delphi 5 or an earlier version. This can be useful if you want to compile the same program with different versions of Delphi and make minor changes to the source code in each of the versions. If you want to add some specific Delphi 5 code, you can write that code as follows:

```
{$IFDEF VER130}
  // Delphi 5 specific code
{$ENDIF}
```

Each of the past versions of the Delphi included a specific define, so you can write a complex statement to provide alternative coding solutions for different Delphi versions. The numbering scheme starts from the last version of Pascal compiler from

74 I'm not sure if this specific number ("five times faster", which I assume was in reference to Visual Basic) makes sense today, with many alternatives between compiled and interpreted code. Delphi programs are still native and remain fast. Some of the options used today for desktop development, like JavaScript, are clearly in a different league, both in terms of slower performance and in terms of the complex deployment dependencies.

Borland before Delphi, Borland Pascal with Object version 7, and also includes the versions of the Pascal compiler included in Borland C++Builder[75]:

- `VER80` for Delphi 1
- `VER90` for Delphi 2
- `VER93` for C++Builder 1
- `VER100` for Delphi 3
- `VER110` for C++Builder 3
- `VER120` for Delphi 4
- `VER125` for C++Builder 4

Exploring a Project[76]

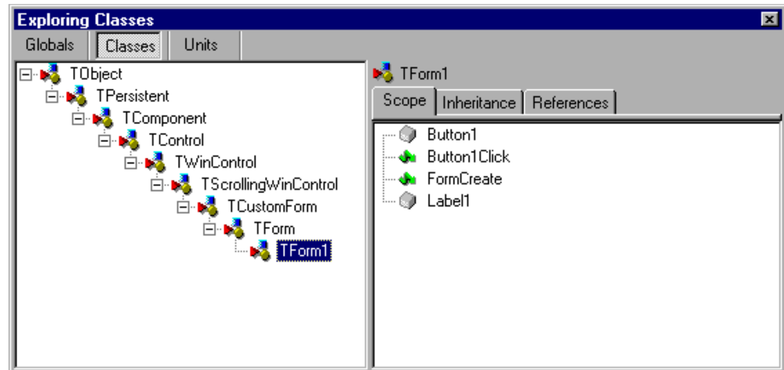
Past versions of Delphi included an Object Browser, which you could use when a project was compiled to see a hierarchical structure of its classes and to look for its symbols and the source code lines where they are referenced. Delphi 5 includes a similar but enhanced tool, with a new name—Project Explorer. Like the Code Explorer, it is updated automatically as you type, without recompiling the project.

The Project Explorer retains from the Object Browser the main structure of Classes, Units, and Globals, but it lets you choose whether to look only for symbols defined within your project or for those from both your project and the VCL. You can see an example with project symbols only in Figure 1.16.

75 The list has been added to the Delphi docwiki, and it can be found at [docwiki.embarcadero.-com/RADStudio/en/Compiler_Versions](docwiki.embarcadero.com/RADStudio/en/Compiler_Versions). The Delphi 12 compiler defines VER360.

76 This entire feature isn't part of recent versions of Delphi, so you can skip this section.

Figure 1.16:
The Project Explorer, a completely updated Object Browser[77]



You can change the settings of this Explorer and those of the Code Explorer in the Explorer page of the Environment Options (see Figure 1.5) or by selecting the Properties command in the shortcut menu of the Project Explorer. Some of the Explorer categories you see in this window are specific to the Project Explorer, others relate to both tools.

Additional and External Delphi Tools

Besides the IDE, when you install Delphi you get other, external tools. Some of them, such as the Database Desktop, the Package Collection Editor (PCE.EXE), and the Image Editor (ImagEdit.EXE), are available from Tools menu of the IDE. In addition, the Client/Server edition has a link to the SQL Monitor (SqlMon.EXE)[78].

Other tools that are not directly accessible from the IDE include many command-line tools you can find in the `Bin` directory of Delphi. For example, there is a command-line Delphi compiler (DCC.EXE), a Borland resource compiler (BRC32.EXE and BRCC32.EXE), and an executable viewer (TDump.EXE).[79]

77 As mentioned earlier, this feature is no longer available in recent versions of Delphi.

78 Most of these tools are now gone. The Tools menu in Delphi 12 includes by default the *Bitmap Style Designer*, the *FireDAC Explorer*, the *FireDAC Monitor*, the *REST Debugger*, the *XML Mapper*, and – in some editions – the *RAD Server Console*.

79 These low level tools, instead, are still available today, even if with some differences. There are multiple Delphi compilers, for example.

Finally, some of the sample programs that ship with Delphi are actually useful tools that you can compile and keep at hand. I'll discuss some of these tools in the book, as needed. Here are a few of the useful and higher-level tools[80]:

- **WinSight** (WS.EXE) is a Windows "message spy" program available in the `Bin` directory.[81]
- **Database Explorer** can be activated from the Delphi IDE or as a stand-alone tool, using the DBExplor.EXE program of the `Bin` directory.[82]
- **Convert** (Convert.EXE) is a command-line tool you can use to convert `DFM` files into the equivalent textual description and vice versa.
- **Turbo Grep** (Grep.EXE) is a command-line search utility, much faster than the embedded Find in Files mechanism but not so easy to use.
- **Turbo Register Server** (TRegSvr.EXE) is a tool you can use to register ActiveX libraries and COM servers. The source code of this tool is available under Demos/ActiveX/TRegSvr.[83]
- **Resource Explorer** is a powerful resource viewer (but not a full-blown resource editor) you can find under Demos/ResXplor[84].
- The Delphi 5 CD also includes a separate installation for **Resource Workshop[85]**. This is an old 16-bit resource editor that can also manage Win32 resource files. It was formerly included in Borland C++ and Pascal compilers for Windows, and it was much better than the standard Microsoft resource editors then available. Although its user interface hasn't been updated and it doesn't handle long file names, this tool can still be very useful for building custom or special resources. It also lets you explore the resources of existing executable files. You'll find more information about Windows resources and the use of Resource Workshop in Chapter 19.

---

80 *Convert*, *Grep*, and *TRegSvr* still exist today. For the other tools, see the respective footnotes.

81 The *WinSight* tool is not available any more. There are similar free utilities for Windows.

82 *DbExplorer* has been replaced by equivalent FireDAC utilities, some of which are listed in the Tools menu, as covered in a previous footnote.

83 The tool is still available, but not its source code.

84 This demo is no longer part of the core product demos.

85 Not only there is no CD, but also no version of the *Resource Explorer* available with the product. There are similar free utilities for Windows.

The Files Produced by the System

Delphi produces a number of files for each project, and you should know what they are and how they are named. There are basically two elements that have an impact on how files are named: the names you give to a project and its units, and the predefined file extensions used by Delphi. Table 1.1 lists the extensions of the files you'll find in the directory where a Delphi project resides[86]. The table also shows when or under what circumstances these files are created and their importance for future compilations. Extensions that are new to Delphi 5 are marked in bold.

Table 1.1: Delphi Project File Extensions

Extension	File Type and Description	Creation Time	Required to Compile?
.BMP, .ICO, .CUR	Bitmap, icon, and cursor files: standard Windows files used to store bitmapped images.	Development: Image Editor	Usually not, but they might be needed at run time and for further editing.
.BPG	Borland Project Group[87]: the files used by the new multiple-target Project Manager. It is a sort of makefile.	Development	Required to recompile all the projects of the group at once.
.BPL	Borland Package Library: a DLL including VCL components to be used by the Delphi environment at design time or by applications at run time. (These files used a .DPL extension in Delphi 3.)	Compilation: Linking	You'll distribute packages to other Delphi developers and, optionally, to end-users.
.CAB	The Microsoft Cabinet compressed-file format used for Web deployment by Delphi. A CAB file can store multiple compressed files.	Compilation	Distributed to users.
.CFG	Configuration file with project options. Similar to the DOF files.	Development	Required only if special compiler options have been set.
.DCP	Delphi Component Package: a file with symbol information for the code that was compiled	Compilation	Required when you use packages. You'll distribute it only to other developers along with DPL files.

86 Most of these file types are still used, but not all of them. I haven't added here new files available, including the new project files in MSBUILD format, but only added a few comments to the original list.

87 This files is not used any more, replaced by Project Group files, with the .groupproj extension.

	into the package. It doesn't include compiled code, which is stored in DCU files.		
.DCU	Delphi Compiled Unit: the result of the compilation of a Pascal file.	Compilation	Only if the source code is not available. DCU files for the units you write are an intermediate step, so they make compilation faster.
.DFM	Delphi Form File: a binary file with the description of the properties of a form (or a data module) and of the components it contains.	Development	Yes. Every form is stored in both a PAS and a DFM file.
.~DF[88]	Backup of Delphi Form File (DFM).	Development	No. This file is produced when you save a new version of the unit related to the form and the form file along with it.
.DFN[89]	**Support file for the Integrated Translation Environment (there is one DFN file for each form and each target language).**	**Development (ITE)**	**Yes (for ITE). These files contain the translated strings that you edit in the Translation Manager.**
.DLL	Dynamic Link Library: another version of an executable file.	Compilation: Linking	See .EXE.
.DOF[90]	Delphi Option File: a text file with the current settings for the project options.	Development	Required only if special compiler options have been set.
.DPK	Delphi Package: the project source code file of a package.	Development	Yes.
.DPR	Delphi Project file. (This file actually contains Pascal source code.)	Development	Yes.
.~DP	Backup of the Delphi Project file (.DPR).	Development	No. This file is generated automatically when you save a new version of a project file.
.DSK	Desktop file: contains information about the position of the Delphi windows, the files open in the editor, and other Desktop settings.	Development	No. You should actually delete it if you copy the project to a new directory.

88 Backup files are now saved in sequence under the __history sub-folder of the project source code folder and they use a different logic. The same is true for all of the backup files listed in this table.

89 This format still exists but the translation support isn't installed any more as part of Delphi. The same is true for other file formats associated with the old translation system. The feature can currently be installed using the GetIt package manager.

90 Project options are now part of the .dproj project file.

.DSM[91]	Delphi Symbol Module: stores all the browser symbol information.	Compilation (but only if the Save Symbols option is set)	No. Object Browser uses this file, instead of the data in memory, when you cannot recompile a project.
.DTI[92]	**Design Time Information, used by the new Data Module Designer**	**Development**	**No. This file stores "design-time only" information, not required by the resulting program but very important for the programmer.**
.EXE	Executable file: the Windows application you've produced	Compilation: Linking	No. This is the file you'll distribute. It includes all of the compiled units, forms, and resources.
.HTM	Or `.HTML`, for HyperText Markup Language: the file format used for Internet Web pages	Web deployment of an ActiveForm	No. This is not involved in the project compilation.
.LIC	The license files related to an OCX file.	ActiveX Wizard and other tools	No. It is required to use the control in another development environment.
.OBJ	Object (compiled) file, typical of the C/C++ world.	Intermediate compilation step, generally not used in Delphi	It might be required to merge Delphi with C++ compiled code in a single project.
.OCX	OLE Control eXtension: a special version of a DLL, containing ActiveX controls or forms.	Compilation: Linking	See .EXE.
.PAS	Pascal file: the source code of a Pascal unit, either a unit related to a form or a stand-alone unit.	Development	Yes.
.~PA	Backup of the Pascal file (.PAS).	Development	No. This file is generated automatically by Delphi when you save a new version of the source code.
.RES, .RC	Resource file: the binary file associated with the project and usually containing its icon. You can add other files of this type to a project. When you create custom resource files you might use also the textual format, .RC.	Development Options dialog box. The ITE (Integrated Translation Environment) generates resource files with special comments.	Yes. The main RES file of an application is rebuilt by Delphi according to the information in the Application page of the Project Options dialog box.
.RPS	**Translation Repository (part of the Integrated Translation**	**Development (ITE)**	**No. Required to manage the translations.**

91 This file format and the associated feature don't exist any more.

92 This feature is also long gone, with the matching file format.

	Environment).		
.TLB	Type Library: a file built automatically or by the Type Library Editor for OLE server applications.	Development	This is a file other OLE programs might need.
.TODO	**To-do list file, holding the items related to the entire project.**	**Development**	**No. This file hosts notes for the programmers.**
.UDL	**Microsoft Data Link**	**Development**	**Used by ADO to refer to a data provider. Similar to an alias in the BDE world (see Chapter 12).**

Besides the files generated during the development of a project in Delphi, there are many others generated and used by the IDE itself. In Table 1.2 I've provided a short list of extensions worth knowing about. Most of these files are in proprietary and undocumented formats, so there is little you can do with them.

Table 1.2: Selected Delphi IDE Customization File Extensions[93]

EXTENSION	FILE TYPE
.DCI	Delphi Code Templates
.DRO	Delphi's Object Repository (The repository should be modified with the Tools ➢ Repository command.)
.DMT	Delphi Menu Templates
.DBI	Database Explorer Information
.DEM	Delphi Edit Mask (Files with country-specific formats for edit masks)
.DCT	Delphi Component Templates
.DST	Desktop settings file (one for each desktop setting you've defined)

Looking at Source Code Files

I've just listed some files related to the development of a Delphi application, but I want to spend a little time to cover their actual format. The fundamental Delphi files are Pascal source code files, which are plain ASCII text files. The bold, italic, and colored text you see in the editor depend on syntax highlighting, but they are not saved with the file. It is worth noting that there is one single file for the whole code of the form, not just small code fragments.

93 Many of these files don't exist any more. Desktops settings and component template files are still used.

note In the listings in the book I've tried to match the bold syntax highlighting of the editor for keywords and the italic for strings and comments.

For a form, the Pascal file contains the form class declaration and the source code of the event handlers. The values of the properties you set in the Object Inspector are stored in a separate form description file (with a `.`DFM extension). The only exception is the `Name` property, which is used in the form declaration to refer to the components of the form.

The DFM file is a binary and in Delphi 5 can be saved either as a plain text file or in the traditional Windows Resource format. You can set the default format you want to use for new projects in the Preferences page of the Environment Options dialog box, and you can toggle the format of individual forms with the Text DFM command of a form's shortcut menu. A plain-text editor can read only the text version. However, you can load DFM files of both types in the Delphi editor, which will, if necessary, first convert them into a textual description. The simplest way to open the textual description of a form (whatever the format) is to select the View As Text command on the shortcut menu in the Form Designer. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View As Form command on the shortcut menu in the editor window.

You can actually edit the textual description of a form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, compilation will stop with an error message and you'll need to correct the contents of your DFM file before you can reopen the form. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

note In the book I'll often show you excerpts of DFM files. With most of these excerpts, I'll only be showing the most relevant components or properties; generally, I will have removed the positional properties, the binary values, and other lines providing little useful information.

In addition to the two files describing the form (PAS and DFM), a third file is vital for rebuilding the application. This is the Delphi project file (DPR), which is another Pascal source code file. This file is built automatically, and you seldom need to change it manually. You can see this file with the View ➢ Project Source menu command.

Some of the other, less relevant, files produced by the IDE use the structure of Windows INI files, in which each section is indicated by a name enclosed in square brackets. For example, this is a fragment of an option file (DOF)[94]:

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

The same structure is used by the Desktop files (DSK), which store the status of the Delphi IDE for the specific project, listing the position of each window. Here is a small excerpt:

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

note A lot of information related to the status of the Delphi environment is saved in the Windows Registry, as well as in DSK and other files. I've already indicated a few special undocumented entries of the Registry you can use to activate specific features. You should explore the HKEY_CURRENT_USER/Software/Borland/Delphi/5.0[95] section of the Registry to examine all the setting of the Delphi IDE (including all those you can modify with the Project Options and the Environment Options dialog boxes, as well as many others).

94 As mentioned earlier, option files content is now part of the .drpoj project file.

95 This is still true, although the location in the registry is now *HKEY_CURRENT_USER\Software\Embarcadero\BDS\xx.0*.

The Object Repository

Delphi has several menu commands you can use to create a new form, a new application, a new data module, a new component, and so on. These commands are located in the File menu and in other pull-down menus. What happens if you simply select File ➢ New[96]? Delphi opens the Object Repository, which is used to create new elements of any kind: forms, applications, data modules, thread objects, libraries, components, automation objects, and more.

The New dialog box (shown in Figure 1.17) has a number of pages, hosting all the new elements you can create, existing forms and projects stored in the Repository, Delphi wizards, and the forms of the current project (for visual form inheritance). The pages and the entries in this tabbed dialog box depend on the specific version of Delphi, so I won't list them here.

note The Object Repository has a shortcut menu that allows you to sort its items in different ways (by name, by author, by date, or by description) and to show different views (large icons, small icons, lists, and details). The Details view gives you the description, the author, and the date of the tool, information that is particularly important when looking at wizards, projects, or forms that you've added to the Repository.

The simplest way to customize the Object Repository is to add new projects, forms, and data modules as templates. You can also add new pages and arrange the items on some of them (not including the New and "current project" pages). Adding a new template to Delphi's Object Repository is as simple as using an existing template to build an application. When you have a working application you want to use as a starting point for further development of similar programs, you can save the current status to a template, ready to use later on. Simply use the Project ➢ Add to Repository command, and fill in its dialog box.

Just as you can add new project templates to the Object Repository, you can also add new form templates. Simply move to the form that you want to add and select the Add To Repository command of its shortcut menu. Then indicate the title, description, author, page, and icon in its dialog box.

You might want to keep in mind that as you copy a project or form template to the repository and then copy it back to another directory, you are simply doing a copy and paste operation. This isn't much different than copying the files manually.

96 Beside the fact that the menu command is now File | New | Other and the totally different UI, the role, content, and behavior of the Object Repository remains very similar to what's described here.

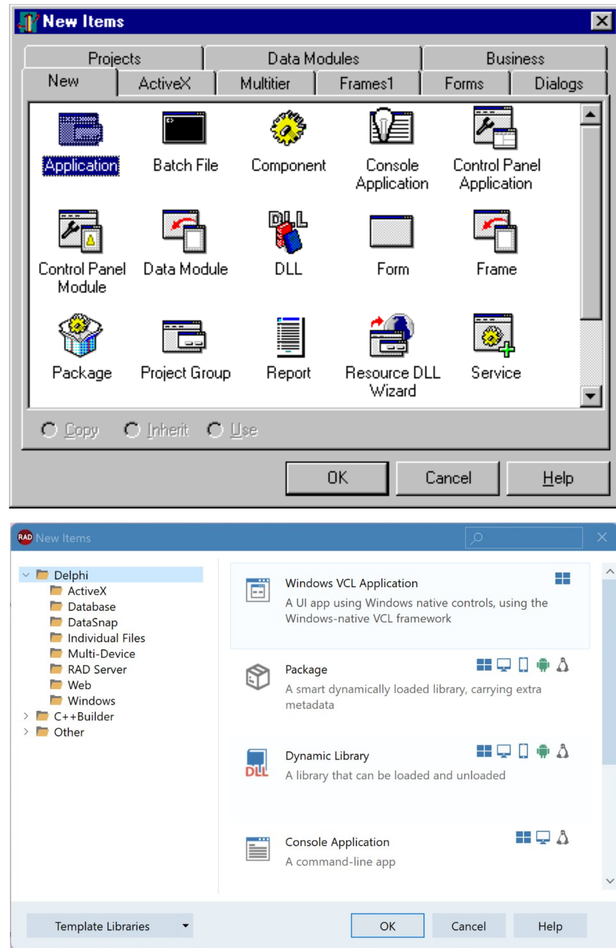


Figure 1.17:
The first page of the New dialog box, generally known as the “Object Repository”. Images captured in Delphi 5 and Delphi 12.

The Empty Project Template

When you start a new project, it automatically opens a blank form, too. If you want to base a new project on one of the form objects or Wizards, this is not what you want, however. To solve this problem, you can add an Empty Project template to the Gallery.

The steps required to accomplish this are simple[97]:

1. Create a new project as usual.
2. Remove its only form from the project.
3. Add this project to the templates, naming it *Empty Project*.

When you select this project from the Object Repository, you gain two advantages: You have your project without a form, and you can pick a directory where the project template's files will be copied. There is also a disadvantage—you have to remember to use the File ➢ Save Project As command to give a new name to the project, because saving the project any other way automatically uses the default name in the template.

To further customize the Repository, you can use the Tools ➢ Repository command. This opens the Object Repository dialog box, which you can use to move items to different pages, to add new elements, or to delete existing ones. You can even add new pages, rename or delete them, and change their order. An important element of the Object Repository setup is the use of defaults:

- Use the New Form check box below the list of objects to designate a form as the one to be used when a new form is created (File ➢ New Form).
- The Main Form check box indicates which type of form to use when creating the main form of a new application (File ➢ New Application) when no special New Project is selected.
- The New Project check box, available when you select a project, marks the default project that Delphi will use when you issue the File ➢ New Application command.

Only one form and only one project in the Object Repository can have each of these three settings marked with a special symbol placed over its icon. If no project is selected as New Project, Delphi creates a default project based on the form marked as Main Form. If no form is marked as the main form, Delphi creates a default project with an empty form.

When you work on the Object Repository, you work with forms and modules saved in the `OBJREPOS` subdirectory of the Delphi main directory[98]. At the same time, if you

97 I haven't actually tried these steps, but I assume they still work.

98 Current folder is still under the application folder, at C:\Program Files (x86)\Embarcadero\Studio\xx.0\ObjRepos

use a form or any other object directly without copying it, then you end up having some files of your project in this directory. It is important to realize how the Repository works, because if you want to modify a project or an object saved in the Repository, the best approach is to operate on the original files, without copying data back and forth to the Repository.

Installing New DLL Wizards

Technically, new wizards come in two different forms: They may be part of components or packages, or they may be distributed as stand-alone DLLs. In the first case, they would be installed the same way you install a component or a package. When you've received a stand-alone DLL, you should add the name of the DLL in the Windows Registry under the key Software\Borland\Delphi\5.0\Experts[99]. Simply add a new string key under this key, choose a name you like (it doesn't really matter what it is), and use as text the path and filename of the wizard DLL. You can look at the entries already present under the `Experts` key to see how the path should be entered.

What's Next?

This chapter has presented an overview of the new and more advanced features of Delphi 5 programming environment, including a number of tips and suggestions about some lesser-known features that were already available in previous Delphi versions. I didn't provide a step-by-step description of the IDE, partly because it is generally simpler to start *using* Delphi than it is to read about how to use it. Moreover, there is a detailed Help file describing the environment and the development of a new simple project; and you might already have some exposure to one of the past versions of Delphi or a similar development environment.

We haven't finished covering new features of Delphi 5 IDE, though. I'll discuss the new Data Module Designer in Chapter 10, new debugging features in Chapter 18, and TeamSource and the Integrated Translation Environment in Chapter 19[100]. But

99 The registry key is now HKEY_CURRENT_USER\Software\Embarcadero\BDS\xx.0\Experts

100 As already mentioned, these two features don't exist any more (or are not officially supported any more).

now we are ready to spend the next three chapters looking into the Object Pascal language and the VCL library. Then, in Part II, we'll start focusing on the user interface of applications and using the components available in Delphi.

Chapter 2: Object-Oriented Programming In Delphi

Most modern programming languages support *object-oriented programming* (OOP). OOP languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of

the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

A single chapter doesn't allow space for a full introduction to the principles of object-oriented programming and the Object Pascal language[101]. Instead, I will outline the key OOP features of the language and show how they relate to everyday Delphi programming. Even if you don't have a precise knowledge of OOP, the chapter will introduce each of the key concepts so that you won't need to refer to other sources.

note If you don't know the basics of the Pascal language (which is not covered in this book), you can refer to the online electronic version of the text *Essential Pascal* at `www.marcocantu.com`. The language has not changed significantly from Delphi 4 to Delphi 5.[102]

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. A *class* is a user-defined data type, which has a state (its representation) and some operations (its behavior). A class has some internal data and some methods, in the form of procedures or functions, and usually describes the generic characteristics and behavior of a number of similar objects.

An *object* is an instance of a class, or 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 object and class is the same as the one between variable and type.

To declare a new class data type in Object Pascal, with some local data fields and some methods, use the following syntax:

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
```

101 I've published a book covering the Delphi Object Pascal language in detail. It's called "Object Pascal Handbook" and it's available in print on Amazon. See www.marcocantu.com/objectpascalhandbook/ for more information.

102 My ebook Essential Pascal remains available for free.

```
end;
```

The function and the procedure declared above should be fully defined in the implementation portion of the same unit, including the class declaration. You can let Delphi generate a skeleton of the definition of the methods by using the Class Completion feature of the editor (simply press Ctrl+C while the cursor is within the class definition). You can tell the methods are part of the `TDate` class by class-name prefixing (using a dot in between), 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[103]
  Result := IsLeapYear (Year);
end;
```

note The convention in Delphi is to use the letter *T* as a prefix for the name of every class you write and every other type (*T* stands for *Type*). 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. In the book I'll try to stick with this convention.

Once the class has been defined, we can create an object and use it as follows:

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;
  // use
  ADay.SetValue (1, 1, 2000);
  if ADay.LeapYear then
    ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
  // destroy
  ADay.Free;
end;
```

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

103 Using today's notation, the unit name is now *System.SysUtils.pas*.

`ADay.Year`, although the first is a function call and the second a direct data access. As we'll see in the next chapter, the notation used by Object Pascal to access properties is again the same.

Delphi's Object Reference Model

In some OOP languages, declaring a variable of a class type creates an instance of that class. Object Pascal, instead, is based on an *object reference model*. The idea is that each variable of a class type, such as `ADay` in the code fragment above, does not hold the value of the object. Rather, it contains a reference, or a *pointer,* to indicate the memory location where the object has been stored.

note The object reference model is powerful yet easier to use than other models. Other OOP languages use similar models, notably Eiffel and Java[104]. In my opinion, adopting this model was one of the best design decisions made by the Delphi development team at Borland.

The only problem with this approach is that when you declare a variable, you don't create an object in memory, you only reserve the memory location for a reference to an object. Object instances must be created manually, at least for the objects of the classes you define. Instances of a component you place on a form are built automatically by Delphi.

To create an instance of an object, we can call its `Create` method, which is a constructor. As you can see in the last code fragment, the constructor is applied to the class, not to the object. Where does the `Create` method come from? It is a constructor of the class `TObject`, from which all the other classes inherit. Once you have created an object and you've finished using it, you need to dispose of it (to avoid filling up memory you don't need any more, which causes what is known as a *memory leak*). This can be accomplished by calling the `Free` method (yet another method of the `TObject` class), as demonstrated in the previous listing. As long as you create objects when you need them and free them when you're finished with them, the object reference model works without a glitch.

Private, Protected, and Public

A class can have any amount of data and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated,* inside the

104 Also C# uses the same machanism.

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 30. 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. Encapsulation is important because it allows the class writer 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. Of course, you are supposed to use methods to access the data, which is shielded from unauthorized access. This is the object-oriented approach to a classical programming concept known as *information hiding*.

Object Pascal has three access specifiers: `private`, `protected`, and `public`[105]. A fourth one, `published`, will be discussed in the next chapter. Here are the three basic ones:

- The `private` directive denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.
- The `public` directive 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` directive is used to indicate methods and fields with limited visibility. Only the current class and its subclasses can access `protected` elements. We’ll discuss this keyword again in the “Protected Fields and Encapsulation” section.

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 computation. Fields can be `protected` or `public` when you want an easy and direct access and you are fairly sure that their type definition is not going to change.

---

105 Two further access specifiers, *strict private* and *strict protected* were added to match the behavior of other OOP languages having no special rules for classes declared within the same unit or source code file. The strict versions of the access specifiers provide an even more robust encapsulation, but remain rarely used by Delphi developers.

note Instead of having `public` fields, you should generally use properties, as we'll see in detail in the next chapter. Properties are an extension to the encapsulation mechanism of other OOP languages and are very important in Object Pascal.

Access specifiers only restrict code outside your unit from accessing certain members of classes declared in the interface section of your unit. This means that if two classes are in the same unit, there is no protection for their private fields. Only by placing a class in the interface portion of a unit will you limit the visibility from classes and functions in other units to the public method and fields of the class.

As an example, 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`[106], 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 and make it harder to modify the internal implementation of a class. Access functions should be provided only if they are part of the logical interface of the class you are implementing.

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 nonleap years. What I'll do to make it easier to write the code is to change the internal implementation of the class to use Delphi's `TDateTime` type for the internal implementation. The class will change to

```
type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (m, d, y: Integer);
```

106 You could consider using *strict private*, instead.

```
    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 The `TDateTime` type is actually a floating-point number. The integral portion of the number indicates the date since 12/30/1899, the same base date used by OLE Automation and Microsoft applications. (Use negative values to express previous years.) The decimal portion indicates the time as a fraction. For example, a value of `3.75` stands for the second of January 1900, at 6:00 p.m. (three-quarters of a day). To add or subtract dates, you can simply add or subtract the number of days, which is much simpler than adding days with a day/month/year representation.

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 and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with an 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 one of the forms:

```
var
  Form1: TForm1;
  nClicks: Integer;
```

This works but has two problems. First, the data 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)
  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 the next chapter.

The Self Keyword

We've seen that methods are very similar to procedures and functions. The real difference is that methods have an implicit parameter, which is a reference to the current object. 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.

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 Delphi 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. (The C++ and Java languages[107] have a similar feature based on the keyword `this`.)

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 an extra hidden parameter, `Self`. Because all this happens behind the scenes, you do not need to know how `Self` works at this time.

note If you look at the definition of the `TMethod` data type in the VCL, you'll see that it is a record with a `Code` field and a `Data` field. The first is a pointer to the function's address in memory, the second the value of the `Self` parameter to use when calling that function address. We'll discuss method pointers in the next chapter.

Creating Components Dynamically

In Delphi, 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. (The difference between `Owner` and `Parent` properties is discussed in the next chapter.) In both cases, you have to supply the current form as parameter or value, and the best way to do this is to use the `Self` keyword.

To demonstrate this kind of code, I've written the CreateC[108] example (the name stands for *Create Component*). This program has a simple form with no components and a handler for its `OnMouseDown` event. I've used `OnMouseDown` because it receives as its parameter the position of the mouse click (differently from the `OnClick` event). 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.Left := X;
```

107 And C#, as well.

108 A lot of the example of the book are short, to comply with the 8 char file name limitations of the DOS word. I know it can sound odd, but it was common at the time. The original Delphi library unit names were all 8 char maximum for the same reason.

```
  Btn.Top := Y;
  Btn.Width := Btn.Width + 50;
  Btn.Caption := Format ('Button at %d, %d', [X, Y]);
end;
```

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 2.1. 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.

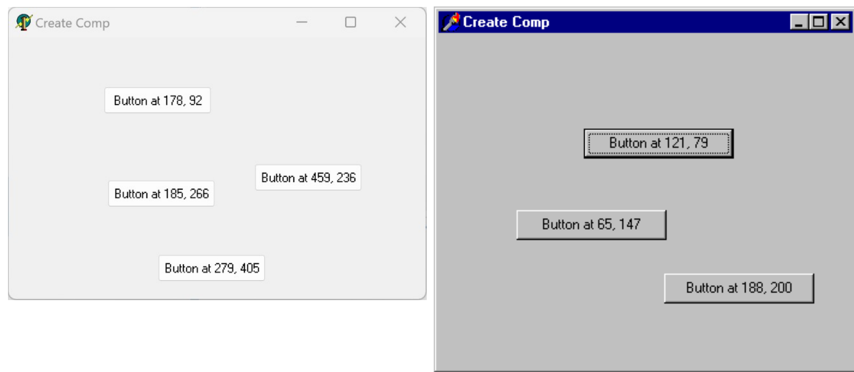


Figure 2.1: The output of the CreateC example, which creates Button components at run time. Images from the original book and captured today after rebuilding it in Delphi 12, running on Windows 11.

It is very common to write code like the above method using a `with`[109] statement, as in the following listing:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  with TButton.Create (Self) do
  begin
    Parent := Self;
    Left := X;
    Top := Y;
    Width := Width + 50;
    Caption := Format ('Button in %d, %d', [X, Y]);
  end;
end;
```

109 I've later changed my mind regarding the *with* statement: I don't recommend using it as it can lead to hard-to-spot bugs, given the scope of the symbols used is not always clear.

note 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, but if there are multiple instances of a form, using `Form1` would really be an error. In fact, if the `Form1` variable refers to the first form of that type being created, by clicking in another form of the same type, the new button will always be displayed in the first form. Its owner and Parent will be `Form1` and not the form the user has clicked onto. In general, referring to a particular instance of a class when the current object is required is a bad OOP practice.[110]

Constructors

To allocate the memory for the object, we call the `Create` method. This is a *constructor,* a special method that you can apply to a class to allocate memory for an instance of that class. The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. The default `TObject.Create` constructor initializes all the data of the new instance to zero.

If you want your instance data to start out with a nonzero value, 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: simply use the `constructor` keyword in front of it. Notice that in this case, you don't need to call `TObject.Create`: every constructor can automatically allocate the memory for an object instance simply by applying this special method to the related class.

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 technique 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.

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 constructor might bypass the initialization code you've provided. By defining a `Create` constructor with some parameters, you

110 I've defined a rule "Never use Form1" (or a reference to a specific form) in your code. While not an absolute rules, it's a very good idea in almost all cases.

replace the default definition with a new one and make its use compulsory. This is possible for generic classes, but it should be avoided for custom components. As we'll see in Chapter 13, when you inherit from `TComponent`, you should override the default `Create` constructor with one parameter and avoid disabling it.

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 called `Destroy`, which can perform some resource cleanup before an object is destroyed. Just as a constructor call allocates memory for the object, a destructor call frees the memory. Destructors are needed only for objects that acquire resources in their constructors or during their lifetime.

Instead of calling `Destroy` directly, a program should call `Free`, which calls `Destroy` only if the object exists—that is, if it is not `nil`[111]. Keep in mind, however, that calling `Free` doesn't set the object to `nil` automatically; this is something you should do yourself! The reason is that the object doesn't know which variables may be referring to it, so it has no way to set them all to `nil`.

note Delphi 5 has finally introduced a simple `FreeAndNil` procedure you can use to free an object and set its reference to `nil` at the same time. Simply call `FreeAndNil (Obj1)` instead of calling `Obj1.Free` and then setting `Obj1` to `nil`.[112]

Overloaded Methods and Constructors

Starting with Delphi 4, Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. By checking the parameters, the compiler can determine which of the versions of the routine you want to call.

There are two basic rules:

- 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.

111 Delphi's *nil* is the equivalent of *null* in other programming languages.

112 There have been discussions in the Delphi community whether *FreeAndNil* should be used or if its use implies a bad code architecture. In general, I tend to agree that the use of *FreeAndNil* should be a rare occurrence. Then a lcoal variable is about to get out of scope, setting it to *nil* has no benefit.

Overloading can be applied to global functions and procedures and to 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++ to allow the use of multiple constructors that each have the same name (the name of the class). In Object Pascal, this feature was considered unnecessary simply because multiple constructors can have different specific names. The increased integration of Delphi with C++Builder has motivated Borland to make this feature available in both languages. Technically, when C++Builder constructs an instance of a Delphi VCL class, it looks for a Delphi constructor named `Create` and nothing but `Create`. If the Delphi class has constructors by other names, they cannot be used from C++Builder code. Therefore, when creating classes and components you intend to share with C++Builder programmers, you should be careful to name all your constructors `Create` and distinguish between them by their parameter lists (using `overload`). This is not required by Delphi, but it is required for C++Builder to use your Delphi classes.

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 (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
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 as the default value today's date:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;

constructor TDate.Create (y, m, d: Integer);
```

```
begin
  fDate := EncodeDate (y, m, d);
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
```

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;
    function GetYear: Integer;
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: 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;
```

`GetText` returns a string with the formatted date, using the `DateToStr` function:

```
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 has a caption to display a date and six buttons, which can be used to modify the date. You can see the main form of the ViewDate example at run time in Figure 2.2. 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`.

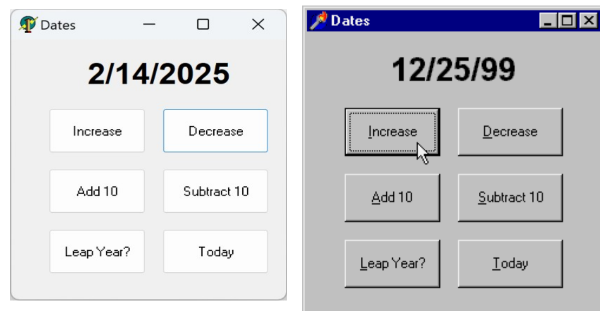


Figure 2.2: The output of the ViewDate example at start-up. Images captured now and in the original book.

The start-up code of this program is in the `OnCreate` event handler. In the corresponding method, we create an instance of the `TDate` class, initialize this object, and then show its textual description in the `Caption` of the label, as shown in Figure 2.2.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Create (1999, 12, 25);
  LabelDate.Caption := TheDay.GetText;
```

```
end;
```

`TheDay` is a private field of the class of the form, `TDateForm`. By the way, the name for the class is automatically chosen by Delphi when we change the `Name` property of the form to `DateForm`. The object is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  TheDay.Free;
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the `TheDay` object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  TheDay.SetValue (Date);
  LabelDate.Caption := TheDay.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);
var
  NewDay: TDate;
begin
  TheDay.Free;
  NewDay := TDate.Create;
  TheDay := NewDay;
  LabelDate.Caption := TheDay.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, Delphi simply copies the reference to the object in memory to the new object/reference.

If you really want to change the data inside an existing object, copy each field, or provide a specific method to copy the internal data. Some classes of the VCL have an `Assign` method, which does this *deep-copying*. To be more precise, all the VCL 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.

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 (including Borland), you might want to keep your changes separate.

A typical alternative 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. 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 *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, Delphi does this 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.

As a simple example of inheritance, we can change the ViewDate program 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 ViewD2 example.

```
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 *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 available in Windows; these names depend on the user's regional and language settings. Many of these regional settings are actually copied by Delphi into constants defined in the library, such as `LongMonthNames`, `ShortMonthNames`, and many others you can find under the *Currency and date/time formatting variables* topic in the Delphi Help file. Here is the `GetText` method, where *'dddddd'* stands for the long data format:

```
function TNewDate.GetText: string;
begin
  GetText := FormatDateTime ('dddddd', fDate);
end;
```

---

note Using regional information, the ViewD2 program automatically adapts itself to different Windows user settings. If you run this same program on a computer with regional settings referring to a language other than English, it will automatically show month names in that language. To test this behavior, you just need to change the regional settings; you don't need a new version of Windows. Notice that regional-setting changes immediately affect the running programs.

Once we have defined the new class, we need to use this new data type in the code of the form of the ViewD2 example. Simply define the `TheDay` object of type `TNewDate`, and call its constructor in the `FormCreate` method:

```
type
  TDateForm = class(TForm)
    ...
  private
    TheDay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TNewDate.Create (1998, 12, 25); // updated
  DateLabel.Caption := TheDay.GetText;
end;
```

Without any other changes, the new ViewD2 example 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 Delphi 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 2.3).

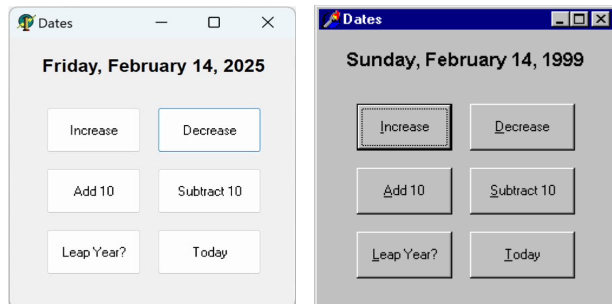


Figure 2.3: The output of the ViewD2 program, with the name of the month and of the day depending on Windows regional settings. Images captured now and in the original book.

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.

Accessing Protected Data of Other Classes

If you are new to Delphi 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 confusing.

We've seen that in Delphi, 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 example):

```
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
  ShowMessage (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. (This is what Delphi manuals and most Delphi books say.) 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, if you make a direct cast of the object to the new class and access the protected data through it, this is how the code will look:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  ShowMessage (Obj.GetValue);
  Obj.Free;
end;
```

This code compiles and works properly, as you can see by running the Protection program. How is it possible for this approach to work? Well, if you think about it, 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. As you would expect, if you move the declaration of the `TFake` class to a secondary unit, the program won't compile any more.

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 times when using this technique is the best solution, as you'll see by looking at the VCL source code and the code of many Delphi components. Two simple examples that come to mind are accessing the `Text` property of the `Tcontrol` class and the `Row` and `Col` positions of the `DBGrid` control. These two ideas are demonstrated by the TextProp and DBGridCol examples respectively. (These examples are quite advanced, so I suggest that only programmers with a good background of Delphi programming read them at this point in the text—other readers might come back later.)

Although the first example shows a reasonable example of using the typecast *cracker,* the DBGrid example of `Row` and `Col` is actually a counter example, one that illustrates the risks of accessing bits that the class writer chose not to expose. The row and column of a `DBGrid` do not mean the same thing as they do in a `DrawGrid` or `StringGrid` (the base classes). First, `DBGrid` does not count the fixed cells as actual cells (it distinguishes data cells from decoration), so your row and column indexes will have to be adjusted by whatever decorations are currently in effect on the grid (and those can change on the fly). Second, the `DBGrid` is a virtual view of the data. When you scroll up in a `DBGrid`, the data may move underneath it, but the currently selected row might not change.

This technique is often described as a *hack,* and it should be avoided whenever possible. The problem is not accessing protected data of a class in the same unit but declaring a class for the sole purpose of accessing protected data of an existing object of a different class! The danger of this technique is in the hard-coded typecast of an object from a class to a different one.

Inheritance and Type Compatibility

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 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;
```

As a general rule, 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!!!
```

Before we look at the implications of this important feature of the language, you can try out the Animals1 example, which defines the two simple `TAnimal` and `TDog` classes:

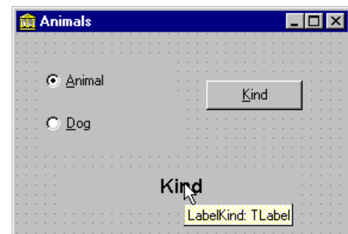
```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    Kind: string;
```

```
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

The two `Create` methods simply set the value of `Kind`, which is returned by the `GetKind` function. The form displayed by this example, shown in Figure 2.4, has a private field of type `TAnimal`. An instance of this class is created and initialized when the form is created and each time one of the radio buttons is selected.

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RbtnDogClick(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

Figure 2.4:
The form of the Animals1 example (in Delphi 5)



Finally, the Kind button calls the `GetKind` method for the current animal and displays the result in the label:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  KindLabel.Caption := MyAnimal.GetKind;
end;
```

Late Binding and Polymorphism

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 function or procedure resides. (This is 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 The term *polymorphism* is quite a mouthful. A glance at the dictionary tells us that in a general sense it refers to something having more than one form. In the OOP sense, then, it refers to the fact that there may be several versions of a given method and that a single method call can refer to any of these versions.

For example, suppose that a class and its subclass (let's say `TAnimal` and `TDog`) 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 example extends the Animals1 program 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:

```
uses
  MMSystem;

function TAnimal.Voice: string;
begin
  Voice := 'Voice of the animal';
  PlaySound ('Anim.wav', 0, snd_Async);
end;

function TDog.Voice: string;
begin
  Voice := 'Arf Arf';
  PlaySound ('dog.wav', 0, snd_Async);
end;
```

note This example uses a call to the `PlaySound` API function, defined in the MMSystem unit. The first parameter of this function is the name of the WAV sound file or the system sound you want to execute. The second parameter indicates an optional resource file containing the sound. The third parameter indicates (among other options) whether the call should be synchronous or asynchronous; that is, whether the program should wait for the sound to finish before continuing with the following statements.

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 program 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:

```
procedure TFormAnimals.BtnVerseClick(Sender: TObject);
begin
  LabelVoice.Caption := MyAnimal.Voice;
end;
```

In Figure 2.5, you can see an example of the output of this program. By running it, you'll also hear the corresponding sounds produced by the PlaySound API call.

Figure 2.5: The output of the Animals2 example. Image from the original book.

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.

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
  MyClass = class
    procedure One; virtual;
    procedure Two; {static method}
  end;

  MySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

There are typically two ways to override a method. One is to replace the method of the ancestor class with a new version. The other is to add some more code to the existing method. This can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write

```
procedure MySubClass.One;
begin
  // new code
  ...
  // call inherited procedure MyClass.One
  inherited One;
end;
```

You might wonder why you need to use the `override` keyword. In other languages, when you redefine a method in a subclass, you automatically override the original one. However, having a specific keyword allows the compiler to check the correspondence between the names of the methods of the ancestor class and the subclass (misspelling a redefined function is a common error in other OOP languages), check that the method was virtual in the ancestor class, and so on.

Furthermore, 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 code).

The support for overloading introduced in Delphi 4 added 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 new `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 Reintr example and experiment with it further.

Virtual versus Dynamic Methods

In Delphi, there are two different ways to activate late binding. You can declare the 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* (VMT, 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.

Message Handlers

A late-bound method can be used to handle a Windows message, too, although the technique is somewhat different. For this purpose Delphi provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter. 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, although there are 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.

note The ability to handle Windows messages and call API functions as you do when you are programming Windows with the C language may horrify some programmers and delight others. But in Delphi, when writing Windows applications, you will seldom need to use `message` methods. Only when you are writing complex components in Delphi will you have to deal with low-level messages and API calls.

Abstract Methods

The `abstract` keyword is used to declare methods that will be defined only in subclasses of the current class[113]. 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, Delphi's 32-bit 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 an exception, as demonstrated by the following Animals3 example.

note C++ and Java 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 `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 example, 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;
```

113 In recent versions of Delphi you can also use the abstract keyword to decorate a class as a whole, a syntax originally introduced in the .NET version of the compiler.

```
  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:

```
LabelVoice.Caption := MyAnimal.Voice;
```

How can we call the `Eat` method? We cannot apply it to an object of the `TAnimal` class. The statement

```
LabelVoice.Caption := MyAnimal.Eat;
```

generates the compiler error *Field identifier expected*.

To solve this problem, you can use run-time type information (RTTI) to cast the `TAnimal` object to a `TCat` or `TDog` object; but without the proper cast, the program will raise an exception. You will see an example of this approach in the next section. Adding the method definition to the `TAnimal` class is a typical solution to the problem, and the presence of the `abstract` keyword favors this choice.

Run-Time Type Information[114]

The Object Pascal type–compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned earlier, 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, it should be possible to call

---

114 The core form of RTTI, described in this section, is still available today. On top of it, there is now in Delphi an extended RTTI and a specific unit with classes you can use to access a large amount of type information at runtime.

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 "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 (discussed in the next chapter). 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 `TDog`. This means that if you test whether a `TDog` object is of type `TAnimal`, 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`).

Now that you know for sure that the animal is a dog, you can make a safe typecast (or type conversion). You can accomplish this direct cast 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 the second RTTI 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 raises an exception if the type of the object

is not compatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described at the end of this chapter).

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):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both RTTI operators are very useful in Delphi 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. 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 in Delphi, and I'll use it in a number of examples throughout the book. In Chapter 4, we'll discuss the `is` and `as` operators again, while focusing on some alternative RTTI techniques based on methods of the `TObject` class. The two RTTI 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 RTTI operators to complement it. *Do not use RTTI instead of polymorphism*. This is bad programming practice, and it results in slower programs. RTTI, in fact, has a 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 require just a memory lookup, which is much faster.

Visual Form Inheritance

To better understand derivation among classes, you can use visual form inheritance. In short, you can simply inherit a form from an existing one, adding new compo-

nents or altering the properties of the existing ones. But what is the real advantage of visual form inheritance?

Well, this 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 sub-classes. For example, if you prepare a standard parent form with a toolbar, a logo, default sizing and closing code, and the handlers of some Windows messages, you can then use it as the parent class for each of the forms of an application.

You can also use visual form inheritance to customize an application for different clients, 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 in 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 Delphi 5 includes a new feature, called *Frames,* which resembles visual form inheritance. In both cases you can work at design time 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 class and an instance. Frames will be discussed in detail in Chapter 4.

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 re-synchronize 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 multi-

ple properties, you can re-synchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

An alternative technique is to open the textual description of the inherited form and remove the line that changes the value of the property. (We will look at the structure of this file in a second.) 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 describe how visual form inheritance works, I've built a very simple example, called VFI. I'll describe step-by-step how to build it. First, start a new project, and add four buttons to its main form. Then select File ➢ New[115], and choose the page with the name of the project in the New Items dialog box (see Figure 2.6). Here you can choose the form from which you want to inherit. The new form has the same four 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 move the form and add the caption of one of the buttons, the textual description will change accordingly:

115 File | New | Other in recent versions of Delphi. The category is "Inheritable items", rather than the name of the project.

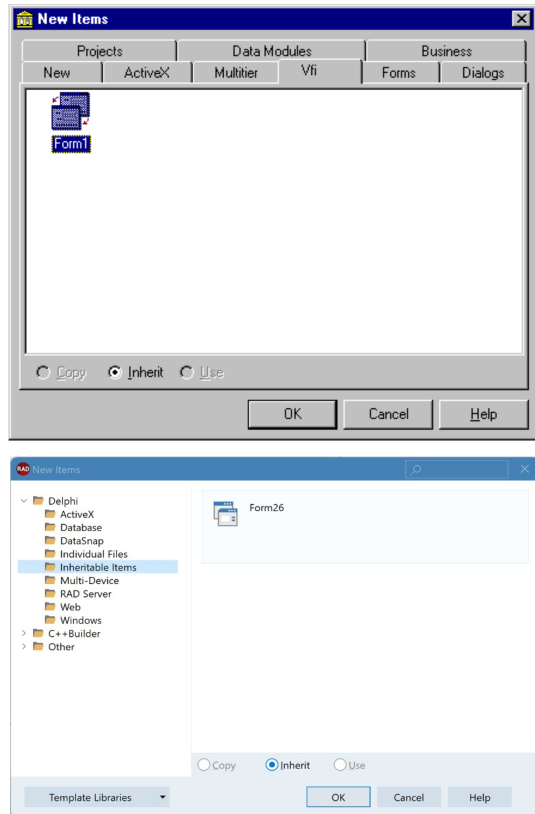


Figure 2.6:
The New Items dialog box allows you to create an inherited form. Images captured in Delphi 5 and Delphi 12.

```
inherited Form2: TForm2
  Left = 313
  Top = 202
  Caption = 'Form2'
  inherited Button2: TButton
    Caption = 'Beep...'
  end
end
```

Only the properties with a different value are listed (and by removing these properties from the textual description of the inherited form, you can reset them to the value of the base form, as I mentioned before). I've actually changed the captions of most buttons, as you can see in Figure 2.7.

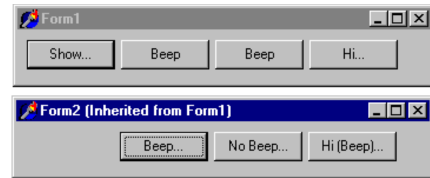


Figure 2.7: The two forms of the VFI example at run time. Image from the original book.

Each of the buttons of the first form has an `OnClick` handler, with simple code. The first button shows the inherited form calling its `Show` method; the second and the third buttons call the `Beep` procedure; and the last button displays a simple message calling `ShowMessage ('Hi')`.

What happens in the inherited form? First we should remove the Show button because the secondary form is already visible. However, we cannot delete a component from an inherited form. An alternative solution is to leave the component there but set its `Visible` property to `False`. The button will still be there but not visible (as you can guess from Figure 2.7). The other three buttons will be visible but with different handlers. This is simple to accomplish. If you select the `OnClick` event of a button in the inherited form (by double-clicking it), you'll get an empty method slightly different from the default one:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
end;
```

The `inherited` keyword stands for a call to the corresponding event handler of the base form. This keyword is always added by Delphi, even if the handler is not defined in the parent class (and this is reasonable, because it might be defined later) or if the component is not present in the parent class (which doesn't seem like a great idea to me). It is very simple to execute the code of the base form and perform some other operations:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hi');
end;
```

This is not the only choice. An alternative approach is to write a brand-new event handler and not execute the code of the base class, as I've done for the third button of the VFI example:

```
procedure TForm2.Button3Click(Sender: TObject);
```

```
begin
  ShowMessage ('Hi');
end;
```

Still another choice includes calling a base-class method after some custom code has been executed, calling it when a condition is met, or calling the handler of a different event of the base class, as I've done for the fourth button:

```
procedure TForm2.Button4Click(Sender: TObject);
begin
  inherited Button3Click (Sender);
  inherited;
end;
```

You probably won't do this very often, but you must be aware that you can. Of course, you can consider each method of the base form as a method of your form, and call it 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 real-world examples more complex than this book has room to explore. There is something else I want to show you here: *visual form polymorphism*.

Polymorphic Forms

The problem is simple. If you add an event handler to a form and then change it in an inherited form, there is no way to refer to the two methods using a common variable of the base class, because the event handlers use static binding by default.

Confusing? Here is an example, which is intended for experienced Delphi programmers. Suppose you want to build a bitmap viewer form and a text viewer form in the same program. The two forms have similar elements, a similar toolbar, a similar menu, an OpenDialog component, and different components for viewing the data. So you decide to build a base-class form containing the common elements and inherit the two forms from it. You can see the three forms at design time in Figure 2.8.

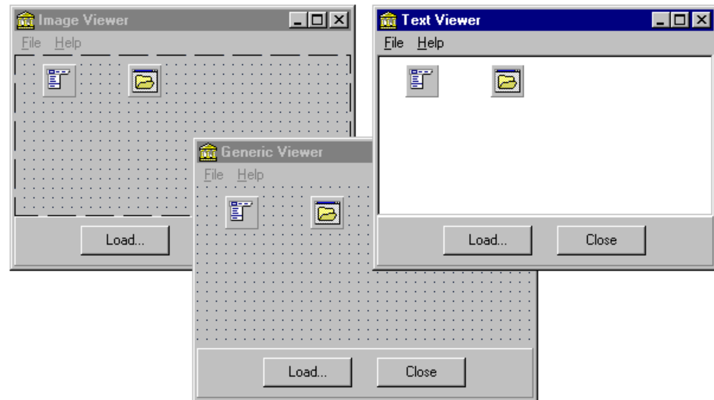


Figure 2.8: The base-class form and the two inherited forms of the PoliForm example at design time (in Delphi 5)

Here is the textual description of the main form:

```
object ViewerForm: TViewerForm
  Caption = 'Generic Viewer'
  Menu = MainMenu1
  object Panel1: TPanel
    Align = alBottom
    object ButtonLoad: TButton…
    object CloseButton: TButton…
  end
  object MainMenu1: TMainMenu
    object File1: TMenuItem...
      object Load1: TMenuItem...
      object N1: TMenuItem...
      object Close1: TMenuItem...
    object Help1: TMenuItem...
      object AboutPoliform1: TMenuItem...
  end
  object OpenDialog1: TOpenDialog...
end
```

The two inherited forms have only minor differences, but they feature a new component, either an image viewer (`TImage`) or a text viewer (`TMemo`):

```
inherited ImageViewerForm: TImageViewerForm
  Caption = 'Image Viewer'
  object Image1: TImage [0]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Bitmap file|*.bmp|Any file|*.*'
  end
end
inherited TextViewerForm: TTextViewerForm
```

```
  Caption = 'Text Viewer'
  object Memo1: TMemo [1]
    Align = alClient
  end
  inherited OpenDialog1: TOpenDialog
    Filter = 'Text files|*.txt|Any file|*.*'
  end
end
```

The main form includes some common code. The Close button and the File ➢ Close command call the `Close` method of the form. The Help ➢ About command shows a simple message box. The Load button of the base form has the follow-ing code:

```
procedure TViewerForm.ButtonLoadClick(Sender: TObject);
begin
  ShowMessage ('Error: File loading code missing');
end;
```

The File ➢ Load command, instead, calls another method:

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
  LoadFile;
end;
```

This method is defined in the `TViewerForm` class as

```
public
  procedure LoadFile; virtual; abstract;
```

Because this is an abstract method, we will need to redefine it (and override it) in the inherited forms:

```
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject);
  public
    procedure LoadFile; override;
  end;
```

The code of this `LoadFile` method simply uses the `OpenDialog1` component to ask the user to select an input file and loads it into the image component:

```
procedure TImageViewerForm.LoadFile;
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (
      OpenDialog1.Filename);
end;
```

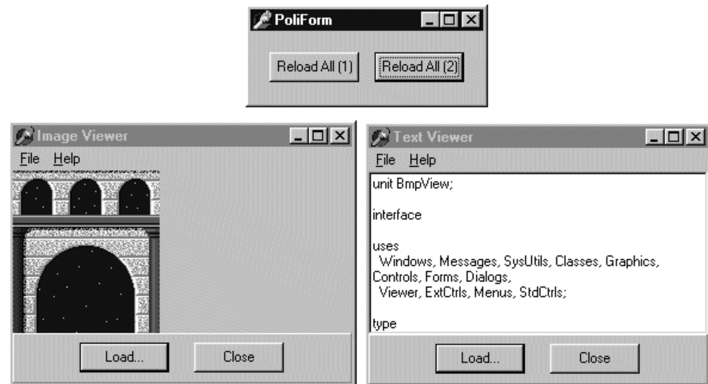
The other inherited class has similar code, loading the text into the memo component. The project has one more form, a main form with two buttons, used to reload the files in each of the viewer forms. The main form is the only form created by the project when it starts. The generic viewer form is never created: it is only a generic base class, containing common code and components of the two sub-classes. The forms of the two subclasses are created in the `OnCreate` event handler of the main form:

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create (Application);
  FormList [2] := TImageViewerForm.Create (Application);
  for I := 1 to 2 do
    FormList[I].Show;
end;
```

See Figure 2.9 for the resulting forms (with text and image already loaded in the viewers). `FormList` is a polymorphic array of forms, declared in the `TMainForm` class as:

```
private
  FormList: array [1..2] of TviewerForm;
```

Figure 2.9:
The PoliForm example at run time. Image from the original book.



Note that to make this declaration in the class, you need to add the Viewer unit (but not the specific forms) in the `uses` clause of the interface portion of the main form. The array of forms is used to load a new file in each viewer form when one of the two buttons is pressed. The handlers of the two buttons' `OnClick` events use different approaches:

```
procedure TMainForm.ReloadButton1Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].ButtonLoadClick (self);
end;

procedure TMainForm.ReloadButton2Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 2 do
    FormList [I].LoadFile;
end;
```

The second button simply calls a virtual method, and it will work without any problem. The first button calls an event handler and will always reach the generic `TFormView` class (displaying the error message of its `ButtonLoadClick` method). This happens because the method is static, not virtual.

Is there a way to make this approach work? Sure. Declare the `ButtonLoadClick` method of the `TFormView` class as virtual, and declare it as overridden in each of the inherited form classes, as we do for any other virtual method:

```
type
  TViewerForm = class(TForm)
    // components and plain methods...
    procedure ButtonLoadClick(Sender: TObject); virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;
...
type
  TImageViewerForm = class(TViewerForm)
    Image1: TImage;
    procedure ButtonLoadClick(Sender: TObject); override;
  public
    procedure LoadFile; override;
  end;
```

Simple, isn't it? This trick really works, although it is never mentioned in the Delphi documentation. This ability to use virtual event handlers is what I actually mean by visual form polymorphism.

What’s Next?

In this chapter, we have discussed the foundations of object-oriented programming in Object Pascal. We have considered the definition of classes, the use of methods, encapsulation, inheritance, polymorphism, and run-time type information. This is certainly a lot of information if you are a newcomer, but if you are fluent in another OOP language or if you’ve already used past versions of Delphi, you should be able to apply the topics covered in this chapter to your programming.

The next chapter continues our discussion of how Delphi implements OOP. It covers other language features, such as method pointers, class references, properties, events, and exceptions, which are particularly important to support Delphi’s visual development style. Chapter 3 also shows how to define your own components. Chapter 4 then focuses on the structure of the VCL (Visual Component Library) and discusses a few important classes.

Understanding the secrets of Object Pascal and the structure of the VCL is vital for becoming an expert Delphi programmer. These topics form the foundation of working with the VCL; after exploring them in the next two chapters, we’ll *finally* go on in Part II of the book to explore the development of real applications using all the various components provided by Delphi.

Chapter 3: Advanced Object Pascal

In the last chapter you've seen the foundations of the Object Pascal language used by Delphi: classes, objects, methods, constructors, inheritance, late binding, and run-time type information. Now we need to move one step further, by looking at some more advanced features of the language[116]. Some of the extensions discussed in this chapter, particularly the `published` keyword, properties, and events, are

116 Since the time this book was published, the Delphi language has been largely extended with significant improvements to classes (with strict access specifiers, nested types, class data, class properties, class constructors, and more), to the core language (inline routines, for-in loops, records with methods and operators overloading), and later with features opening up for different programming models, like generics, anonymous methods, and extended RTTI. All of these features build on top of the core capabilities of Delphi discussed in this book, which remain relevant.

strictly related to Delphi's visual programming model. In fact, while discussing these topics, I'll show you how to build a simple custom component.

Some other elements of Object Pascal, such as exceptions and interfaces, are not so closely related with the visual elements of Delphi. Still, it's important to know them, as well as a few other elements discussed in this chapter, to write correct code in your Delphi applications.

Class Methods and Class Data

When you define a field in a class, you actually specify that the field should be added to each object instance of that class. Each instance has its own independent representation (referred to by the `Self` pointer). In some cases, however, it might be useful to have a field that is shared by all the objects of a class.

Other object-oriented programming languages have formal constructs to express this, such as `static` in C++. But in Object Pascal, we can simulate this feature using the encapsulation provided at the unit level[117]. You can simply add a variable in the `implementation` portion of a unit, to obtain a class variable—a single memory location shared by all of the objects of a class.

If you need to access this value from outside the unit, you might use a method of the class. However, this forces you to apply this method to one of the instances of the class. An alternative solution is to declare a *class method*. A class method cannot access the data of any single object but can be applied to a class as a whole rather than to a particular instance. A class method is related to the class, not to its objects or instances (like a `static` member function in C++ or Java[118]).

To declare a class method in Object Pascal, you simply add the `class` keyword in front of it:

```
type
  MyClass = class
```

117 The lack of a formal declaration for class data has been filled with the "*class var*" construct, which let's you define true class data and works properly in case on inheritance and for generic classes, two areas in which the technique proposed in Mastering Delphi 5 falls short.

118 Another addition to the language, since the days this book was written, is the availability of "static" class methods, which are very similar to their C++, Java, or C# counterparts. The difference with the standard class methods in Delphi is that these have a hidden *self* parameter referring to the class, unlike static class methods which are for all purposes identical to global functions (to the point that they can be used as Windows callback functions).

```
    class function ClassMeanValue: Integer;
```

The use of class methods is not very common in Object Pascal, because you can obtain the same effect by adding a procedure or function to a unit declaring a class. Object-oriented purists, however, will definitely prefer the use of a class method over a routine unrelated to a class. And actually the VCL uses class methods quite often, although there are also many global subroutines. Notice that in Delphi, class methods can also be virtual[119], so they can be overridden and used to obtain polymorphism.

A Class with an Object Counter

When unit data is used to maintain general information related to the class (such as the number of objects created or a list of these objects), you can use class methods to access that data. That is exactly what the next example does.

The CountObj program is an extension of the CreateC example from the last chapter. The form is still quite bare, but I've added some new code. In particular I've added a brand-new class, which inherits from the `TButton` class of the VCL and adds a new feature, namely object counting. Here is the declaration of the new class:

```
type
  TCountButton = class (TButton)
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetTotal: Integer;
  end;
```

note What you see here is a perfectly working custom component. In this case, we won't register it and won't add it to Delphi's Components palette, even if this is not a particularly difficult operation. Customizing existing components can be really that simple! We'll cover this topic a little further in this chapter and in much more detail in Chapter 13.

Every time an object is created, the program increments the counter before calling the constructor of the base class. Every time an object is destroyed, the counter is decreased:

```
constructor TCountButton.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
```

119 This is a unique feature across programming languages, which combines nicely with another uncommon Delphi feature, class references.

```
  Inc (TotBtns);
end;

destructor TCountButton.Destroy;
begin
  Dec (TotBtns);
  inherited Destroy;
end;
```

The counter is a variable declared in the `implementation` portion of the unit and so is not accessible outside the unit. Only the class method allows us to read its current value. You can directly initialize this variable when it is defined:

```
implementation

var
  TotBtns: Integer = 0;

class function TCountButton.GetTotal: Integer;
begin
  Result := TotBtns;
end;
```

Now we can create objects of this new type by changing the code of the `FormMouseDown` method slightly:

```
begin
  with TCountButton.Create (Self) do
  begin
    Parent := Self;
    // same code as before...
```

Every time a `TCountButton` object is created, the current number of objects is displayed at the beginning of its caption. We can call the `GetTotal` class method for the newly created object (notice that we are inside a `with` statement), just as we call any plain method. However, we can call the same method without a valid object instance. This is what we do when the interval of a timer I've added to the form elapses:

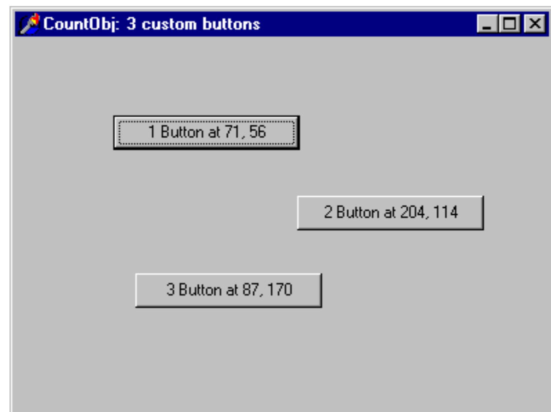
```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

The `Caption` property in this code refers to the caption of the form. You can see the effect of this call in Figure 3.1. The drawback of this example is that we can only create objects and never destroy them, so we see the total number of live objects always increasing and never reducing its value.

To see that the number of objects in existence goes down to zero, we can try to check the number of objects after the form has been destroyed, along with the TCountButton objects it owns. This is the code I've added at the end of the unit:

```
finalization
  MessageBox (0, PChar (Format (
    'There are %d CountButton objects',
    [TCountButton.GetTotal])),
    'Finalization', mb_OK);
end.
```

Figure 3.1: The output the CountObj example after a couple of TCountButton objects have been created. Image from the original book.



note In the finalization code above I had to use a Windows API function (MessageBox) instead of a Delphi procedure (such as ShowMessage). The reason is that the finalization code of the unit is executed after some of the Delphi global objects have been destroyed, so it is better not to rely on them.

The program simply displays a Windows message box indicating the number of objects in existence, a value obtained by calling the GetTotal class method. If you run the program, the number in the output is zero, although I have to say that this is not guaranteed but is due to the order in which objects are destroyed. The compiler uses a specific order for units initialization and finalization: starting with the project source code, the units referred to are initialized before and finalized after the units that refer to them. Typically, the project will initialize the Forms unit, which in turn initializes other VCL units, and then it initializes your form unit, which will first initialize the units describing the components you use (that is, those in the uses statement).

However, at first sight, it is not simply to determine when in this sequence the main form of the application and the components it owns are going to be destroyed. To test that everything actually works, I've added the same `MessageBox` call code in the handler of the `OnDestroy` event of the form, triggered before the form is destroyed.

If you run the program, you'll see that when the `FormDestroy` method is executed, all of the objects you've created still exist; but right after that, the objects are destroyed and the count decreases to zero. We'll see a more complete example, in which we'll destroy the buttons at run time, after we discuss method pointers in the following section "The Updated Counter Example."

Method Pointers

Another Delphi addition to the Object Pascal language is the concept of *method pointers*. A method pointer type is like a procedural type, but one that refers to a method[120]. Technically, a method pointer type is a procedural type that has an implicit `Self` parameter. In other words, a method pointer stores two addresses: the address of the method code and the address of an object instance (data). The address of the object instance will show up as `Self` inside the method body when the method code is called using this method pointer. This explains the definition of Delphi's generic `TMethod` type, a record with a `Code` field and a `Data` field.

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords `of object` at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

When you have declared a method pointer, such as the one above, you can declare a variable of this type and assign to it a compatible method of another object. What's a compatible method? One that has the same parameters as those requested by the method pointer type, such as a single Integer parameter in the example above.

At first glance, the goal of this technique may not be clear, but this is one of the cornerstones of Delphi component technology. The secret is in the word *delegation*. If someone has built an object that has some method pointers, you are free to change the object's behavior simply by assigning new methods to the pointers. Does this sound familiar? It should.

---

120 The language now offers a different, related feature: anonymous methods. Methods pointers remain the foundation for event handlers.

When you add an `OnClick` event handler for a button, Delphi does exactly that. The button has a method pointer, named `OnClick`, and you can directly or indirectly assign a method of the form to it. When a user clicks the button, this method is executed, even if you have defined it inside another class (typically, in the form).

What follows is a listing that sketches the code actually used by Delphi to define the event handler of a button component and the related method of a form:

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

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;
```

Now inside a procedure, you can write

```
MyButton.OnClick := Form1.Button1Click;
```

The only real difference between this code fragment and the code of the VCL is that `OnClick` is a property name, and the actual data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector, in fact, is nothing more than a property of a method pointer type.

This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it. For example, we could add to the form of the Counter example the following method:

```
procedure TForm1.ButtonClick (Sender: TObject);
begin
  ShowMessage ('Button pressed');
end;
```

and then write for each newly created button the following code:

```
with TCountButton.Create (Self) do
begin
  OnClick := ButtonClick;
```

With this code each of the buttons will react to a click of the mouse by showing a common message, because all components share the same handler. However, we

can use the `Sender` parameter of the event to customize it for each button. This is what I'll do in the example discussed in the sidebar "The Updated Counter Example," which is an even more complete extension of the Counter program.

The Updated Counter Example

Now that we know how to use method pointers, we can update the CountObj example by using them. The name of the new example is CountOb2. Its purpose is to add a handler for the `OnKeyPress` event of the new objects a user creates dynamically. Add the following code in the form class declaration:

```
procedure ButtonKeyPress(Sender: TObject; var Key: Char);
```

The parameters are those required for an event of this kind. If you select the `OnKeyPress` event for a component of a form and press the F1 key to invoke the Help file, you'll find the following declaration:

```
TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
property OnKeyPress: TKeyPressEvent;
```

As you can see in this last line, the event is based on the `TKeyPressEvent` method pointer type, listed in the line before. Therefore, we need to write a method that complies with this method pointer type, like the one presented in the previous section.

To connect this method with the `OnKeyPress` event of the buttons we create dynamically, we need just one line of code in the `FormMouseDown` method:

```
with TCountButton.Create (Self) do
begin
  ...
  // set the event handler
  OnKeyPress := ButtonKeyPress;
  // grab the input focus
  SetFocus;
end;
```

The second line of code moves the input focus to the newly created button, so that subsequent keyboard input will be directed to it.

Now we can write the code of the `ButtonKeyPress` method. Press the Ctrl+Shift+C key combination to activate Delphi's Code Completion, and then fill the method declaration with some actual code. In this example, we should destroy the button when the user presses the Backspace key. Because keyboard input is sent to the control that has the input focus, you can simply click a button or use the Tab key to select the button you want to destroy; then press the Backspace key.

The first approach I tried in developing this application was simply to destroy the object passed as `Sender` parameter, which is the object that received the event:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
  var Key: Char);
begin
  if Key = #8 then
    Sender.Free
end;
```

This code generates an exception. We cannot destroy an object while we are processing one of its events. Instead, we must delay the object destruction. There are basically two approaches. We can save the object we want to destroy in a private field of the form class and later destroy it, inside some code periodically activated by a timer. You can find this code in the CountOld example. Notice that the use of the timer causes a little flaw in the program: if two backspace keys are processed before the timer fires, only one button is going to be destroyed.

The second approach, implemented in the CountOb2 example, is to send a custom Windows message (such as `wm_User`) to the form using the `PostMessage` API function. This introduces a delay, because the message has to reach the window and will be retrieved and elaborated after the current event handler has completed its execution. To follow this second approach, we can write the following handler for the `OnKeyPress` event of each new button:

```
procedure TForm1.ButtonKeyPress(Sender: TObject;
  var Key: Char);
begin
  // if user pressed backspace
  if Key = #8 then
  begin
    // set this as the object to destroy
    ToDestroy := Sender as TButton;
    // post message to perform destruction
    PostMessage (Handle, wm_User, 0, 0);
  end;
end;
```

In this code `ToDestroy` is a private field of the form of the `TButton` data type. This field is automatically set to `nil` (no object to destroy) when the form is first created (this is the default initialization for class fields). When the user presses the Backspace key, the current button object (the `Sender` of the `ButtonKeyPress` method) is stored in the `ToDestroy` field. At this point, the `PostMessage` Windows API call sends a message to the current window (identified by the value of its `Handle` property). The handler of this message is defined in the form class as follows:

```
type
```

```
  TForm1 = class(TForm)
    ...
  private
    ToDestroy: TButton;
  public
    procedure WmUser (var Msg: TMessage); message wm_User;
```

Now we can look at the code of this method, in which the program can double-check whether there is a button to destroy before destroying it and setting it to `nil`:

```
procedure TForm1.WmUser (var Msg: TMessage);
begin
  // if there is an object to destroy
  if Assigned (ToDestroy) then
  begin
    // moves the input focus to the next control
    SelectNext (ToDestroy, True, True);
    // destroy the object and set the reference to nil
    FreeAndNil (ToDestroy);
  end;
  // update the form caption
  Caption := Format ('CountObj: %d custom buttons',
    [TCountButton.GetTotal]);
end;
```

To make the program behave a little better before destroying an object, I moved the input focus to the next control by calling the `SelectNext` method. Then the program calls the `FreeAndNil` procedure, which calls the `Free` method of the object, which in turn invokes the destructor `Destroy`. Because the destructor is virtual, the program invokes the overridden destructor of the `TCountButton` class, which decrements the object counter. For this reason I've placed the code that destroys the object before the code that updates the form caption. Before calling `Free`, `FreeAndNil` sets the `ToDestroy` reference to `nil`.

More about freeing objects and memory management is in the section “Objects and Memory,” later in this chapter.

Class References

Having looked at several topics related to methods, we can now move on to the topic of *class references* and extend our example of dynamically creating components even further. The first point to keep in mind is that a class reference isn't a class, it isn't an object, and it isn't a reference to an object; it is simply a reference to a class type.

A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code might make this a little clearer. Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. Actually, there are not many such expressions, but the few cases are interesting. The simplest case is the creation of an object. We can rewrite the two lines above as follows:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

This time I've applied the `Create` constructor to the class reference instead of to an actual class; I've used a class reference to create an object of that class.

---

note Class references remind us of the concept of *metaclass* available in other OOP languages. In Object Pascal, however, a class reference is not itself a class but only a type pointer. Therefore, the analogy with metaclasses (classes describing other classes) is a little misleading. Actually, `TMetaclass` is also the term used in C++Builder.

Class reference types wouldn't be as useful if they didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef` above, you can then assign to it that specific class and any subclass. So if `MyNewClass` is a subclass of my class, you can also write

```
AClassRef := MyNewClass;
```

Delphi declares a lot of class references in the run-time library and the VCL, including the following:

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

```
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Delphi, because every class is ultimately derived from `TObject`. The `TFormClass` reference, instead, is used in the source code of most Delphi projects. The `CreateForm` method of the `Application` object, in fact, requires as parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference, the second is a variable that stores a reference to the created object instance.

Finally, when you have a class reference you can apply to it the class methods of the related class. Considering that each class inherits from `TObject`, you can apply to each class reference some of the methods of `TObject`, including `InstanceSize`, `ClassName`, `ParentClass`, and `InheritsFrom`. I'll discuss these class methods and other methods of `TObject` class in the next chapter.

Creating Components Using Class References

What is the *practical* use of class references in Delphi? Being able to manipulate a data type at run time is a fundamental element of the Delphi environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what Delphi does for you behind the scenes.)

To give you a better idea of how class references work, I've built an example named ClassRef. The form displayed by this example is quite simple. It has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, push buttons, and edit boxes.

To make this program run properly, you need to change the names of the three components. The form must also have a class reference field:

```
private
  ClassRef: TControlClass;
  Counter: Integer;
```

The first field stores a new data type every time the user clicks one of the three radio buttons. Here is one of the three methods:

```
procedure TForm1.RadioButtonRadioClick(Sender: TObject);
begin
  ClassRef := TRadioButton;
end;
```

The other two radio buttons have `OnClick` event handlers similar to this one, assigning the value `TEdit` or `TButton` to the `ClassRef` field. A similar assignment is also present in the handler of the `OnCreate` event of the form, used as an initialization method.

The interesting part of the code is executed when the user clicks the form. Again, I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  NewCtrl: TControl;
  MyName: String;
begin
  // create the control
  NewCtrl := ClassRef.Create (Self);
  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := False;
  // set parent and position
  NewCtrl.Parent := Self;
  NewCtrl.Left := X;
  NewCtrl.Top := Y;
  // compute the unique name (and caption)
  Inc (Counter);
  MyName := ClassRef.ClassName + IntToStr (Counter);
  Delete (MyName, 1, 1);
  NewCtrl.Name := MyName;
  // now show it
  NewCtrl.Visible := True;
end;
```

The first line of the code for this method is the key. It creates a new object of the class data type stored in the `ClassRef` field. We accomplish this simply by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give it a name (which is automatically used also as `Caption` or `Text`), and make it visible.

Notice in particular the code used to build the name; to mimic Delphi's default naming convention, I've taken the name of the class with the expression `ClassRef.ClassName`, using a class method of the `TObject` class. Then I've added a

number at the end of the name and removed the initial letter of the string. For the first radio button, the basic string is `TRadioButton`, plus the *1* at the end, and minus the *T* at the beginning of the class name—*RadioButton1*. Sounds familiar?

You can see two examples of the output of this program in Figure 3.2. Notice that the naming is not exactly the same as used by Delphi. Delphi uses a separate counter for each type of control; I've used a single counter for all of the components. If you place a radio button, a push button, and an edit box in a form of the ClassRef example, their names will be *RadioButton1, Button2,* and *Edit3*.

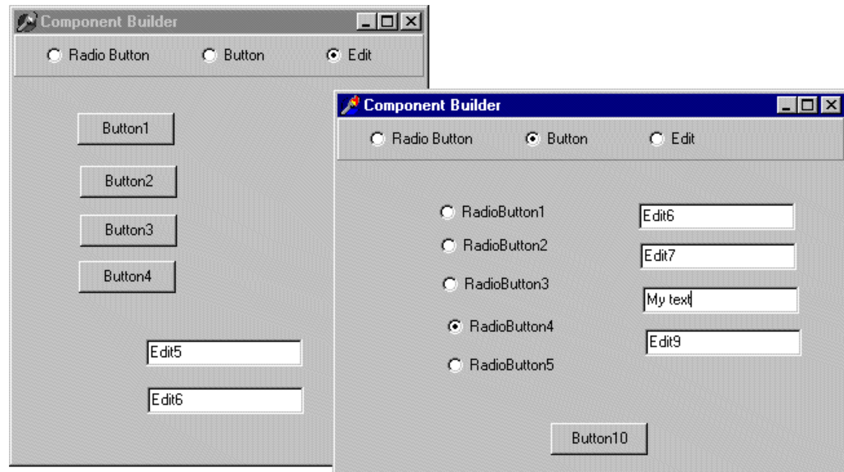


Figure 3.2:
Two examples of the output of the ClassRef example, in two different windows

Objects and Memory

Memory management in Delphi is subject to two simple rules: You must destroy every object you create, and you must destroy each object only once. Delphi supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area):

- Every time you create an object, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects, until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form)

becomes responsible for destroying all the objects it owns. In other words, when you free the form, it frees all the components it owns. So, if you create a component and give it an owner, you don't have to remember to destroy it.

- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables (discussed at the end of this chapter), Delphi automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

We'll see how the issue of memory management affects actual examples when discussing applications with multiple forms in Part II of the book.

Destroying Objects Only Once

Another problem is that if you call the destructor of an object twice, you get an error. A *destructor* is a method that de-allocates an object's memory. We can write code for a destructor, generally overriding the default `Destroy` destructor, to let the object execute some code before it is destroyed. In your code, of course, you don't have to handle memory de-allocation—this is something Delphi does for you.

`Destroy` is simply a virtual destructor of the `TObject` class. Most of the classes that require custom clean-up code when the objects are destroyed override this virtual method. The reason you should never define a new destructor is that objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor (possibly the overridden version) for you.

As I've just mentioned, `Free` is simply a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor.

note You might wonder why you can safely call `Free` if the object reference is `nil`, but you can't call `Destroy`. The reason is that `Free` is a known method at a given memory location, whereas the virtual function `Destroy` is determined at run time by looking at the type of the object, a very dangerous operation if the object doesn't exist any more.

Here is its pseudo-code (the actual code in the RTL source code files is written in assembler):

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

Next, we can turn our attention to the `Assigned` function. When we pass a pointer to this function, it simply tests whether the pointer is `nil`. So the following two statements (from the CountOb2 example) are equivalent, at least in most cases:

```
if Assigned (ToDestroy) then ...
if ToDestroy <> nil then ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code

```
ToDestroy.Free;
if ToDestroy <> nil then
  ToDestroy.DoSomething;
```

the test will be satisfied, and you'll get an error on the line with the call to the method of the object. It is important to realize that calling `Free` doesn't set the object to `nil`.

Automatically setting an object to `nil` is not possible. You might have several references to the same object, and Delphi doesn't track them. At the same time, within a method (such as `Free`) we can operate on the object, but we know nothing about the object reference—the memory address of the variable we've used to call the method. In other words, inside the `Free` method or any other method of a class, we know the memory address of the object (`Self`), but we don't know the memory location of the variable referring to the object, such as `ToDestroy`. Therefore, the `Free` method cannot affect the `ToDestroy` variable.

However, when we call an external procedure, such as `FreeAndNil` in Delphi 5, the procedure knows about the object reference, passed as a parameter, and can act on it. Here is Delphi code for `FreeAndNil`[121]:

```
procedure FreeAndNil(var Obj);
var
  P: TObject;
begin
  P := TObject(Obj);
  // clear the reference before destroying the object
  TObject(Obj) := nil;
  P.Free;
end;
```

To sum things up, here are a couple of guidelines:

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.

121 The code today is slightly different, as it assigns the nil value before freeing the object, to be safe in multi-threaded applications.

- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

Passing and Copying Objects

Another important element to discuss is passing objects as parameters or assigning an object to another one. If you write

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

you don't create a new object but rather a new reference to the same object in memory. There is only one object in memory, and both the `Button1` and `Button2` variables refer to it. The same happens if you pass an object as parameter to a function: you don't create a new object, but you refer to the same one in two different places of the code.

For example, by writing this procedure and calling it as follows, you'll modify the caption of `Button1`, or `Button` if you prefer:

```
procedure ChangeCaption (Button: TButton; Text: string);
begin
  Button.Caption := Text;
end;

// call...
ChangeCaption (Button1, 'Hello')
```

What if you need to create a new object, instead? You'll basically have to create it and then copy each of the relevant properties. Some classes, notably some VCL classes derived from `TPersistent`, define an `Assign` method to copy the data of an object. For example, you can write

```
ListBox1.Items.Assign (Memo1.Lines);
```

Even if you assign those properties directly, Delphi will execute a similar code for you. In fact, the `SetItems` method connected with the items property of the list box calls the `Assign` method of the `TStringList` class representing the actual items of the listbox.

You can use complex techniques based on streaming to clone a component in memory, but most of the time, creating a new object of the same type as the current one and assigning a few properties to it might do the trick. To do this, you can ask the

component its class and then use the class reference to create a new object of that type. Here is the code (extracted from the ObjClone example), which clones the `Sender` object:

```
procedure TForm1.ClickComp(Sender: TObject);
var
  ControlText: string;
begin
  with TControlClass (Sender.ClassType).Create (Self) do
  begin
    Parent := (Sender as TControl).Parent;
    Left := (Sender as TControl).Left + 10;
    Top := (Sender as TControl).Top + 10;
    SetLength (ControlText, 50);
    (Sender as TControl).GetTextBuf(
      PChar(ControlText), 50);
    ControlText := PChar(ControlText) + ‘ *’;
    SetTextBuf (PChar (ControlText));
  end;
end;
```

This method takes the class of the `Sender` object, the component clicked by the user, and calls the `Create` constructor. To call the `Create` constructor of the `TControl` class instead of calling that of the `TObject` class, the program has to cast the class reference to the proper type. When we cast to `TControlClass` and then call `Create`, the result is an object of class `TControl`. This object is used inside the `with` statement, and the program sets its `Parent`, `Left`, and `Top` properties using information extracted from the `Sender` control.

At the end of the `with` statement, the program extracts the text of the `Sender` object, using the `GetTextBuf` method, which is available for every control. In fact, the `Text` and `Caption` properties aren't defined inside the `TControl` class. After adding an asterisk to the string, the program uses the string as the new text of the control, again calling the `SetTextBuf` method of the `TControl` class.

You can see the effect of cloning on some of the controls in Figure 3.3. The ObjClone program is also capable of cloning an entire form, using a similar technique.

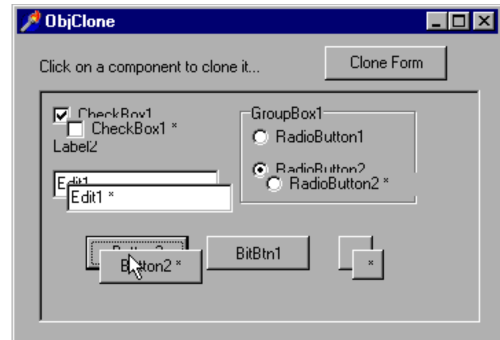


Figure 3.3:
The ObjClone example. Image from the original book.

Handling Exceptions

The last interesting feature of Object Pascal we will cover in this chapter is *exception handling*. The idea of exceptions is to make programs more robust by adding the capability of handling software or hardware errors 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 Delphi components. At run time, Delphi 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, Delphi handles it, by displaying a standard error message and trying to continue the program.

The whole mechanism is based on four 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 statements, with this syntax form:

  ```
  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 generate an exception. Most exceptions you'll encounter in your Delphi programming will be generated by the system, but you can also `raise` exceptions in your own code when it discovers invalid or inconsistent data at run time. The `raise` keyword can also be used inside a handler to *re-raise* an exception; that is, to propagate it to the next handler.

Here is an example of a simple protected block:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
      Result := 0;
  end;
end;
```

In the exception-handling statement, we catch the `EDivByZero` exception, which is defined by Delphi. There are a number of these exceptions referring to run-time problems (such as a division by zero or a wrong dynamic cast), to Windows resource problems (such as out-of-memory errors), or to component errors (such as a wrong index). Programmers can also define their own exceptions; simply create a new sub-class of the default exception class or one of its sub-classes:

```
type
  EArrayFull = class (Exception);
```

When you add a new element to an array that is already full (probably because of an error in the logic of the program), you can raise the corresponding exception by creating an object of this class:

```
if MyArray.Full then
  raise EArrayFull.Create ('Array full');
```

This `Create` method (inherited from the `Exception` class) has a string parameter to describe the exception to the user. 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.

The code presented in the previous excerpts is part of a sample program, called Except. Some of the routines have actually been slightly modified, as in the following `DivideTwicePlusOne` function:

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
  try
    // error if B equals 0
    Result := A div B;
    // do something else... skip if exception is raised
    Result := Result div B;
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero corrected',
        mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
      MessageDlg (E.Message,
        mtError, [mbOK], 0);
    end;
  end; // end except
end;
```

note When you run a program in the debugger, the debugger will stop the program by default when an exception is encountered. 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. In the case of the Except test program, however, this behavior will confuse the program's execution. In fact, even if the code is prepared to properly handle the exception, the debugger will stop the program execution at the source code line closest to where the exception was raised. Then, moving step-by-step through the code, you can see how it is handled. If you just want to let the program run when the exception is properly handled, run the program with the "Run without debugging" menu command.

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. For this reason, you need to place the broader handlers (the handlers of the ancestor `Exception` classes) at the end.

In fact, using a hierarchy of exceptions, a handler is also called for the subclasses of the type it refers to, as any procedure will do. This is polymorphism in action again. But keep in mind that using a handler for every exception, such as the one above, is not usually a good choice. It is better to leave unknown exceptions to Delphi. The default exception handler in the VCL displays the error message of the exception

class in a message box, and then resumes normal operation of the program. You can actually modify the normal exception handler with the `Application.OnException` event, as demonstrated in the ErrorLog example later in this chapter.

Another important element of the code above is the use of the exception object in the handler (see `on E: Exception do`). The object `E` of class `Exception` receives the value of the exception object passed by the `raise` statement. When you work with exceptions, remember this rule: You raise an exception by creating an object and handle it by indicating its type. This has an important benefit, because as we have seen, when you handle a type of exception, you are really handling exceptions of the type you specify as well as each descendant type.

Delphi defines a hierarchy of exceptions, and you can choose to handle each specific type of exception in a different way or handle groups of them together. You can find a list of the Delphi exception classes on www.marcocantu.com/d5ref[122].

Exceptions and the Stack

When the program raises an exception and the current routine doesn't handle it, what happens to your 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 message boxes to the code, to be informed when a certain source code statement is executed. In the next example, Except2, I've followed this second approach.

For example, when you press the Raise2 button in the form of the Except2 example, an exception is raised and not handled, so that the final part of the code will never be executed:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  // unguarded call
  AddToArray (24);
  ShowMessage ('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:

122 That page and that list don't exists any more.

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
    ShowMessage ('Program never gets here');
  except
  on EArrayFull do
      ShowMessage ('Handle the exception');
  end;
  ShowMessage ('ButtonRaise1Click call completed');
end;
```

The last `ShowMessage` 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.

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 Except3 example), which performs some time-consuming operations and uses the hourglass cursor to show the user that it's doing something:

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;
  MessageDlg ('Total: ' + IntToStr (J),
    mtInformation, [mbOK], 0);
  Screen.Cursor := crDefault;
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 default cursor. This is what a `try-finally` block is for:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J),
      mtInformation, [mbOK], 0);
  finally
    Screen.Cursor := crDefault;
  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. Strangely enough, this is not possible. A `try` block can be followed by either an `except` or a `finally` statement but not both of them at the same time. 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 Except3 example:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Screen.Cursor := crHourglass;
  J := 0;
  try try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    MessageDlg ('Total: ' + IntToStr (J),
      mtInformation, [mbOK], 0);
  finally
    Screen.Cursor := crDefault;
  end;
  except
    on E: EDivByZero do
    begin
      // re-raise the exception with a new message
      raise Exception.Create ('Error in Algorithm');
```

```
    end;
  end;
end;
```

You should always protect blocks with the `finally` statement, to avoid resource or memory leaks in case an exception is raised. Handling the exception is probably less important, since Delphi can survive most of them.

Logging Errors

Most of the time, you don't know which operation is going to `raise` an exception, and you cannot (and should not) wrap each and every piece of code in a `try-except` block. An alternative approach is to let Delphi handle all the exceptions and pass them all to you, by handling the `OnException` event of the global `Application` object. In early versions of Delphi you could handle this event by writing a proper method and connecting in the code. Now Delphi provides the ApplicationEvents component we can use to build this example. (More on the global `Application` object and the ApplicationEvents component in Chapter 6).

In the ErrorLog example, I've added to the main form a copy of the Application-Events component, and added a handler for its `OnException` event:

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);
var
  Filename: string;
  LogFile: TextFile;
begin
  // prepares log file
  Filename := ChangeFileExt (Application.Exename, '.log');
  AssignFile (LogFile, Filename);
  if FileExists (FileName) then
    Append (LogFile) // open existing file
  else
    Rewrite (LogFile); // create a new one

  // write to the file and show error
  Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);
  if not CheckBoxSilent.Checked then
    Application.ShowException (E);

  // close the file
  CloseFile (LogFile);
end;
```

note The ErrorLog example uses the simple text file support provided by the traditional Turbo Pascal TextFile data type. You can assign a text file variable to an actual file and then simply read or write it. You can find more on TextFile operations in the book *Essential Pascal* (available on `www.marcocantu.com/epascal`).[123]

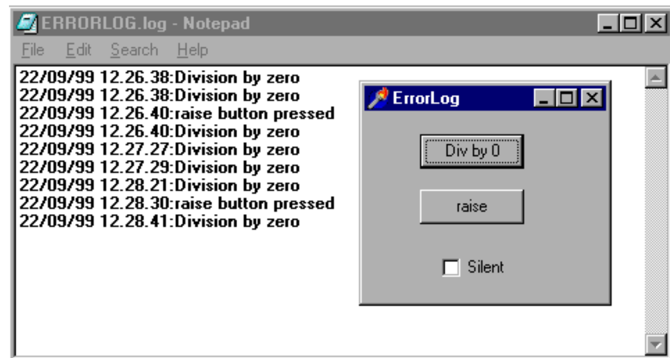
In the global exceptions handler, you can write to the log, for example, the date and time of the event, and also decide whether to show the exception as Delphi usually does (executing the `ShowException` method of the `TApplication` class). In fact, Delphi by default executes `ShowException` only if there is no `OnException` handler installed.

Finally, remember to close the file, flushing the buffers, every time the exception is handled or when the program terminates. I've chosen the first approach to avoid keeping the log file open for the lifetime of the application, potentially making it difficult to work on it. You can accomplish this in the `OnDestroy` event handler of the form:

```
procedure TFormLog.FormDestroy(Sender: TObject);
begin
  CloseFile (LogFile);
end;
```

The form of the program includes a check box to determine its behavior and two buttons generating simple exceptions. In Figure 3.4, you can see the ErrorLog program running and a sample exceptions log open in Notepad.

Figure 3.4:
The ErrorLog example and the log it produces. Image from the original book.



123 Using the very old TextFile type is not recommended at all, however the code does work today in Delphi 12.

The published Access Specifier

Along with the `public`, `protected`, and `private` access directives, you can use a fourth one, called `published`. A `published` field or method is available not only at run time but also at design time. In fact, every component in the Delphi Components Palette has a published interface that is used by some Delphi tools, in particular the Object Inspector. A regular use of `published` fields is important when you write components. Usually, the `published` part of a component contains no fields or methods but has a new element of the language: properties.

When Delphi generates a form, it places the definitions of its components and methods in the first portion of its definition, before the `public` and `private` keywords. These fields and methods of the initial portion of the class are `published`. The default is `published` when no special keyword is added before an element of a component class.

note To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. As this directive is used in the `TPersistent` class, most classes of the VCL and all of the component classes default to `published`. However, non-component classes in Delphi (such as `TStream` and `TList`) are compiled with `$M-` and default to public visibility.

The methods assigned to any event should be `published` methods, and the fields corresponding to your components in the form should be `published` to be automatically connected with the objects described in the DFM file and created along with the form. Only the components and methods in the initial `published` part of your form declaration can show up in the Object Inspector (in the list of components of the form or in the list of the available methods displayed when you select the drop-down list for an event).

Defining Properties

Now that we have looked at the `published` keyword, we can start focusing on other extensions of the Object Pascal language specifically tailored for visual, component-based programming. This section covers properties; later on, we'll look at events and build a first simple component.

Properties are attributes that determine the status and behavior of an object. A property is basically a name that is mapped to some `read` and `write` methods or that accesses some data directly. In other words, every time you read the value of a property or change it, you might be accessing a field (even a private one) or calling a method. For example, here is the definition of a property for a date object:

```
property Month: Integer
  read FMonth write SetMonth;
```

To access the value of the `Month` property, this code has to read the value of the private field `FMonth`, while to change the value it calls the method `SetMonth`. Different combinations are possible (for example, we could also use a method to read the value or directly change a field in the `write` directive), but the use of a method to change the value of a property is very common. Here are some alternatives:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write Fmonth;
```

note When you write code that accesses a property, it is important to realize that a method might be called. The issue is that some of these methods take some time to execute; they can also produce a number of side effects, often including a (slow) repainting of the component on the screen. Although side effects of properties are seldom documented, you should be aware that they exist, particularly when you are trying to optimize your code.

The `write` directive of a property can also be omitted, making it a *read-only* property. Technically you can also omit the `read` directive and define a *write-only* property, but that doesn't make much sense. Another distinction is between *design-time* properties and *run-time only* properties. Design-time properties are declared in a `published` section of the class declaration. Anything that is declared in the `public` section is not available at design time—it is run-time only. All the read-only properties must be defined in the `public` section (or in the `protected` or `private` sections) because published properties must be *read-write*.

To see the value of a `published` property at design time or to change it, you can use the Object Inspector. This is the tool that Delphi's visual programming environment provides to give access to properties. At run time, you can access any `public` or `published` property by reading or writing its value.

note Remember that the Object Inspector lists only the design-time properties of a component, omitting the run-time only properties. Also, in Delphi 5 some properties can be hidden, if their category has been filtered out. For a complete list of the properties of a component, refer to the Delphi help files, not to the Object Inspector.

To summarize, along with the properties listed in the Object Inspector (design-time), there are other properties (run-time only), some of which can only be read (read-only). Note that you can usually assign a value to a property or read it, and you can even use properties in expressions, but you cannot always pass a property as a parameter to a procedure or method. This is because a property is not a memory location, so it cannot be used as a `var` parameter; it cannot be passed by reference.

Not all of the VCL classes have properties. Properties are present in components and in other subclasses of the `TPersistent` class, because properties usually can be streamed and saved to a file. A DFM file, in fact, is nothing but a collection of published properties of the components on the form. Delphi has extensive support for saving this kind of information, an advanced topic discussed in *Delphi Developers' Handbook* (Sybex, 1998)[124].

Adding Properties to Forms

Properties are a very sound OOP mechanism, a very well thought out application of the idea of encapsulation. Essentially, you have a name that hides the implementation of how to access the information of a class (either accessing the data directly or calling a method). In fact, using properties you end up with an interface that is unlikely to change. At the same time, if you only want to allow users access to some fields of your class, you can easily wrap those fields into properties instead of making them public. You have no further code to write (coding simple `Get` and `Set` methods is terribly boring), and you are still able to change the implementation of your class. Even if you replace the direct data access with method-based access, you won't have to change the source code that uses these properties at all. You'll only need to recompile it. Think of this as the concept of encapsulation raised to the maximum power!

Also keep in mind that using properties is very simple thanks to Code Completion. For this reason, every time you want to make some information of a form available to other forms, you should really use a property. Simply write in the form class declaration the property name and type:

```
property Clicks: Integer;
```

---

124 That book is now hard to find, given it was published over 25 years ago. The content of that book is still applicable today and th book is focused on Delphi RTL internals and writing components.

Then press Ctrl+Shift+C to activate Code Completion. You'll see the following effect:

```
type
  TForm1 = class(TForm)
  private
    FClicks: Integer;
    procedure SetClicks(const Value: Integer);
  public
    property Clicks: Integer
      read FClicks write SetClicks;
  end;

implementation

procedure TForm1.SetClicks(const Value: Integer);
begin
  FClicks := Value;
end;
```

This saves you a lot of typing, compared to the previous versions of Delphi, and it should make the use of properties the standard technique for accessing form data.

In my opinion, properties should also be used in the form classes to encapsulate the access to the components of a form. For example, if you have a main form with a status bar used to display some information (and with the `SimplePanel` property set to `True`), and you want to modify the text from a secondary form, you might be tempted to write:

```
Form1.StatusBar1.SimpleText := 'new text';
```

This is a standard practice in Delphi, but it's not a good one, because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (replacing `StatusBar` with another control or activating multiple panels), you'll have to fix the code in many places.

The alternative is to use a method or, even better, a property, to hide the specific control. Simply type

```
property StatusText: string
  read GetText write SetText;
```

and press the Ctrl+Shift+C combination again, to let Delphi add the definition of both methods for reading and writing the property:

```
function TForm1.GetText: string;
begin
  Result := StatusBar1.SimpleText;
```

```
end;

procedure TForm1.SetText(const Value: string);
begin
  StatusBar1.SimpleText := Value;
end;
```

In the other forms of the program, you can simply refer to the `StatusText` property of the form, and if the user interface changes, only the `Set` and `Get` methods of the property are affected.

Adding Properties to the TDate Class

In the previous chapter we developed the `TDate` class. Now we can extend it by using properties. This new example, DateProp, is basically an extension of the ViewD2 example from Chapter 2. Here is the new declaration of the class. It has some new methods (used to set and get the values of the properties) and four properties:

```
type
  TDate = class
  private
    fDate: TDateTime;
    function GetYear: Integer;
    function GetDay: Integer;
    function GetMonth: Integer;
    procedure SetDay (const Value: Integer);
    procedure SetMonth (const Value: Integer);
    procedure SetYear (const Value: Integer);
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string; virtual;
    property Day: Integer read GetDay write SetDay;
    property Month: Integer read GetMonth write SetMonth;
    property Year: Integer read GetYear write SetYear;
    property Text: string read GetText;
  end;
```

The `Year`, `Day`, and `Month` properties read and write their values using corresponding methods. Here are the two related to the `Month` property:

```
function TDate.GetMonth: Integer;
var
```

```
  y, m, d: Word;
begin
  DecodeDate (fDate, y, m, d);
  Result := m;
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRange.Create ('Invalid month');
  SetValue (Year, Value, Day);
end;
```

The call to `SetValue` performs the actual encoding of the date, raising an exception in case of an error. I've defined a custom exception class, which is raised every time a value is out of range:

```
type
  EDateOutOfRange = class (Exception);
```

The fourth property, `Text`, maps only to a read method. This function is declared as virtual, because it is replaced by the `TNewDate` subclass. There is no reason the `Get` or `Set` method of a property should not use late binding.

note What is important to acknowledge in this example is that the properties do not map directly to data. They are simply computed.

Having updated the class with the new properties, we can now update the example to use properties when appropriate. For example, we can use the `Text` property directly, and we can use some edit boxes to let the user read or write the values of the three main properties (as you can see in Figure 3.5). This happens when the Read button is pressed:

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```

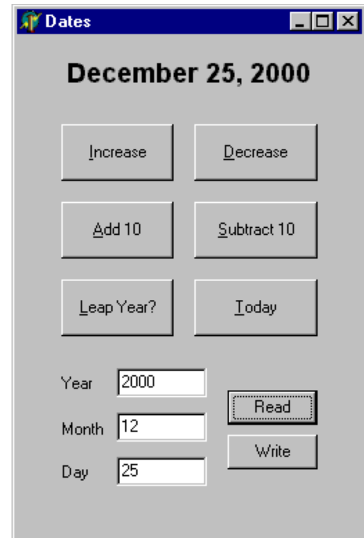


Figure 3.5:
The updated form of the DateProp example at run time. Image from the original book.

The Write button does the reverse operation. You can write the code in either of the two following ways:

```
// direct use of properties
TheDay.Year := StrToInt (EditYear.Text);
TheDay.Month := StrToInt (EditMonth.Text);
TheDay.Day := StrToInt (EditDay.Text);

// update all values at once
TheDay.SetValue (StrToInt (EditMonth.Text),
  StrToInt (EditDay.Text),
  StrToInt (EditYear.Text));
```

The difference between the two approaches relates to what happens when the input doesn't correspond to a valid date. When we set each value separately, the program might change the year and then raise an exception and skip executing the rest of the code, so that the date is only partially modified. When we set all the values at once, either they are correct and are all set, or one is invalid and the date object retains the original value.

note The `SetValue` method of this class and the three properties have the same relationship as the `SetBounds` method of the `TControl` classes has with the `Left`, `Top`, `Width`, and `Height` properties. Actually, in some special circumstances the same problem described above arises with these positional properties of controls.

Events in Delphi

When a user does something with a component, such as clicking it, the component generates an event. Other events are generated by the system, in response to a method call or a change to one of that component's properties (or even a different component's). For example, if you set the focus on a component, the component currently having the focus loses it, triggering the corresponding event.

Technically, most Delphi events are triggered when a corresponding Windows message is received, although the events do not match the messages on a one-to-one basis. Delphi events tend to be higher-level than Windows messages, and Delphi provides a number of extra intercomponent messages.

From a theoretical point of view, an event is the result of a message sent to a window, and this window (or the corresponding component) can respond to the message. Following this approach, to handle the click event of a button, we would need to subclass the `TButton` class and add the new event handler.

In practice, creating a new class is too complex to be a reasonable solution. In Delphi, the event handler of a component usually is a method of the form that holds the component, not of the component itself. In other words, the component relies on its owner, the form, to handle its events. This technique is called *delegation,* and it is fundamental to the Delphi component-based model.

Events Are Properties

Another important concept is that events are properties. This means that to handle an event of a component, you assign a method to the corresponding event property, as we did in the CountOb2 example earlier in this chapter. When you double-click an event in the Object Inspector, a new method is added to the owner form and assigned to the proper event property of the component.

This is why it is possible for several events to share the same event handler or change an event handler at run time. To use this feature, you don't need much knowledge of the language. In fact, when you select an event in the Object Inspector, you can press the arrow button on the right of the event name to see a drop-down list of "compatible" methods—a list of methods having the same method pointer type. Using the Object Inspector, it is easy to select the same method for the same event of different components or for different, compatible events of the same component.

Adding an Event to the TDate Class

As we've added some properties to the `TDate` class, we can add one event. The event is going to be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the value of the date has changed. To define an event, we simply define a property corresponding to it, and we add some data to store the actual method pointer the event refers to. These are the new definitions added to the class:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FonChange write FOnChange;
    ...
  end;
```

The property definition is actually very simple. A user of this class can assign a new value to the property and, hence, to the `FOnChange` private field. The class doesn't assign a value to this `FOnChange` field. It is the user of the component who does the assignment. The `TDate` class simply calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned. The `DoChange` method (declared as a `dynamic` method as it is traditional with event firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

The `DoChange` method in turn is called every time one of the values changes, as in the following method:

```
procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
  // fire the event
  DoChange;
```

Now if we look at the program that uses this class, we can simplify its code considerably. First, we add a new custom method to the form class:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

The code of this method simply updates the label with the current value of the `Text` property of the `TDate` object:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;
```

This event handler is then installed in the `FormCreate` method:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (7, 4, 1997);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

Well, this seems like a lot of work. Was I lying when I told you that the event handler would save us some coding? No. Now, after we've added some code, we can completely forget about updating the label when we change some of the data of the object. Here, as an example, is the handler of the `OnClick` event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

The same simplified code is present in many other event handlers. Once we have installed the event handler, we don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that we had to write some code at the beginning because this is not a component installed in Delphi but simply a class. With a component, you simply select the event handler in the Object Inspector and write a single line of code to update the label. That's all. How difficult is it to write a new component in Delphi? It's actually so simple I'm going to show you how to do it in the next section.

note This is meant to be just a short introduction to the role of properties and events and to writing components. A basic understanding of these features is important for every Delphi programmer. If your aim is to write complex new components, you'll find a lot more information on all of these topics in Chapter 13.

Creating a TDate Component

The next step, actually a very simple one, is to turn our `TDate` class into a component. First, we have to inherit our class from the `TComponent` class, instead of the default `TObject` class. Here is the code:

```
type
  TDate = class (TComponent)
    ...
  public
    constructor Create (AOwner: TComponent); overload; override;
    constructor Create (y, m, d: Integer); reintroduce; overload;
```

As you can see, the second step was to add a new constructor to the class, overriding the default constructor for components to provide a suitable initial value. Because there is an overloaded version, we also need to use the `reintroduce` directive for it, to avoid a warning message from the compiler. The code of the new constructor simply sets the date to today's date, after calling the base class constructor:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
  fDate := Date;
```

Having done this, we need to add to the unit that defines our class (the file `DATES.PAS` in the `DATECOMP` directory) a `Register` procedure. (Make sure this identifier start with an uppercase *R*, otherwise it won't be recognized.) This is required in order to add the component to Delphi's Components Palette. Simply declare the procedure, which requires no parameters, in the `interface` portion of the unit, and then write this code in the `implementation` section:

```
procedure Register;
begin
  RegisterComponents ('Md', [TDate]);
end;
```

This code adds the new component to the *Md* page of the Components Palette[125], creating the page if necessary. By the way, this is the same page I'll use for all the components built in the book.

125 Nowadays, the Palette pane hosts the components. The behavior described here remains the same.

The last step is to install the component. For this simple example we won't create a new package. Instead, we can install the component in the default Borland User's Components package (a file named `DCLUSR50.DPK` and stored in the `LIB` directory of Delphi). We'll see how to build new packages in Chapter 13.

To make the component available, select the Component ➢ Install Component menu item, choose the *Into existing package* page (this should be the default), select the `DCLUSR50.DPK` package filename[126] (again the default if you've never installed components), and enter the unit filename of the component, `DATES.PAS`. Now simply click OK and Delphi will update the package, compile it, and ask you to install it in Delphi (if you haven't already done so).

If you now move to the Components Palette, it should have a new *Md* page with the new component. This will be shown using the default icon for Delphi components. At this point you can place the component on the form of a new application and start manipulating its properties in the Object Inspector, as you can see in Figure 3.6. You can also handle the `OnChange` event in a much easier way than in the last example.

Besides trying to build your own sample application using this component (something I really suggest you do), you can now open the DateComp example, which is an updated version of the component we've built step-by-step over the last few sections of this chapter. This is basically a simplified version of the DateEvt example, because now the event handler is directly available in the Object Inspector.

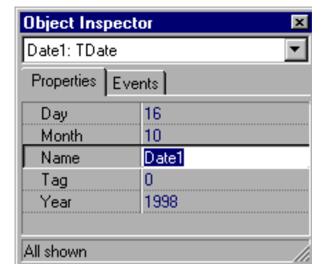


Figure 3.6: The properties of our new TDate component in the Object Inspector. Image from the original book.

126 The package is still called "dclusr.dpk" today. It's description, oddly enough, is "CodeGear User Components". Notice that in the first page of the dialog box you need to select the components source code file, while in the second you can pick the package you want to install it into.

note If you open the DateComp example before installing the new component, Delphi won't recognize the component as it opens the form and will give you an error message. You won't be able to compile the program or make it work until you install the new component.

Using Interfaces

Contrary to what happens in C++, the Delphi inheritance model doesn't support multiple inheritance. This means that each class can have only a single base class. The usefulness of multiple inheritance is a topic of heated debate. The absence of this construct in Delphi can be considered both a disadvantage (because you lose some of the power of C++) and an advantage (because you get a simpler language and fewer problems). My point is that Delphi's interfaces provide the flexibility and power of declaring support for multiple interfaces implemented on a class, while avoiding the problems of inheriting multiple implementations. Rather than get bogged down in this debate, I'll simply assume that it is useful to treat a single object from multiple "perspectives," to consider it a generic object of different base classes. But before I build an example following this principle, we have to introduce the role of interfaces in Object Pascal.

note The techniques covered in this section are used also to implement COM objects, and I'll cover them in more detail in Chapter 15. For the moment, let's consider them simply as language elements.

Declaring an Interface

Besides declaring abstract classes (classes with abstract methods), in Delphi you can also write a *purely abstract class;* that is, a sort of class with only virtual abstract methods. This is accomplished using a specific keyword, `interface`. For this reason we refer to these classes as *interfaces*. Technically, in fact, an interface is not a class, although it may resemble one. Interfaces are not classes, because they are considered a totally separate element, with its own common base interface, `IUnknown`[127], which has the same role as `TObject` for classes.

---

127 More recently the *IUnknown* interface has been renamed *IInterface*, to underline the fact you can use interface in Delphi even outside of the COM realm. The actual behavior of *IInterface*, though, is still identical to the previous one of *IUnknown*.

Borland introduced interfaces in Delphi 3 along with the support COM programming. If the interface language syntax may have been created to support COM, interfaces do not require COM. You can use interfaces to implement abstraction layers within your applications, without building COM server objects. For example, the Delphi IDE uses interfaces extensively in its internal architecture. In general, interfaces also have some distinctive advantages that can be useful for different types of programming:

- A class can inherit from a single base class, but it can also implement multiple interfaces. The drawback is that when a class implements an interface, it must provide the implementation for each of the methods of the interface.
- Interface type objects are reference-counted and automatically destroyed when there are no more references to the object. This mechanism is similar to how Delphi manages long strings and makes memory management almost automatic.
- The VCL already provides a few base classes to implement the basic behavior required by the `IUnknown` interface. The simplest one is the `TInterfacedObject` class.

note From a more general point of view, interfaces support a slightly different object-oriented programming model than classes. Objects implementing interfaces are subject to polymorphism for each of the interfaces they support. Indeed, the interface-based model is powerful. But having said that, I'm not interested in trying to assess which approach is better in each case. Certainly, interfaces favor encapsulation and provide a more loose connection between classes than inheritance.

Here is the syntax of the declaration of an interface (which, by convention, starts with the letter *I*):

```
type
  ICanFly = interface
    ['{10000000-0000-0000-0000-000000000000}']
    function Fly: string;
  end;
```

note To function properly, each interface requires a numeric ID, like the one above. In theory these should be unique GUIDs, generated in the Delphi editor by pressing Ctrl+Shift+G, but if you don't plan to export these objects, any number will do (more on GUIDs in Chapter 15). These GUIDs are required even if you don't plan exporting these classes, because they are used by the compiler to type-check interface types instead of the plain interface and class names.[128]

128 The GUID in the code snippet above is not a real one. I'd recommend you replace it in the code with an actual GUID, generated by pressing Ctrl+Shift+G, even if the code works anyway.

Once you've declared an interface, you can define a class to implement it, as in:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string; virtual;
  end;
```

As mentioned, this class can derive from `TInterfacedObject` to inherit the implementation of the `IUnknown` methods. Although it is not compulsory to implement interface methods with virtual methods, this is a good approach to use if you want to be able to modify these methods in further sub-classes. An alternative technique is to re-declare the interface type in a derived class and rebind the interface methods to static methods declared in that class.

Now that we have defined an implementation of the interface, we can write as usual

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  Airplane1.Fly;
  Airplane1.Free;
end;
```

But we can also use an interface-type variable:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

As soon as you assign an object to an interface variable, Delphi automatically checks to see whether the object implements that interface, using a special version of the `as` operator. You can explicitly express this operation as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

Whether we use the direct assignment or the `as` statement, Delphi does one extra thing: it calls the `_AddRef` method of the object, increasing its reference count. At the same time, as soon as the `Flyer1` variable goes out of scope, Delphi calls the `_Release` method, which decreases the reference count, checks whether the reference count is zero, and if necessary, destroys the object. For this reason in the listing above, there is no code to free the object we've created[129].

129 There are many other techniques you can use with interfaces these days, including weak interfaces and unsafe ones. This is an advanced concept I cannot really cover in a footnote.

In other words, in Delphi objects referenced by interface variables are reference-counted, and they are automatically de-allocated when no interface variable refers to them any more.

note When using interface-based objects, you should generally access them only with object variables or only with interface variables. Mixing the two approaches breaks the reference counting scheme provided by Delphi and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables.

Interface Properties, Delegation, Redefinitions

To demonstrate a few technical elements related to interfaces, I've written the IntfDemo example. This example is based on two different interfaces, `IWalker` and `IJumper`, defined as follows:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;
```

Notice that the first interface defines also a property. An interface property is just a name mapped to a `read` and a `write` method. You cannot map an interface property to a field, simply because an interface cannot have a field.

Here comes a sample implementation of the `IWalker` interface. Notice that you don't have to define the property, only its access methods:

```
TRunner = class (TInterfacedObject, IWalker)
private
  Pos: Integer;
public
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

The code is trivial, so I'm going to skip it (you can find it in the IntfDemo example). In a similar way, I've defined a class implementing the `IJumper` interface:

```
TJumperImpl = class (TInterfacedObject, IJumper)
private
  Pos: Integer;
public
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

Although this class isn't different from the other one, I'm going to use it in a different way. In the following class, `TMyJumper`, I don't want to repeat the implementation of the `IJumper` interface with similar methods. Instead, I want to delegate the implementation of that interface to a class already implementing it. This cannot be done through inheritance (we cannot have two base classes); instead, you can use specific features of the language interface delegation:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
  fJumpImpl: IJumper;
public
  constructor Create;
  property Jumper: IJumper
    read fJumpImpl implements IJumper;
end;
```

This declaration indicates that the `IJumper` interface is implemented for the `TMyJumper` class by the `fJumpImpl` field. This field, of course, must actually implement all the methods of the interface. To make this work, you need to create a proper object for the field when a `TMyJumper` object is created:

```
constructor TMyJumper.Create;
begin
  fJumpImpl := TJumperImpl.Create;
end;
```

This example is simple, but in general, things get more complex as you start to modify some of the methods or add other methods that still operate on the data of the internal `fJumpImpl` object. This final step is demonstrated, along with other features, by the `TAthlete` class, which implements both the `IWalker` and `IJumper` interfaces:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
  fJumpImpl: TJumperImpl;
public
  constructor Create;
  function Run: string; virtual;
  function Walk1: string; virtual;
  function IWalker.Walk = Walk1;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Jumper: TJumperImpl
    read fJumpImpl implements IJumper;
end;
```

One of the interfaces is implemented directly, whereas the other is delegated to the internal `fJumpImpl` object. Notice also that by implementing two interfaces, which have a method in common, we end up with a name clash. The solution is to rename one of the methods, with the statement

```
function IWalker.Walk = Walk1;
```

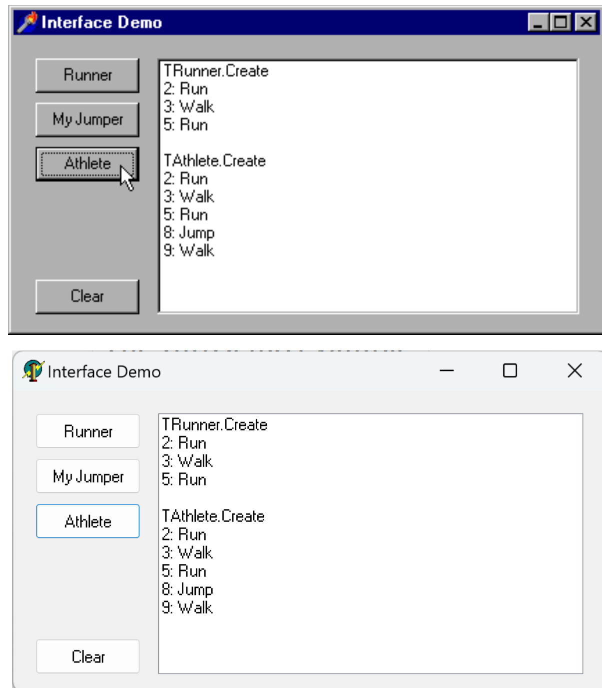
This declaration indicates that the class implements the `Walk` method of the `IWalker` interface with a method called `Walk1` (instead of with a method having the same name). Finally, in the implementation of all of the methods of this class, we need to refer to the Position property of the `fJumpImpl` internal object. By declaring a new implementation for the `Position` property, we'll end up with two positions for a single athlete, a rather odd situation. Here are a couple of examples:

```
function TAthlete.GetPos: Integer;
begin
  Result := fJumpImpl.Position;
end;

function TAthlete.Run:string;
begin
  fJumpImpl.Position := fJumpImpl.Position + 2;
  Result := IntToStr (fJumpImpl.Position) + ': Run';
end;
```

You can further experiment with the IntfDemo example, which has a simple form with buttons to create and call methods of the various objects. Nothing fancy,

though, as you can see in Figure 3.7. Simply keep in mind that each call returns the position after the requested movement and a description of the movement itself.

Figure 3.7:
The IntfDemo example



An Example of Multiple Inheritance

After this example, let me move to a more complex series of interfaces. Suppose you have a hierarchy of classes related to animals. You can base the hierarchy on the standard taxonomic classifications (with categories such as mammals, birds, insects, and so on), or you can categorize them by capability (flying animals, quadrupeds or bipeds, meat eaters, and so on).

There is no easy way to express such a complex structure with single inheritance. You can use multiple inheritance if the language you are using supports this feature, or you can use interfaces. This is what I've done in my example, which represents a rather common study case for multiple inheritance. This program, named MultInh, has both a hierarchy of classes (representing the standard zoological classifications) and a hierarchy of interfaces (expressing the capabilities).

Both the hierarchy of classes and the hierarchy of interfaces actually use single inheritance. It is only when you look at how classes implement the various interfaces that the two hierarchies actually merge, as represented in Figure 3.8.

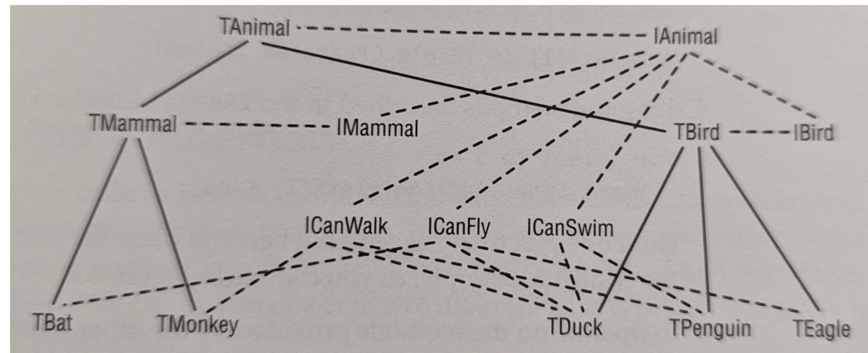


Figure 3.8: The complex relationships among the classes and interfaces of the MultInh example. Image from the original book, captured with a picture, as the original version got lost.

The declarations of these interfaces and their methods are quite long, so I've decided to skip them. Each of them has a specific GUID and defines one or more functions returning strings. The actual classes implement one or more of these interfaces, as depicted in Figure 3.8. Here are a couple of declarations:

```
type
  TBird = class (TAnimal, IBird)
    function LayEggs: string; virtual;
  end;

  TEagle = class (TBird, ICanFly)
    function Kind: string; override;
    function Fly: string; virtual;
  end;

  TPenguin = class (TBird, ICanWalk, ICanSwim)
    function Kind: string; override;
    function Walk: string; virtual;
    function Swim: string; virtual;
```

Now that we have designed this infrastructure, how can we use it? How do we create objects of these classes, and how can we use polymorphism in classes that implement multiple interfaces?

Interface Polymorphism

To use polymorphism with interfaces, I've declared and filled an array inside the form of the program:

```
private
  AnimIntf: array [1..5] of IAnimal;
```

The program extracts the `IAnimal` interface from newly created objects to initialize this array. This is done automatically by Delphi when you write

```
AnimIntf[1] := TEagle.Create;
```

which corresponds to writing

```
AnimIntf[1] := TEagle.Create as IAnimal;
```

Calling the methods described in the `IAnimal` interface is straightforward:

```
for I := 1 to 5 do
  Memo1.Lines.Add (AnimIntf[I].Kind);
```

This code is actually executed when you press the first button of the main form of the MultInh example, as you can see in Figure 3.9.

To operate on the methods provided by the other interface, we must first check to see whether any given object supports it. Because there is no `is` operator for interfaces[130], we can accomplish it by calling the `QueryInterface` method:

```
var
  Fly1: ICanFly;
begin
  AnimIntf[i].QueryInterface (ICanFly, Fly1);
  if Assigned (Fly1) then
    Memo1.Lines.Add (Fly1.Fly);
```

---

130 The *is* operators for interfaces has later been added to the language.

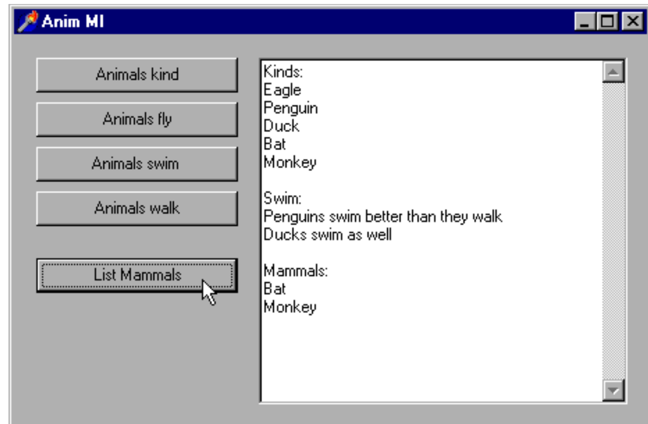


Figure 3.9: The simple user interface of the MultInh example. Image from the original book.

`QueryInterface` requires as parameters a variable for the return value and the type of interface to check for. Because it returns also an error code, we can also check this, as I've done in another case:

```
var
  Swim1: ICanSwim;
begin
  if AnimIntf[i].QueryInterface (
    ICanSwim, Swim1) <> E_NoInterface then
  Memo1.Lines.Add (Swim1.Swim);
```

We can also use the `as` statement using a `try-except` block, but this is not a solution I really like. (It is in the source code of the program for you to check, anyway.)

Is This Multiple Inheritance?

The last two code fragments combined indicate that we can use an object and cast it to the multiple interfaces it supports. In other words, we can consider a duck to be a swimming animal or a flying animal and call methods of both interfaces for a single object. We can cast an object to two different base types, so this really is like multiple inheritance.

What we don't get is the inheritance of the actual implementation of the methods; there is no code in the `ICanFly` interface, and if there were any code shared by all the "flying" objects, it would need to be reimplemented in each class that supports this interface. However, we already know that it is possible to define a single imple-

mentation class and delegate to it the implementation of an interface in many other classes, as I did in the previous example.

As I mentioned earlier, Borland added interfaces to Delphi to support Microsoft's COM, but they can really be used as an extra language feature. The biggest drawback is that interfaces must have an ID even for internal objects, because the type checking of interfaces depends on this number. The other minor problem is that there isn't an `is` operator to check whether an object supports a given interface, but we've seen it is very simple to mimic this behavior by calling `QueryInterface` with a single method call.

Summing up, does it really make sense to use interface types and variables in a program that doesn't need to support COM? If the program is designed around a complex hierarchy that might benefit from multiple inheritance, then the answer is yes. Considering the extra complexity of this design, however, you might disagree.

What's Next?

By reading this chapter, you might have had the impression that I've covered a number of unrelated topics. This was only partially the case. Class references, method pointers, properties, events, the `published` keyword, and exceptions are all language features upon which Delphi's Visual Component Library is built. Other topics, such as as class method or interfaces, are important additions to the language every Delphi developer should at least be familiar with.

Having covered the basics of OOP in the last chapter and all these language extensions in the current one, we can now focus on the structure of the VCL in the next chapter.

There is actually one extra step we've done in this chapter: we've built a first simple component, and we've installed it in the Delphi environment. This already demonstrates the fact that a component is actually an Object Pascal class that inherits from a specific base class, `TComponent`. *Delphi components are classes:* this apparently simple statement describes the nature of the Delphi programming model, underlying its differences with tools as Visual C++ or Visual Basic. The only other language coming close to Object Pascal in terms of components development is Java.

Chapter 4: VCL Programming Techniques

To simplify the work of programming, Delphi provides many powerful, ready-to-use functions and classes. It includes, for example, a number of standard routines. (The Help files no longer have a complete list of these routines, but you can find such a list at my `www.marcocantu.com` Web site[131].) Even larger and more important is Delphi's set of classes. Some of them are component classes, which show up in the Component Palette, while others are more general-purpose. This chapter focuses on the structure of the Delphi class library—known as the *Visual Component Library (VCL)*, although it includes more than components—and gives an overview of some general-purpose classes.

131 It's not there any more, and I doubt I'll be able to create an updated version

If you simply want to put the built-in components to work and don't care about the ins and outs of the VCL, you may want to skip this chapter for now and move on to Part II, which focuses on the use of components and other classes related to Windows, or to Part III, which covers database programming (including the standard data-aware components). Remember to come back to this chapter when you are ready to leverage your Delphi programming knowledge.

The TObject Class

At the heart of Delphi is a hierarchy of classes. Every class in the system is a subclass of the `TObject` class, so the whole hierarchy has a single root. This allows you to use the `TObject` data type as a replacement for the data type of any class type in the system.

For example, event handlers usually have a `Sender` parameter of type `TObject`. This simply means that the `Sender` object can be of any class, since every class is ultimately derived from `TObject`. The typical drawback of such an approach is that to work on the object, you need to know its data type. In fact, when you have a variable or a parameter of the `TObject` type you can apply to it only the methods and properties defined by `TObject`. If this variable or parameter happens to refer to an object of the `TButton` type, for example, you cannot directly access its `Caption` property. The solution to this problem lies in the fact that each object "knows" its actual class, and you can access this information using the `ClassType` and `ClassName` methods. For example, `ClassName` returns a string with the name of the class. Because it is a class method, you can apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

There are occasions when you need to use the name of a class, but it can also be useful to retrieve a class reference to the class itself or to its base class. The class reference, in fact, allows you to operate on the class at run time (as we've seen in the last chapter), while the class name is just a string. We can get these class references with the `ClassType` and `ClassParent` methods. Once you have a class reference, you can use it as if it were an object—for example, to call the `ClassName` method.

Another method that might be useful is `InstanceSize`, which returns the run-time size of an object. (Although you might think that the `SizeOf` global function pro-

vides this information, that function actually returns the size of an object reference—a pointer, which is invariably four bytes—instead of the size of the object itself.)

There are other methods you can apply to any object (and also to any class or class references). Here is a partial list[132]:

`ClassName`	Returns a string with the name of the class.
`ClassNameIs`	Checks the class name.
`ClassParent`	Returns a class reference to the parent class.
`ClassInfo`	Returns a pointer to the internal Run Time Type Information (RTTI) of the class, discussed in *Delphi Developer's Handbook*.
`ClassType`	Returns a reference to the object's class (this cannot be applied directly to a class, only to an object).
`InheritsFrom`	Tests whether the class inherits (directly or indirectly) from a given base class (similar to the `is` operator).
`InstanceSize`	Returns the size of the object's data.

These methods of `TObject` are available for objects of every class, since `TObject` is the common ancestor class of every class. Here is how we can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
  Memo1.Lines.Add ('Class Name: ' +
    Sender.ClassName);

  if Sender.ClassParent <> nil then
    Memo1.Lines.Add ('Parent Class: ' +
      Sender.ClassParent.ClassName);

  Memo1.Lines.Add ('Instance Size: ' +
    IntToStr (Sender.InstanceSize));
```

The code checks to see whether the `ClassParent` is nil in case you are actually using an instance of the `TObject` type, which has no base type. You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

132 There have been several notable additions to the *TObject* class methods over the years, including *Equals*, *GetHashCode*, *QualifiedClassName*, *ToString*, *UnitName*. Some of these are virtual methods you can override in your derived classes. See docwiki.embarcadero.com/Libraries/en/System.TObject for more details.

```
if Sender.ClassType = TButton then ...
```

You can also check if the `Sender` parameter corresponds to a given object, with this test:

```
if Sender = Button1 then...
```

All these code fragments are part of the IfSender example.

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class or one of its subclasses. This lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

All these techniques are demonstrated by the IfSender example, which has a single event handler, called `ShowSender`, connected with the `OnClick` event of several controls: three buttons, a check box, and an edit box. One of the buttons is actually a Bitmap button, an object of a `TButton` subclass. You can see an example of the output of this program at run time in Figure 4.1.

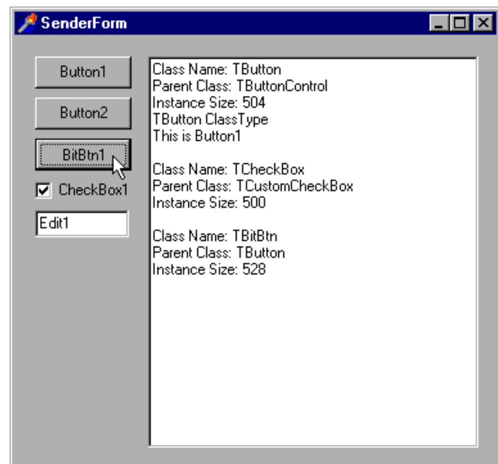


Figure 4.1: The output of the IfSender example. Image from the original book.

Showing Class Information

The IfSender example can be extended to show a complete list of base classes. Once you have a class reference, in fact, you can add all of its base classes to the ListParent list box with the following code:

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

You'll notice that we use a class reference at the heart of the while loop, which tests for the absence of a parent class (so that the current class is TObject). Alternatively, we could have written the while statement in either of the following ways:

```
while not MyClass.ClassNameIs ('TObject') do...
while MyClass <> TObject do...
```

The code in the with statement referring to the ListParent list box is part of the ClassInfo example, which displays the list of parent classes and some other information about a few components of the VCL, basically those on the Standard page of the Component Palette. These components are manually added to a dynamic array holding classes and declared as:

```
private
  ClassArray: array of TClass;
```

When the program starts the array is used to show all the class names in a list box. Selecting an item of the list box triggers the visualization of its base classes, as you can see in the output of the program, in Figure 4.2.

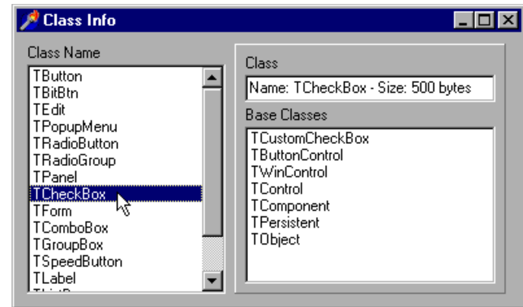


Figure 4.2: The output of the ClassInfo example. Image from the original book.

note As a further extension to this example, we might show all the base classes of the various components in a hierarchy. To do that, I've created the VclHierarchy Wizard, which you can find on my Web site.[133]

The VCL Hierarchy

The VCL defines a number of subclasses of `TObject`. Many of these classes are actually subclasses of other subclasses, forming a very complex hierarchy. Unless you are interested in developing new components, you'll usually use only the *terminal* classes of this hierarchy—the leaf nodes of the hierarchy tree. This is not really a precise description, as some of the leaf nodes can be further extended by deriving new components, and some of the classes in higher-level nodes can be instantiated directly.

note Delphi's documentation includes a large poster of the VCL class hierarchy. Although its size makes it a little cumbersome, this can be a precious reference for understanding the VCL class hierarchy. Again, you can also find a VCL class hierarchy on my Web site.[134]

We can divide the VCL hierarchy into three main areas: components, generic objects, and exceptions. Components can be modified visually in the Delphi IDE, typically using the Form Designer, while the other types of classes are referenced

133 While I have similar code available and in active use, this specific code is no longer on my site.

134 The poster was a great tool in the early days of the product. As the library kept growing and the printed documentation started reducing, it was removed. With the switch to digital distribution, there was later no point in considering it.

only in the source code. As a detailed description would take too much space, this chapter includes some general notes, mainly on components but also on some other important classes of the VCL.

Components

Components are the central elements of Delphi applications. When you write a program, you basically choose a number of components and define their interactions. That's all there is to Delphi visual programming.

There are different kinds of components in Delphi. Most components are included in the Component Palette, but some of them (including `TForm` and `TApplication`) are not. Technically, components are sub-classes of the `TComponent` class. As such, they can be streamed in a `DFM` file (since they inherit from the `TPersistent` class, which provides the information needed for streaming) and they may have published properties and events you can manipulate visually. We saw a simple example (DateComp) of building a component in the last chapter.

The part of the VCL hierarchy related to components is generally divided into three areas, as you can see in Figure 4.3[135]. These groups indicate components with a similar internal structure:

- *Controls* or *visual components* are all the classes that descend from `TControl`. Controls have a position and a size on the screen and show up in the form at design time in the same position they'll have at run time. Controls have two different specifications, window-based or graphical:
    - *Window-based controls* (also called *windowed controls*) are visual components based on an operating system window. From a technical point of view, this means that these controls have a window handle and descend from `TWinControl`. From a user perspective, windowed controls can receive the input focus and some of them can contain other controls. This is the biggest group of components in the Delphi VCL. We can further divide windowed controls in two groups: wrappers of Windows controls and custom controls.
    - *Graphical controls* (also called *nonwindowed controls*) are visual components that are not based on a window. Therefore, they have no handle, cannot receive the focus, and cannot contain

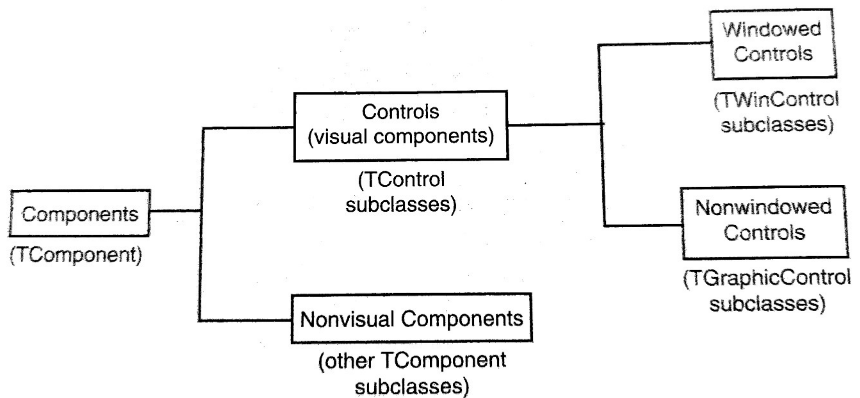
135 This division in terms of component groups and their role is still 100% applicable today.

other controls. These controls inherit from `TGraphicControl` and are painted by their parent form, which sends them mouse-related and other events. Examples of nonwindowed controls are the Label and the SpeedButton components. There are just a few controls in this group, but they are critical to minimizing the use of system resources, particularly for components used often and in number, such as labels or toolbar buttons.

- *Nonvisual components* are all the components that are not controls—all the classes that descend from `TComponent` but not from `TControl`. At design time, a nonvisual component appears on the form as an icon (optionally with a caption below it). At run time, some of these components may be visible at times (for example, the standard dialog boxes), and others are always invisible (for example, the database table component). In other words, nonvisual components are not visible themselves at run time, although they may manage something that is visual, such as a dialog box.

note You can simply move the mouse cursor over a control or component in the Form Designer to see a hint with its name and its class type. You can also use an environment option, Show Component Captions, to see the name of a nonvisual component right under its icon.

Figure 4.3: A graphical representation of the groups of components. Image based on a picture of the original printed book.



Windows Components

You might have asked yourself where the idea of using components for Windows programming came from. The answer is simple: Windows itself has some components, usually called controls. A *control* is technically a predefined window that has a specific behavior and some styles and is capable of responding to specific messages. These controls were the first step in the direction of component development. The second step was probably Visual Basic controls, and the third step is Delphi components.

note Actually, Microsoft's third step is its ActiveX, the designated successor of VBX controls. In Delphi you can use both ActiveX and native components, but if you look at the technology, Delphi components are really ahead of the ActiveX controls. Delphi components use OOP to its full extent, while ActiveX controls do not fully implement the concept of inheritance. I'll focus on the details of using and writing ActiveX controls in Chapter 16.

Windows 3.1 had six kinds of predefined controls, which were generally used in dialog boxes. Still used in Win32, they are buttons (push buttons, check boxes, and radio buttons), static labels, edit fields, list boxes, combo boxes, and scroll bars. Win32 adds a number of new predefined components, such as the list view, the status bar, the spin button, the progress bar, the tab control, and many others. Win32 developers can use the standard common controls provided by the system, and Delphi developers have the further advantage of having corresponding easy-to-use components.

The standard system controls are the basic components of each Windows application, regardless of the programming language used to write it, and are very well known by every Windows user. Delphi literally wraps these Windows predefined controls in some of its basic components. A Delphi wrapper class, for example `TEdit`, simply surfaces the capabilities of the underlying Windows control, making it easier to use. However, Delphi adds nothing to the capabilities of this control. In Windows 95/98 an edit or a memo control has a physical limit of 32KB of text, and this limit is retained by the Delphi component.

Why hasn't Borland overcome this limit? Why can't we change the color of a button[136]? Simply because by replacing a Windows control with a custom version, we would lose the close connection with the operating system. Suppose Microsoft

136 This isn't true any more, as now the VCL library now offers support for styling. The concept of relying on platform controls still applies, but styles offer a higher degree of visual customization for any platform control.

improves some of the controls in the next version of Windows. If we use our own version of the component, the application we build won't have the new features.

By using controls that are based on the operating system capabilities, instead, our programs can easily migrate through different versions of the OS and retain all the features provided by the specific version.

Of course, if you need a control that does something really different from the existing ones, you'll need to write your own custom controls, something the VCL itself does in the classes inheriting from `TCustomControl`. For example, a Delphi grid isn't related to any Windows control. All the classes in that portion of the VCL tree aren't directly related to Windows standard controls or Win32 common controls.

Note that wrapping an existing Windows is an effective way of reusing code and also helps reduce the size of your compiled code. Implementing yet another button control from scratch requires custom code in your application, while a wrapper around the OS-supplied button control requires less code and makes use of system code shared by all Windows applications.

Objects

Although the VCL is basically a collection of components, there are other classes that do not fit in this category, because they do not descend from `TComponent`. All the noncomponent classes are often identified (by the Delphi Help files and documentation, among others sources) as *objects*, although this is not a precise definition. There are two main uses for these classes. Generally, noncomponent classes define the data type of component properties, such as the `Picture` property of an image component (which is a `TGraphic` object) or the `Items` property of a list box (which is a `TStrings` object). These classes generally inherit from `TPersistent`, so they are *streamable*, and they can have sub-properties and even events.

The second use of noncomponent classes is a direct use. In the Delphi code you write, you can allocate and manipulate objects of these classes. You might do this for a number of purposes, including to store a copy of the value of a property in memory and modify it without changing the original component, to store a list of values, to write complex algorithms, and so on. You'll see several examples in this book that show how to use non-component classes directly.

There are several groups of non-component classes in the VCL[137]:

137 The list has been significantly extended over the years, but the core classes listed here are still available and relevant today.

- *Graphic-related objects* include `TBitmap`, `TBrush`, `TCanvas`, `TFont`, `TGraphic`, `TGraphicsObject`, `TIcon`, `TMetafile`, `TPen`, and `TPicture`.
- *Stream/file-related objects* include `TBlobStream`, `TFileStream`, `THandleStream`, `TIniFile`, `TMemoryStream`, `TFiler`, `TReader`, and `TWriter`.
- *Lists and collections* include `TList`, `TStrings`, `TStringList`, `TCollection`, `TCollectionItem` and the new container classes introduced by Delphi 5. We will focus on these classes in a later section of this chapter.
- *COM-related classes:* This is an important area of Delphi programming. COM-related classes are covered in Chapter 15.
- *Exception classes:* These are inherited from the `Exception` class. We discussed exception handling in Chapter 3, so I won't repeat the details here.

Common VCL Properties

Although each component has its own set of properties, you may have already noticed that some properties are common to all of them. Table 4.1 lists some of the common properties along with very short descriptions[138].

Table 4.1: Some Properties Available in Most Components

Property	Available For	Description
`Action`	Some controls	Identifies the Action object connected to the control (see Chapter 5 for details).
`Align`	Some controls	Determines how the control is aligned in its parent control area.
`Anchors`	Most controls	Indicates the side of the form the component is connected with (see Chapter 7 for an example).
`AutoSize`	Some controls	Indicates whether the control can determine its own size depending on its content.
`BiDiMode`	All controls	Provides support for languages written right to left (stands

138 This is still a very good list of the most relevant common properties.

		for BiDirectional Mode).
`BorderWidth`	Windowed controls	The width of the border.
`BoundsRect`	All controls	Defines the bounding rectangle of the control (run-time only).
`Caption`	Most controls	The caption of the control.
`ComponentCount`	All components	The number of components owned by the current one (run-time only and read-only).
`ComponentIndex`	All components	The position of the component in its owner's list of components (run-time only).
`Components`	All components	An array of the components owned by the current one (run-time only and read-only).
`Constraints`	All controls	Determines the maximum and minimum size of a control (or a form) during resizing operations.
`ControlCount`	All controls	The number of child controls of the current one (run-time only and read-only).
`Controls`	All controls	An array of the child controls of the current one (run-time only and read-only).
`Color`	Most controls	Indicates the color of the surface or the background.
`Ctrl3D`[139]	Most components	Determines whether the control has a three-dimensional look.
`Cursor`	All controls	The cursor used when the mouse pointer is over the control.
`DockSite`	Most windowed controls	Indicates whether the windowed control is a docking site. There are other properties related to this, including `DockClientCount`, `DockClients`, `UseDockManager`, and `DockManager`. Docking is discussed in Chapters 7 and 8.

139 This property is now obsolete.

`DragCursor`	Most controls	The cursor used to indicate that the control accepts dragging.
`DragKind`	Most controls	Lets you choose between dragging and docking, if the drag mode is automatic.
`DragMode`	Most controls	Determines whether the drag-and-drop behavior (allowing either dragging or docking, as specified in the `DragKind` property) will be activated automatically.
`Enabled`	All controls and some nonvisual components	Determines whether the control is active or inactive (grayed).
`Font`	All controls	Determines the font of the text displayed inside the component.
`Handle`	All windowed controls	The handle of the system window used by the control (run-time only and read-only).
`Height`	All controls	The vertical size of the control.
`HelpContext`	All controls and the dialog components	A context number used to invoke the context-sensitive Help automatically.
`Hint`	All controls	The string used to display fly-by hints for the control.
`Left`	All controls	The horizontal coordinate of the upper-left corner of the component.
`Name`	All components	The unique name of an instance of the component, which can generally be used in the source code.
`Owner`	All components	Indicates the owner component (run-time only and read-only).
`Parent`	All controls	Indicates the parent control (run-time only).
`ParentColor`	Most controls	Determines if the component uses the same `Color` as the parent.
`ParentCtl3D`[140]	Most components	Determines whether the component uses the same `Ctrl3D` as the parent.

ParentFont	All controls	Determines whether the component uses the same Font as the parent.
ParentShowHint	All controls	Determines whether the component uses the same ShowHint as the parent.
PopupMenu	All controls	The pop-up menu used when the user right-clicks on the control.
ShowHint	All controls	Determines whether hints are enabled.
Showing	All controls	Determines whether the control is currently *showing* on the screen; that is, if all the controls in the parent chain have the Visible property set. In other words a control is Showing if it is Visible, its parent control is Visible, any parent control of the parent control is Visible, and so forth. (Run-time only and read-only.)
TabOrder	All windowed controls	Determines the control's tab order in its parent control.
TabStop	All windowed controls	Determines whether the user can move the control with the Tab key.
Tag	All components	A long integer available to store custom undefined data.
Top	All controls	The vertical coordinate of the upper-left corner of the component.
UndockHeight	Most controls	The height of the control when it is undocked.
UndockWidth	Most controls	The width of the control when it is undocked.
Visible	All controls	Determines whether the control is visible (provided its parent is also visible, as described in the Showing property).
Width	All controls	The horizontal size of the control.

Since there is inheritance among components, it is interesting to see in which ancestor classes the most common properties are introduced. You can look at Figure 4.4 for an overview of the properties introduced by the topmost classes of the VCL hier-

140 This is also irrelevant today

archy. The following sections provide basic descriptions of some of these common properties.

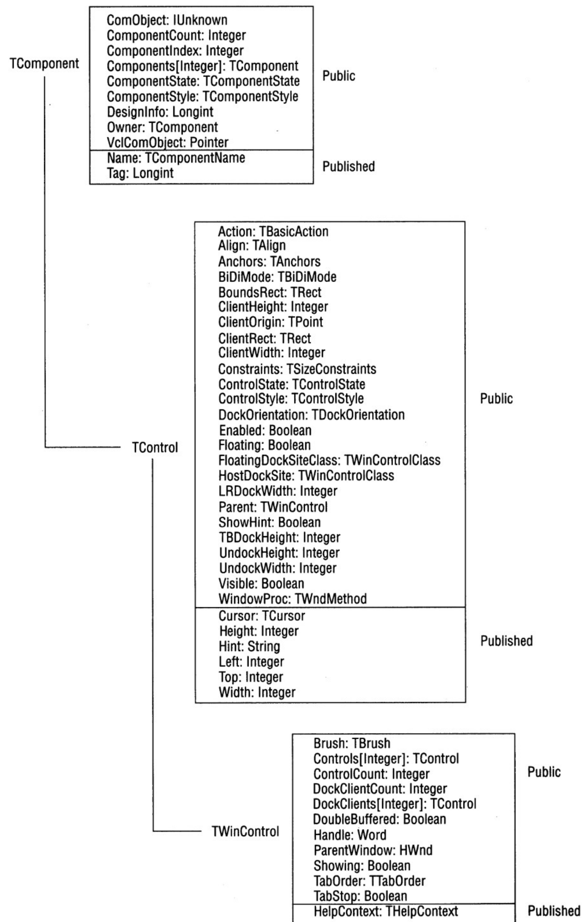


Figure 4.4: The properties introduced by the topmost classes of the VCL hierarchy and available in all of the subclasses. Image based on a picture of the original printed book.

The Name Property

Every component in Delphi should have a name. The name must be unique within the owner component, which is generally the form into which you place the component. This means that an application can have two different forms, each with a component with the same name, although you might want to avoid this practice to

prevent confusion. It is generally better to keep component names unique throughout an application.

Setting a proper value for the `Name` property is very important: If it's too long, you'll need to type a lot of code to use the object; if it's too short, you may confuse different objects. Usually the name of a component has a prefix with the component type; this makes the code more readable and allows Delphi to group components in the combo box of the Object Inspector, where they are sorted by name. There are three important elements related to the `Name` property of the components:

- First, the value of the `Name` property is used to define the name of the object in the declaration of the form class. This is the name you're generally going to use in the code to refer to the object. For this reason, the value of the name property must be a legal Pascal identifier.
- Second, if you set the `Name` property of a control before changing its `Caption` property, the new name is copied to the caption. That is, if the name and the caption are identical, then changing the name will also change the caption.
- Third, Delphi uses the name of the component to create the default name of the methods related to its events. If you have a `Button1` component, its default `OnClick` event handler will be called `Button1Click`, unless you specify a different name. If you later change the name of the component, Delphi will modify the names of the related methods accordingly. For example, if you change the name of the button to `MyButton`, the `Button1Click` method automatically becomes `MyButtonClick`.

The Components Array

Besides accessing a component by name, you can use the `Components` property of its owner, usually a form. Here is an example of the code you can use to add to a list box the names of all the components of a form (this code is actually part of the ChangeOwner example, presented in the next section):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Items.Clear;
  for I := 0 to ComponentCount - 1 do[141]
    ListBox1.Items.Add (Components [I].Name);
```

141 These days, you can also navigate the components owned by a component using a *for..in* loop.

```
end;
```

This code uses the `ComponentCount` property, which holds the total number of components owned by the current form, and the `Components` property, which is actually the list of the owned components. When you access a value from this list you get a value of the `TComponent` type. For this reason you can directly use only the properties common to all components, such as the `Name` property. To use properties specific to particular components, you have to use the proper type-downcast (`as`).

In Delphi, there are some components that are also component containers: the GroupBox, the Panel, the PageControl, and, of course, the Form component. When you use these controls, you can add other components inside them. In this case, the container is the parent of the components (as indicated by the `Parent` property), while the form is their owner (as indicated by the `Owner` property). You can use the `Controls` property of a form or group box to navigate the child controls, and you can use the `Components` property of the form to navigate all the owned components, regardless of their parent.

Using the `Components` property, we can always access each component of a form. If you need access to a specific component, however, instead of comparing each name with the name of the component you are looking for, you can let Delphi do this work, by using the `FindComponent` method of the form. This method simply scans the `Components` array looking for a name match.

The Owner Property

Every component usually has an owner. When a component is created at design time (or from the resulting DFM file) its owner will invariably be its form. When you create a component at run time, the owner is passed as a parameter to the `Create` constructor.

The owner is a read-only property, so you cannot change it. However, you can affect its value by calling the `InsertComponent` and `RemoveComponent` methods of the owner itself, passing the current component as parameter. Using these methods you can change a component's owner. However, you cannot apply them directly in an event handler of a form, as we attempt to do here:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RemoveComponent (Button1);
  Form2.InsertComponent (Button1);
end;
```

This code produces a memory access violation, because when you call `RemoveComponent`, Delphi disconnects the component from the form field (`Button1`), setting it to `nil`. The solution is to write a procedure like this:

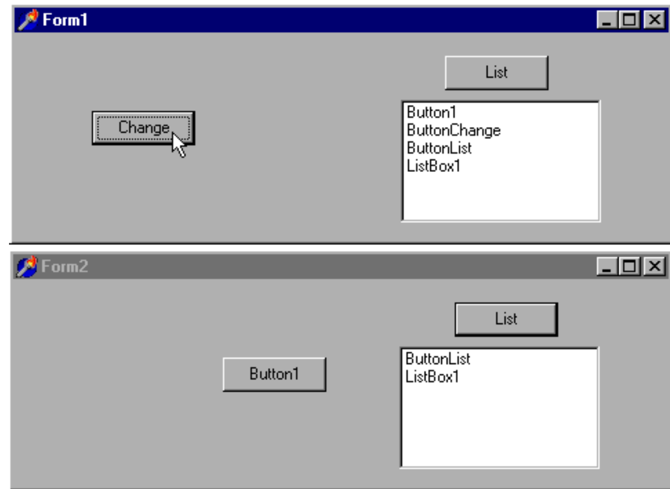
```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
  Component.Owner.RemoveComponent (Component);
  NewOwner.InsertComponent (Component);
end;
```

This method (extracted from the ChangeOwner example) changes the owner of the component. It is called along with the simpler code used to change the parent component; the two commands combined move the button *completely* to another form, changing its owner:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
  if Assigned (Button1) then
  begin
    // change parent
    Button1.Parent := Form2;
    // change owner
    ChangeOwner (Button1, Form2);
  end;
end;
```

The method checks whether the `Button1` field still refers to the control, because while moving the component, Delphi will set `Button1` to `nil`. You can see the effect of this code in Figure 4.5.

Figure 4.5:
In the ChangeOwner example, pressing the Change button moves the Button1 component to the second form. Image from the original book.



To demonstrate that the `Owner` of the `Button1` component actually changes, I've added another feature to both forms. The List button fills the list box with the names of the components each form owns, using the procedure shown in the previous section. Press the two List buttons before and after moving the component, and you'll see what happens behind the scenes. As a final feature, the `Button1` component has a simple handler for its `OnClick` event, to display the caption of the owner form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage ('My owner is ' +
    ((Sender as TButton).Owner as TForm).Caption);
end;
```

Removing Form Fields

Every time you add a component to a form, Delphi adds its complete description, including all of its properties, to the DFM file. To the Pascal file, Delphi adds the corresponding field in the form class declaration. When the form is created, Delphi loads the DFM file and uses it to re-create all the components and set their properties back. Then it hooks the new object with the form field corresponding to its `Name` property.

For this reason, it is certainly possible to have a component without a name. If your application will not manipulate the component or modify it at run time, you can

remove the component name from the Object Inspector. Examples are a static label with fixed text, or a menu item, or even more obviously menu item separators. By blanking out the name, you'll remove the corresponding element from the form class declaration. This reduces the size of the form object (by only four bytes, the size of the object reference) and it reduces the DFM file by not including a useless string (the component name). Reducing the DFM also implies reducing the final EXE file size, even if only slightly.

note If you blank out component names, just make sure to leave at least one named component of each class used on the form so that the smart linker will link in the required code. If, as an example, you remove from a form all the fields referring to labels, the Delphi linker will remove the implementation of the `TLabel` class from the executable file. The effect is that when the system loads the form at run time, it is unable to create an object of an unknown class and issues an error indicating that the class is not available.

You can also keep the component name and manually remove the corresponding field of the form class. Even if the component has no corresponding form field, it is created anyway, although using it (through the `FindComponent` method of the form, for example) will be a little more difficult.

Hiding Form Fields[142]

Many OOP purists complain that Delphi doesn't really follow the encapsulation rules, because all of the components of a form are mapped to public fields and can be accessed from other forms and units. However, Delphi does that only as a default to help beginners learn to use the Delphi visual development environment quickly. A programmer can follow a different approach and use properties and methods to operate on forms. The risk, however, is that another programmer of the same team might inadvertently bypass this approach, directly accessing the components if they are left in the published section. The solution, which many programmers don't know about, is to move the components to the private portion of the class declaration.

As an example, I've taken a very simple form with an edit box, a button, and a list box. When the edit box contains text and the user presses the button, the text is added to the list box. When the edit box is empty, the button is disabled. This is the simple code of the HideComp example:

142 This section is still very relevant today, given Delphi's architecture in terms of the form class structure hasn't changed.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  Button1.Enabled := Length (Edit1.Text) <> 0;
end;
```

I've listed these methods only to show you that in the code of a form we usually refer to the available components, defining their interactions. For this reason it seems impossible to get rid of the fields corresponding to the component. However, what we can do is hide them, moving them from the default published section to the private section of the form class declaration:

```
TForm1 = class(TForm)
  procedure Button1Click(Sender: TObject);
  procedure Edit1Change(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  Button1: TButton;
  Edit1: TEdit;
  ListBox1: TListBox;
end;
```

Now if you run the program you'll get in trouble: The form will load fine, but because the private fields are not initialized, the events above will use `nil` object references. Delphi usually initializes the published fields of the form using the components created from the DFM file. What if we do it ourselves, with the following code?

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1 := FindComponent ('Button1') as TButton;
  Edit1 := FindComponent ('Edit1') as TEdit;
  ListBox1 := FindComponent ('ListBox1') as TListBox;
end;
```

It will *almost* work, but it generates a system error, similar to the one we discussed in the previous section. This time, the private declarations will cause the linker to link in the implementations of those classes, but the problem is that the streaming system needs to know the names of the classes in order to locate the class reference needed to construct the components while loading the DFM file.

The final touch we need is some registration code to tell Delphi at run time about the existence of the component classes we want to use. We should do this before the form is created, so I generally place this code in the initialization section of the unit:

```
initialization
  RegisterClasses ([TButton, TEdit, TListBox]);
```

Now the question is, is this really worth the effort? What we obtain is a higher degree of encapsulation, protecting the components of a form from other forms (and other programmers writing them). I have to say that replicating these steps for each and every form can be tedious, and I'd really like to have a wizard generating this code for me on the fly while I do the standard operations in Delphi. However, for a large project built according to the principles of object-oriented programming, I recommend you consider this or a similar technique[143].

Properties Related to Control Size and Position

Other important properties, common to all controls, are those related to size and position. The position of a control is determined by its `Left` and `Top` properties; its size by the `Height` and `Width` properties. Technically, all components have a position, because when you reopen an existing form at design time, you want to be able to see the icons for the nonvisual components in exactly the position where you've placed them. This position is visible in the DFM file.

An important feature of the position of a component is that, like any other coordinate in Windows, it always relates to the client area of its parent component (which is the component indicated by its `Parent` property). For a form, the client area is the surface included within its borders (excluding the borders themselves). It would have been messy to work in screen coordinates, although there are some ready-to-use methods that convert the coordinates between the form and the screen and vice versa.

Note, however, that the coordinates of a control are always relative to the parent control, which is usually a form but can also be a panel or another *container* component. If you place a panel in a form, and a button in a panel, the coordinates of the button relate to the panel and not to the form containing the panel. In fact, in this case, the parent component of the button is the panel.

143 I've later build a Delphi Form Wizard, which can automate these steps. It is still available as part of my Cantools, see github.com/marcocantu/cantools.

Activation and Visibility Properties

There are two basic properties you can use to let the user activate or hide a component. The simplest is the `Enabled` property. When a component is disabled (when `Enabled` is set to `False`), there is usually some visual hint to specify this state to the user. At design time, the "disabled" property does not always have an effect, but at run time, disabled components are generally grayed.

For a more radical approach, you can completely hide a component, either by using the corresponding `Hide` method or by setting its `Visible` property to `False`. Be aware, however, that reading the status of the `Visible` property does not tell you if the control is actually visible. In fact, if the container of a control is hidden, even if the control is set to `Visible`, you cannot see it. For this reason, there is another property, `Showing`, which is a run-time and read-only property. You can read the value of `Showing` to know if the control is really visible to the user; that is, if it is visible, its parent control is visible, the parent control of the parent control is visible, and so on.

The Customizable Tag Property

The `Tag` property is a strange one, because it has no effect at all. It is merely an extra memory location, present in each component class, where you can store custom values. The kind of information stored and the way it is used are completely up to you.

It is often useful to have an extra memory location to attach information to a component without needing to define your component class. Technically, the `Tag` property stores a long integer[144] so that, for example, you can store the entry number of an array or list that corresponds to an object. Using typecasting, you can store in the `Tag` property a pointer, an object, or anything else that is four bytes wide. This allows a programmer to associate virtually anything with a component using its tag. We'll see how to use this property in several examples in future chapters, including the ODMenu examples in Chapter 5.

144 The property is now defined as *NativeInt*, so that its size with be different in a 32-bit or 64-bit application, matching the pointer size on each platform.

The User Interface: Color and Font

Two properties often used to customize the user interface of a component are `Color` and `Font`. There are several properties related to the color. The `Color` property itself usually refers to the background color of the component. Also, there is a `Color` property for fonts and many other graphic elements. Many components also have a `ParentColor` and a `ParentFont` property, indicating whether the control should use the same font and color as its parent component, which is usually the form. You can use these properties to change the font of each control on a form by setting only the `Font` property of the form itself.

When you set a font, either by entering values for the attributes of the property in the Object Inspector or by using the standard font selection dialog box, you can choose one of the fonts installed in the system. The fact that Delphi allows you to use all the fonts installed on your system has both advantages and drawbacks. The main advantage is that if you have a number of nice fonts installed, your program can use any of them. The drawback is that if you distribute your application, these fonts might not be available on your users' computers.

If your program uses a font that your user doesn't have, Windows will select some other font to use in its place. A program's carefully formatted output can be ruined by the font substitution. For this reason, you should probably rely only on standard Windows fonts (such as MS Sans Serif, System, Arial, Times New Roman, and so on). The alternative is to ship some fonts with your application, if the font's user license allows it.

There are a number of ways to set the value of a color. The type of this property is `TColor`. For properties of this type, you can choose a value from a series of predefined name constants or enter a value directly. The constants for colors include `clBlue`, `clSilver`, `clWhite`, `clGreen`, `clRed`, and many others. As a better alternative, you can use one of the colors used by Windows for system elements, such as the background of a window (`clWindow`), the color of the text of a highlighted menu (`clHightlightText`), the active caption (`clActiveCaption`), and the ubiquitous button face color (`clBtnFace`). All the color constants mentioned here are listed in Delphi's Help under the *TColor type* topic.

Another option is to specify a `TColor` as a number (a four-byte hexadecimal value) instead of using a predefined value. If you use this approach, you should know that the low three bytes of this number represent RGB color intensities for blue, green, and red, respectively. For example, the value `$00FF0000` corresponds to a pure blue color, the value `$0000FF00` to green, the value `$000000FF` to red, the value `$00000000`

to black, and the value `$00FFFFFF` to white. By specifying intermediate values, you can obtain any of the 16 million possible colors.

Instead of specifying these hexadecimal values directly, you should use the `RGB` function, which has three parameters, all ranging from 0 to 255. The first indicates the amount of red, the second the amount of green, and the last the amount of blue. Using the `RGB` function makes programs generally more readable than using a single hexadecimal constant.

note RGB is *almost* a Windows API function. It is defined by the Windows-related units and not by Delphi units, but a similar function does not exist in the Windows API. In C, there is a macro that has the same name and effect, so this is a welcome addition to the Pascal interface to Windows.

The highest-order byte of the `TColor` type is used to indicate which palette should be searched for the closest matching color, but palettes are too advanced a topic to discuss here. (Sophisticated imaging programs also use this byte to carry transparency information for each display element on the screen.) Regarding palettes and color matching, note that Windows sometimes replaces an arbitrary color with the closest available solid color, at least in video modes that use a palette. This is always the case with fonts, lines, and so on. At other times, Windows uses a dithering technique to mimic the requested color by drawing a tight pattern of pixels with the available colors. In 16-color (VGA) adapters[145] and at higher resolutions, you often end up seeing strange patterns of pixels of different colors and not the color you had in mind.

Common VCL Methods

Component methods are just like any other methods. There are procedures and functions you can call to perform the corresponding action. As mentioned earlier, you can often use methods to accomplish the same effect as reading or writing a property. Usually, the code is easier to read and understand when you use properties. However, not all methods have corresponding properties. Most of them are procedures, which execute an action instead of reading or writing a value. Again, some methods are available in all of the components; other methods are shared only by controls (visual components), and so on. Table 4.2 lists some common compo-

145 This tells you how old this book is, as this was current hardware back than!

nent methods. We'll see examples of using most of these methods throughout the book[146].

Table 4.2: Some Methods Available for Most VCL Components

Method	Available For	Description
BeginDrag	All controls	Starts manual dragging.
BringToFront	All controls	Puts the control in front of all others.
CanFocus	All controls	Determines whether the control will accept the keyboard input focus.
ClientToScreen	All controls	Translates client coordinates into screen coordinates.
ContainsControl	All controls	Determines whether a certain control is contained by the current one.
Create	All components	Creates a new instance (constructor).
Destroy	All components	Destroys the instance (destructor). You should actually call Free.
Dragging	All controls	Indicates whether the controls are being dragged.
EndDrag	All controls	Manually terminates dragging.
ExecuteAction	All components	Activates the action connected with the component.
FindComponent	All components	Returns the component in the Components array property having a given name (we've just used it in the HideComp example).
FlipChildren	All windowed controls	Moves child controls from the left side to the right side and vice versa. Used for supporting right-to-left languages (such as Arabic or Hebrew), along with the IsRightToLeft property.
Focused	All windowed controls	Determines whether the control has the focus.

146 These remains a fairly good list today, as well. Same for the list of events below.

`Free`	All components	Deletes the object from memory (forms should use the Release method).
`GetTextBuf`	All controls	Retrieves the text (or caption) of the control.
`GetTextLen`	All controls	Returns the length of the text (or caption) of the control.
`HandleAllocated`	All controls	Returns `True` if a system handle has been allocated for the control.
`HandleNeeded`	All controls	Allocates a corresponding system handle if one doesn't already exist.
`Hide`	All controls	Makes the control invisible (the same as setting the `Visible` property to `False`).
`InsertComponent`	All components	Adds a new element to the list of owned components.
`InsertControl`	All controls	Adds a new element to the list of controls that are the children of the current one.
`Invalidate`	All controls	Forces a repaint of the control.
`ManualDock`	All controls	Manually activates docking.
`ManualFloat`	All controls	Sets the docking control as a floating one.
`RemoveComponent`	All components	Removes a component from the `Components` list.
`ScaleBy`	All controls	Scales the control by a given percentage.
`ScreenToClient`	All controls	Translates screen coordinates into client coordinates.
`ScrollBy`	All controls	Scrolls the contents of the control.
`SendToBack`	All controls	Puts the control behind all the others.
`SetBounds`	All controls	Changes the position and size of the control (faster than accessing the related properties one by one).
`SetFocus`	All controls	Gives the input focus to the control.

SetTextBuf	All controls	Sets the text (or caption) of the control.
Show	All controls	Makes the control visible (the same as setting the Visible property to True).
Update	All controls	Immediately repaints the control, if there are pending painting requests.

Common VCL Events

Just as there is a set of properties common to all components, there are some events that are available for all of them. Table 4.3 provides short descriptions of these events. Again, this table is meant only as a starting point. You'll see examples using most of these events throughout the book.

Table 4.3: Some Events Available for Most Components

Event	Available For	Description
OnCanResize	Many controls	Occurs when the control is resized and allows you to stop the operation.
OnChange	Many components	Occurs when the object or its data change.
OnClick	Most controls	Occurs when the left mouse button is clicked over the component.
OnContextPopupMenu	All controls (new in Delphi 5)	Occurs when the user right-clicks the control. It allows you to do a different action than showing the attached popup menu.
OnDblClick	Many controls	Occurs when the user double-clicks with the mouse over the component.
OnDockDrop	Windowed controls	Occurs when the docking operation terminates over the current control.
OnDockOver	Windowed controls	Occurs when the user drags the mouse over the component during a docking operation.
OnDragDrop	Most controls	Occurs when a dragging operation terminates over the component; it is sent by the component that *received* the dragging operation.
OnDragOver	Most controls	Occurs when the user drags the mouse over the component.

OnEndDock	Most controls	Occurs when the docking operation of the current control terminates.
OnEndDrag	Most controls	Occurs when the dragging terminates; it is sent by the component that *started* the dragging operation.
OnEnter	All windowed controls	Occurs when the component is activated; that is, the component receives the focus.
OnExit	All windowed controls	Occurs when the component loses the focus.
OnGetSiteInfo	Windowed controls	Returns the control's docking information.
OnKeyDown	Some windowed controls	Occurs when the user presses a key on the keyboard; it is sent to the component with the input focus.
OnKeyPress	Some windowed controls	Occurs when the user presses a key; it is sent to the component with the input focus.
OnKeyUp	Some windowed controls	Occurs when the user releases a key; it is sent to the component with the input focus.
OnMouseDown	Most controls	Occurs when the user presses one of the mouse buttons; it is sent to the component under the mouse cursor.
OnMouseMove	Most controls	Occurs when the user moves the mouse over a component; it is sent to the component under the mouse cursor.
OnMouseUp	Most controls	Occurs when the user releases one of the mouse buttons; it is sent to the component under the mouse cursor.
OnMouseWheel, OnMouseWheelDown, OnMouseWheelUp	Windowed controls	Occur when the user rotates the mouse wheel or clicks on it as if it was a button.
OnResize	Most controls	Occurs when the resizing operation terminates.
OnStartDock	Most controls	Occurs when the user starts docking.
OnStartDrag	Most controls	Occurs when the user starts dragging; it is sent to the component *originating* the dragging operation.
OnUnDock	Windowed controls	Occurs when another control is undocked from the current one.

Understanding Frames

Chapter 1 introduced frames as one of the new features of Delphi 5. We've seen that you can create a new frame, place some components in it, write some event handlers for the components, and then add the frame to a form. In other words, a frame

is similar to a form, but it defines only a portion of a window, not a complete window. This is certainly not a feature worth a new construct. The totally new element of frames is that you can create multiple instances of a frame at design time and you can modify the class and the instance at the same time. This makes frames an effective tool for creating customizable composite controls at design time, something close to a visual component-building tool.

You are probably familiar with the Delphi concept of visual form inheritance (discussed in Chapter 2). You can work on both a base form and a derived form at design time, and any changes you make to the base form are propagated to the derived one, unless this overrides some property or event. With frames, you work on a class (as usual in Delphi), but the difference is that you can also customize one or more instances of the class created at design time. When you work on a form, you cannot change a property of the `TForm1` class for the `Form1` object at design time. With frames, you can.

Once you realize you are working with a class and one or more of its instances at design time, there is nothing more to understand about frames. In practice, frames are useful when you want to use the same group of components in multiple forms within an application. In this case, in fact, you can customize each of the instances at design time. Wasn't this already possible with component templates? It was, but component templates were based on the concept of copying and pasting some components and their code. There was no way to change the original definition of the template and see the effect in every place it was used. That is what happens with frames (and in a different way with visual form inheritance); changes to the original version (the class) are reflected in the copies (the instances).

There are many other uses of frames, which will become more apparent as Delphi programmers adopt this feature. Frames can be very useful when building multiple-page forms, as I'll demonstrate in Chapter 8.

Let's discuss a few more elements of frames with an example, called Frames2. This program has a frame with a list box, an edit box, and three buttons with simple code operating on the components. The frame also has a bevel aligned to its client area, because frames have no border. This is the definition of the frame in its own DFM file:

```
object FrameList: TFrameList
  Left = 0
  Top = 0
  Width = 202
  Height = 306
  TabOrder = 0
  object Bevel: TBevel
    Align = alClient
```

```
    Shape = bsFrame
  end
  object ListBox: TListBox...
  object Edit: TEdit
    Text = 'Some text'
  end
  object btnAdd: TButton
    Caption = '&Add'
    OnClick = btnAddClick
  end
  object btnRemove: TButton
    Caption = '&Remove'
    OnClick = btnRemoveClick
  end
  object btnClear: TButton
    Caption = '&Clear'
    OnClick = btnClearClick
  end
end
```

Of course, the frame has also a corresponding class, which looks like a normal form class:

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

What is different is that you can add the frame to a form. I've used two instances of the frame in the example (as you can see in Figure 4.6) and modified the behavior slightly. The first instance of the frame has the list box items sorted. When you change a property of a component of a frame, the DFM file of the hosting form will list the differences, as it does with visual form inheritance:

```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
```

```
    inherited ListBox: TListBox
      Sorted = True
    end
  end
  inline FrameList2: TFrameList
    Left = 232
    Top = 8
    inherited btnClear: TButton
      OnClick = FrameList2btnClearClick
    end
  end
end
```

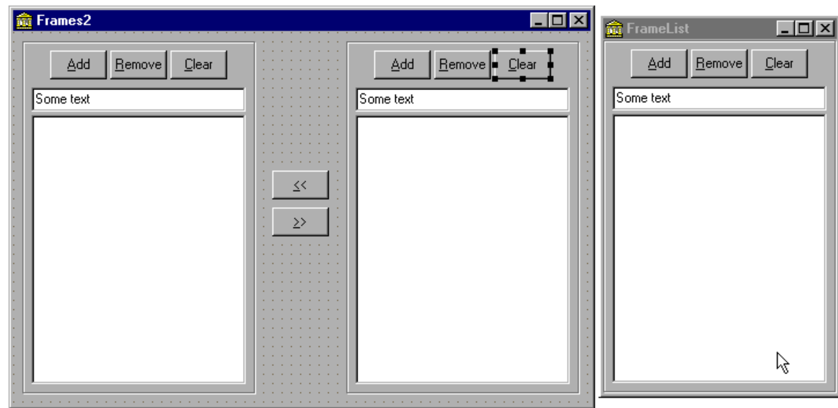


Figure 4.6:
A frame and two instances of it at design time, in the Frames2 example. Image from the original book.

As you can see from the listing, the DFM file for a form that has frames uses a new DFM keyword, `inline`. The references to the modified components of the frame, instead, use the `inherited` keyword, although this term is used with an extended meaning. `inherited` here doesn't refer to a base class we are inheriting from, but to the class we are instancing (or inheriting) an object from. It was probably a good idea, though, to use an existing feature of visual form inheritance and apply it to the new context. The effect of this approach, in fact, is that you can use the Revert to Inherited command of the Object Inspector or of the form to cancel the changes and get back to the default value of properties.

Notice also that unmodified components of the frame class are not listed in the DFM file of the form using the frame, and that the form has two frames with different names, but the components on the two frames have the same name. In fact, these components are not owned by the form, but are owned by the frame. This implies that the form has to reference those components through the frame, as you can see in the code for the buttons that copy items from one list box to the other:

```
procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
  FrameList1.ListBox.Items.AddStrings (
    FrameList2.ListBox.Items);
end;
```

Finally, besides modifying properties of any instance of a frame, you can change the code of any of its event handlers. If you double-click one of the buttons of a frame while working on the form (not on the stand-alone frame), Delphi will generate this code for you:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  FrameList2.btnClearClick(Sender);

end;
```

The line of code automatically added by Delphi corresponds to a call to the inherited event handler of the base class in visual form inheritance. This time, however, to get the default behavior of the frame we need to call an event handler and apply it to a specific instance—the frame object itself. The current form, in fact, doesn't include this event handler and knows nothing about it.

Whether you leave this call in place or remove it depends on the effect you are looking for. In the example I've decided to conditionally execute the default code, depending on the user confirmation:

```
procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
  if MessageDlg ('OK to empty the list box?',
      mtConfirmation, [mbYes, mbNo], 0) = idYes then
    // execute standard frame code
    FrameList2.btnClearClick(Sender);
end;
```

note By the way, note that because the event handler has some code, leaving it empty and saving the form won't remove it as usual: in fact, it isn't empty! Instead, if you simply want to omit the default code for an event, you need to add at least a comment to it, to avoid it being automatically removed by the system!

Lists and Container Classes

It is often important to handle groups of components or objects. Besides using standard arrays and dynamic arrays, there are a few classes of the VCL that represent lists of other objects. These classes can be divided into three groups: simple lists, collections, and containers. The last group has been introduced in Delphi 5.

Lists are represented by the generic list of objects, `TList`, and by the two lists of strings, `TStrings` and `TStringList`:

- `TList` defines a list of pointers, which can be used to store objects of any class[147]. A `TList` is more flexible than a dynamic array, because it is expanded automatically, simply by adding new items to it. The advantage of dynamic arrays over a `TList`, instead, is that dynamic arrays allow you to indicate a specific type for contained objects and perform the proper compile-time type checking.
- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves, such as a list box.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with their own storage. You can use this class to define a list of strings in a program.

The second group, collections, contains only two classes, `TCollection` and `TCollectionItem`. `TCollection` defines a homogeneous list of objects, which are owned by the collection class. The objects in the collection must be descendants of the `TCollectionItem` class. If you need a collection storing specific objects, you have to create both a subclass of `TCollection` and a subclass of `TCollectionItem`. Collections are invariably used to specify values of properties of components. It is very unusual to work with collections directly inside programs. All these lists have a number of methods and properties. You can operate on lists using the array notation (“`[`” and “`]`”) both to read and to change elements. There is a `Count` property, as well as typical access methods, such as `Add`, `Insert`, `Delete`, `Remove`, and search methods (for example, `IndexOf`).

`TStringList` and `TStrings` objects have both a list of strings and a list of objects associated with the strings. This opens up a number of different uses for these

147 Along with the introduction of the support for Generic programming in the Delphi language, the run-time library added a new *TList<T>* generic class, which can hold a list of objects of any specific class (and its sub-classes). Using a generic *TList<T>* makes applications more type safe and robust and it’s highly recommended.

classes. For example, you can use them for dictionaries of associated objects or to store bitmaps or other elements to be used in a list box.

note The `TListbox` component actually uses a `TStringList` object when it needs to store strings while its window handle is invalid; it uses a different descendant of `TStrings` object when it finally associates with a Windows list box control, which stores its own strings.

The two classes of lists of strings also have ready-to-use methods to store or load their contents to or from a text file, `SaveToFile` and `LoadFromFile`. To loop through a list, you can use a simple `for` statement based on its index, as if the list were an array.

Using Lists of Objects

We can write an example focusing on the use of the generic `TList` class. When you need a list of any kind of data, you can generally declare a `TList` object, fill it with the data, and then access the data while casting it to the proper type. The ListDemo example demonstrates just this. It also shows the pitfalls of this approach[148]. Its form has a private variable, holding a list of dates:

```
private
 ListDate: TList;
```

This list object is created when the form itself is created:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ListDate := TList.Create;
end;
```

A button of the form adds a random date to the list (of course, I've included in the project the unit containing the date component built in the previous chapter):

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
  ListDate.Add (TDate.Create (1900 + Random (200),
    1 + Random (12), 1 + Random (30)));
end;
```

148 This pitfalls can be overcome using the generic *TList<T>*.

When you extract the items from the list, you have to cast them back to the proper type, as in the following method, which is connected to the List button (you can see its effect in Figure 4.7):

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add ((
      TObject(ListDate [I]) as TDate).Text);
end;
```

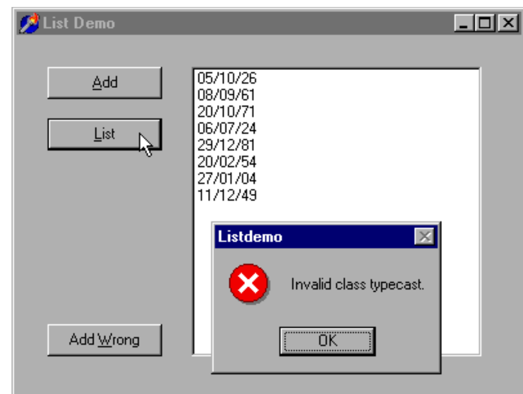


Figure 4.7: The list of dates shown by the ListDemo example. Image from the original book.

At the end of the code above, before we can do an `as` downcast, we first need to hard-cast the pointer returned by the `TList` into a `TObject` reference. This kind of expression can result in an invalid typecast exception, or it can generate a memory error when the pointer is not a reference to an object[149].

To demonstrate that things can indeed go wrong, I've added one more button, which adds a `TButton` object to the list:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add (Sender);
end;
```

149 Again, this can be addressed by using the generic class *TList<T>* based on the specific type of elements we want to add to the list.

If you click this button and then update one of the lists, you'll get an error. Finally, remember that when you destroy a list of objects, you should remember to destroy all of the objects of the list first. The ListDemo program does this in the `FormDestroy` method of the form:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListDate.Count - 1 do
    TObject(ListDate [I]).Free;
  ListDate.Free;
end;
```

Delphi 5 Container Classes

Delphi 5 introduces a new series of container classes, defined in the Contnrs unit. These classes extend the `TList` classes, by adding the idea of ownership and defining specific extraction rules (mimicking stacks and queues). The basic difference between `TList` and the new `TObjectList`[150] class is that the latter is defined as a list of `TObject` objects, not a list of pointers. Even more important, however, is the fact that if the object list has the `OwnsObjects` property set to `True`, it automatically deletes an object when it is replaced by another one and deletes each object when the list itself is destroyed. Here's a list of all the new container classes:

- The `TObjectList` class I've already described represents a list of objects, eventually owned by the list itself.
- The inherited class `TComponentList` represents a list of components, with full support for destruction notification (an important safety feature when two components are connected using their properties; that is, when a component is the value of a property of another component).
- The `TClassList` class is a list of class references. It inherits from `TList` and requires no destruction.
- The classes `TStack`[151] and `TObjectStack` represent lists of pointers and objects, from which you can only extract elements starting from the last one you've inserted. A stack follows the LIFO order (Last In, First Out). The typical methods

---

150 There is now also a generic version, *TObjectList<T>*, available in the *System.Generics.Collections* unit.

151 Or the better equivalent *TStack<T>* in the *System.Generics.Collections* unit.

of a stack are Push for insertion, Pop for extraction, and Peek to preview the first item without removing it. You can still use all the methods of the base class, `TList`.

- The classes `TQueue`[152] and `TObjectQueue` represent lists of pointers and objects, from which you always remove the *first* item you've inserted (FIFO: First In, First Out). The methods of these classes are the same as those of the stack classes but behave differently.

---

note Unlike the `TObjectList`, the `TObjectStack` and the `TObjectQueue` do not own the inserted objects and will not destroy those objects left in the data structure when it is destroyed. You can simply `Pop` all the items, destroy them once you're finished using them, and then destroy the container.

To demonstrate the use of these classes, I've modified the earlier ListDate example into the new Contain example. First, I changed the type of the `ListDate` variable to `TObjectList`. In the `FormCreate` method, I've modified the list creation to the following code, which activates the list ownership:

```
ListDate := TObjectList.Create (True);
```

At this point, we can simplify the destruction code, as applying `Free` to the list will automatically free the dates it holds.

I've also added to the program a stack and a queue object, filling each of them with numbers. One of the form's two buttons displays a list of the numbers in each container, and the other removes the last item (displayed in a message box):

```
procedure TForm1.btnQueueClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do
  begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

By pressing the two buttons, you can see that calling Pop for each container returns the last item. The difference is that the `TQueue` class inserts elements at the beginning, and the `TStack` class inserts them at the end.

---

152 Or the better equivalent TQueue<T> in the *System.Generics.Collections* unit.

Type-Safe Containers and Lists

Containers and lists have a problem: They are not type safe, as I've shown in both examples by adding a button object to a list of dates. To ensure that the data in a list is homogenous, you can check the type of the data you extract before you insert it, but as an extra safety measure you might also want to check the type of the data while extracting it. However, adding run-time type checking slows down a program and is risky—a programmer might fail to check the type in some cases.

To solve both problems, you can create specific list classes for given data types and fashion the code from the existing `TList` or `TObjectList` classes (or another container class). There are two approaches to accomplish this[153]:

- Derive a new class from the list class and customize the `Add` method and the access methods, which relate to the `Items` property. This is also the approach used by Borland for the container classes, which all derive from `TList`.
- Create a brand-new class that contains a `TList` object, and map the methods of the new class to the internal list using proper type checking. This approach defines a wrapper class, a class that "wraps" around an existing one to provide a different or limited access to its methods (in our case, to perform a type conversion).

I've implemented both solutions in the DateList example, which defines lists of `TDate` objects. In the listing below you'll find the declaration of the two classes, the inheritance-based `TDateListI` class and the wrapper class `TDateListW`.

```
type
// inheritance based
TDateListI = class (TObjectList)
protected
  procedure SetObject (Index: Integer; Item: TDate);
  function GetObject (Index: Integer): TDate;
public
  function Add (Obj: TDate): Integer;
  procedure Insert (Index: Integer; Obj: TDate);
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;
// wrapper based
TDateListW = class(TObject)
private
  FList: TObjectList;
```

153 There is now a much better and easier approach, which is using the generic container classes in the *System.Generics.Collections* unit.

```
  function GetObject (Index: Integer): TDate;
     procedure SetObject (Index: Integer; Obj: TDate);
  function GetCount: Integer;
public
  constructor Create;
  destructor Destroy; override;
  function Add (Obj: TDate): Integer;
  function Remove (Obj: TDate): Integer;
  function IndexOf (Obj: TDate): Integer;
  property Count: Integer read GetCount;
  property Objects [Index: Integer]: TDate
    read GetObject write SetObject; default;
end;
```

Obviously, the first class is simpler to write—it has fewer methods, and they simply call the inherited ones. The good thing is that a `TDateListI` object can be passed to parameters expecting a `TList`. The problem is that the code that manipulates an instance of this list via a generic `TList` variable will not be calling the specialized methods, because they are not virtual and might end up adding to the list objects of other data types.

Instead, if you decide not to use inheritance, you end up writing a lot of code, because you need to reproduce each and every one of the original `TList` methods, simply calling the methods of the internal `FList` object. The drawback is that the `TDateListW` class is not type compatible with `TList`, which limits its usefulness. It can't be passed as parameter to methods expecting a `TList`.

Both of these approaches provide good type checking. After you've created an instance of one of these list classes, you can add only objects of the appropriate type, and the objects you extract will naturally be of the correct type. This is demonstrated by the DateList example. This program has a few buttons, a combo box to let a user choose which of the lists to show, and a list box to show the actual values of the list. The program stretches the lists by trying to add a button to the list of `TDate` objects. To add an object of a different type to the `TDateListI` list, we can simply convert the list to its base class, `TList`. This might accidentally happen if you pass the list as a parameter to a method that expects a base class object. In contrast, for the `TDateListW` list to fail we must explicitly cast the object to `TDate` before inserting it, something a programmer should never do:

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);
begin
  ListW.Add (TDate(TButton.Create (nil)));
  TList(ListI).Add (TButton.Create (nil));
  UpdateList;
end;
```

The `UpdateList` call triggers an exception, displayed directly in the list box, because I've used an `as` type cast in the custom list classes. A wise programmer should never write the above code.

To summarize, writing a custom list for a specific type makes a program much more robust. Writing a wrapper list instead of one that's based on inheritance tends to be a little safer, although it requires more coding.

note Instead of rewriting wrapper-style list classes for different types, you can use my List Template Wizard, discussed in *Delphi Developer's Handbook* and available on my Web site.[154]

What's Next?

As we have seen in this chapter, Delphi includes a full-scale class library that is just as complete as Microsoft's MFC C++ class library. Delphi's VCL, of course, is much more component-oriented, and its classes offer a higher-level abstraction over the Windows API than the C++ libraries usually do.

To use components, you only need a clear understanding of the terminal nodes of the VCL hierarchy; that is, the components that show up in the Component Palette plus a few others. You really don't need a deeper knowledge of the VCL internals to use components; this knowledge is only necessary when you write new components or modify existing ones.

This chapter ends Part I of the book, which has covered the foundations of Delphi programming. Part II is fully devoted to examples of the use of the various components. We'll start in Chapter 5 with the advanced use of traditional Windows controls and menus, cover the `TForm` class in Chapter 6, and then examine toolbars, status bars, dialog boxes, and MDI applications in later chapters.

154 This is not available any more, given it's pretty much useless after the introduction of generics to the Delphi language and of generic collections in the RTL.

Chapter 5: Advanced Use Of The Standard Components

Now that you've been introduced to the Delphi environment and have seen an overview of the Object Pascal language and the Visual Component Library, we are ready to delve into the second part of the book: the use of components. This is really what Delphi is about. Visual programming using components is the key feature of this development environment.

Delphi comes with a number of ready-to-use components. I will not describe every component in detail, examining each of its properties and methods. If you need this

information, you can find it easily in the Help system. The aim of Part II of this book is to show you how to use some of the advanced features offered by the Delphi pre-defined components to build applications.

I'll start by trying to list all the various component alternatives you have, since choosing the right component is often a way to get into a project faster. This chapter presents the components in the Standard page of the Component Palette and some of the Win32 controls.

Opening the Component Tool Box

So you want to write a Delphi application[155]. You open a new Delphi project and find yourself faced with a large number of components. The problem is that for every operation there are multiple alternatives. For example, you can show a list of values using a list box, a combo box, a radio group, a string grid, a list view, or even a tree view if there is a hierarchical order. Which one should you use? That is difficult to say. There are many considerations, depending on what you want your application to do. For this reason I've provided a highly condensed summary of alternative options for a few common tasks.

note For some of the controls described in the following sections Delphi also includes a data-aware version, usually indicated by the DB prefix. As you'll see in Chapter 9, the DB version of a control typically serves a role similar to that of its "standard" equivalent; but the properties and the ways you use it are often quite different. For example, in an Edit control you use the `Text` property, while in a DBEdit component you access the `Value` of the related field object.

The Text Input Component

Although a form or a component can handle keyboard input directly, using the `OnKeyPress` event, this isn't a common operation. Windows provides ready-to-use controls you can use to get string input and even build a simple text editor. Delphi has several slightly different components in this area.

155 At the time of this book, using VCL for the UI was the only option. These days you can choose between VCL and FireMonkey, which has similar UI controls but is based on a completely different architecture. FireMonkey is not covered in this book, which is focused on VCL and Windows programming, because that's what was available in the Delphi 5 timeframe.

The Edit Component

The Edit component allows the user to enter a single line of text[156]. (You can also display a single line of text with a Label or a StaticText control, but these components are generally used only for fixed text or program-generated output, not for input.) The Edit component uses the `Text` property, whereas many other controls use the `Caption` property to refer to the text they display. The only condition you can impose on user input is the number of characters to accept. If you want to accept only specific characters, you can handle the `OnKeyPress` event of the edit box. For example, we can write a method that tests whether the character is a number or the Backspace key (which has a numerical value of 8). If it's not, we change the value of the key to the null character (`#0`), so that it won't be processed by the edit control and will produce a warning beep:

```
procedure TForm1.Edit1KeyPress(
  Sender: TObject; var Key: Char);
begin
  // check if the key is a number or backspace
  if not (Key in ['0'..'9', #8]) then
  begin
    Key := #0;
    Beep;
  end;
end;
```

The MaskEdit Component

To customize the input of an edit box further, you can use the MaskEdit component, which has an `EditMask` property. This is a string indicating for each character whether it should be uppercase, lowercase, or a number, and other similar conditions. You can see the editor of the `EditMask` property in Figure 5.1.

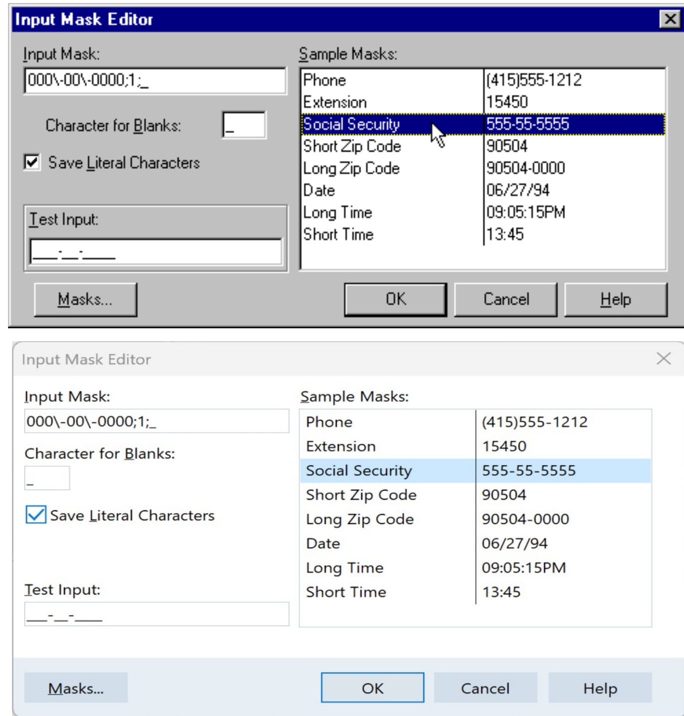
note You can display any property's editor by selecting the property in the Object Inspector and clicking the ellipsis (...) button.

The Input Mask editor allows you to enter a mask, but it also asks you to indicate a character to be used as a placeholder for the input and to decide whether to save the *literals* present in the mask, together with the final string. For example, you can choose to display the parentheses around the area code of a phone number only as

156 There is now also a NumberBox component, which is specific meant for the input of numeric values, including integers, floating point numbers, and currency. It's a much better solution comapred to the code in the snippet below to make an edit accept only numeric characters.

an input hint or to save them with the string holding the resulting number. These two entries in the Input Mask editor correspond to the last two fields of the mask (separated by semicolons).

Figure 5.1:
The MaskEdit component's EditMask property editor. Images captured in Delphi 5 and Delphi 12.



note Pressing the Masks button of the Mask Editor lets you choose predefined input masks for different countries.

The Memo and RichEdit Components

Both of the controls discussed so far allow a single line of input. The Memo component, by contrast, can host several lines of text but (on the Win95/98 platforms) still retains the 16-bit Windows 32KB text limit and allows only a single font for the entire text. You can work on the text of the memo line by line (using the `Lines` string list) or access the entire text at once (using the `Text` property).

If you want to host a large amount of text or change fonts and paragraph alignments, you should use the RichEdit control, a Win32 common control based on the

RTF document format. You can find an example of a complete editor based on the RichEdit component among the sample programs that ship with Delphi. (The example is named RichEdit, too.)

The RichEdit component has a `DefAttributes` property indicating the default styles and a `SelAttributes` property indicating the style of the current selection. These two properties are not of the `TFont` type, but they are compatible with fonts, so we can use the `Assign` method to copy the value, as in the following code fragment:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

Selecting Options

There are two standard Windows controls that allow the user to choose different options, as well as controls for grouping sets of options.

The CheckBox and RadioButton Components

The first is the *check box*, which corresponds to an option that can be selected regardless of the status of other check boxes. Setting the `AllowGrayed` property of the check box allows you to display three different states (selected, not selected, and grayed), which alternate as a user clicks on the check box.

The second type of control is the *radio button*, which corresponds to an exclusive selection. Two radio buttons on the same form or inside the same radio group container cannot be selected at the same time, and one of them should always be selected (as programmer, you are responsible for selecting one of the radio buttons at design time).

The GroupBox Components

To host several groups of radio buttons, you can use a GroupBox control to hold them together, both functionally and visually. To build a group box with radio buttons, simply place the GroupBox component on a form and then add the radio buttons to the group box.

You can handle the radio buttons individually, but it's easier to navigate through the array of controls owned by the group box, as discussed in the previous chapter. Here is a small code excerpt used to get the text of the selected radio button of a group:

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := (GroupBox1.Controls[I] as TRadioButton).Caption;
```

The RadioGroup Component

Delphi has a similar component that can be used specifically for radio buttons, the RadioGroup component. A RadioGroup is a group box with some radio button *clones* painted inside it. The term *clone* in this context refers to the fact that the RadioGroup component is a single control, a single window, with elements similar to radio buttons painted on its surface.

Using the radio group is generally easier than using the group box, since the various items are part of a list, as in a list box. This is how you can get the text of the selected item:

```
Text := RadioGroup1.Items [RadioGroup1.ItemIndex];
```

Technically, a RadioGroup uses fewer resources and less memory, and it should be faster to create and paint. Also, the RadioGroup component can automatically align its radio buttons in one or more columns (as indicated by the `Columns` property), and you can easily add new choices at run time, by adding strings to the `Items` string list. By contrast, adding new radio buttons to a group box would be quite complex.

Lists

When you have many selections, radio buttons are not appropriate. The usual number of radio buttons is no more than five or six, to avoid cluttering the user interface; when you have more choices, you can use a list box or one of the other controls that display lists of items and allow the selection of one of them.

The ListBox Component

The selection of an item in a list box uses the `Items` and `ItemIndex` properties as in the code shown above for the RadioGroup control. If you need access to the text of selected list box items often, you can write a small wrapper function like this:

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := ‘’;
end;
```

Another important feature is that by using the ListBox component, you can choose between allowing only a single selection, as in a group of radio buttons, and allowing multiple selections, as in a group of check boxes. You make this choice by specifying the value of the `MultiSelect` property. There are two kinds of multiple selections in Windows and in Delphi list boxes: *multiple selection* and *extended selection*. In the first case a user selects multiple items simply by clicking on them, while in the second case the user can use the Shift and Ctrl keys to select multiple consecutive or nonconsecutive items. This second choice is determined by the `ExtendedSelect` property.

For a multiple-selection list box, a program can retrieve information about the number of selected items by using the `SelCount` property, and it can determine which items are selected by examining the `Selected` array. This array of `Boolean` values has the same number of entries as the list box. For example, to concatenate all the selected items into a string, you can scan the `Selected` array as follows:

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

The ComboBox Component

List boxes take up a lot of screen space, and they offer a fixed selection. That is, a user can choose only among the items in the list box and cannot enter any choice that the programmer did not specifically foresee.

You can solve both problems by using a ComboBox control, which combines an edit box and a drop-down list. The behavior of a ComboBox component changes a lot depending on the value of its `Style` property. The `csDropDown` style defines a typical combo box, which allows direct editing and displays a list box on request, the `csDropDownList` style defines a combo box that does not allow editing (but uses the keystrokes to select an item), and the `csSimple` style defines a combo box that always displays the list box below it.

Note also that accessing the text of the selected value of a ComboBox is easier than doing the same operation for a list box, since you can simply use the `Text` property. A useful and common trick for combo boxes is to add a new element to the list when a user enters some text and presses the Enter key. The following method first tests whether the user has pressed that key, by looking for the character with the numeric (ASCII) value of 13. It then tests to make sure the text of the combo box is not empty and is not already in the list—if its position in the list is less than zero. Here is the code:

```
procedure TForm1.ComboBox1KeyPress(Sender: TObject; var Key: Char);
begin
  // if the user presses the Enter key
  if Key = Chr (13) then
    with ComboBox3 do
      if (Text <> '') and (Items.IndexOf (Text) < 0) then
        Items.Add (Text);
end;
```

The CheckListBox Component

Another extension of the list box control is represented by the CheckListBox component, a list box with each item preceded by a check box (as you can see in Figure 5.2). A user can select a single item of the list, but can also click on the check boxes to toggle their status. This makes the CheckListBox a very good component for multiple selections or for highlighting the status of a series of independent items (as in a series of check boxes).

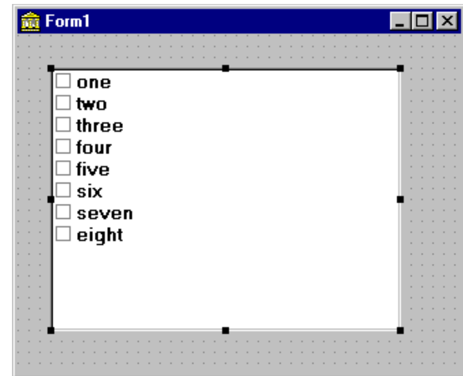


Figure 5.2:
The user interface of the CheckListBox control, basically a list of check boxes. Image from the original book.

To check the current status of each item, you can use the `Checked` and the `State` array properties (use the latter if the check boxes can be grayed). Delphi 5 introduces the `ItemEnabled` array property, which you can use to enable or disable each item of the list. We'll use the CheckListBox in the DragList example, later on in this chapter.

note Most of the list-based controls share a common and important feature. Each item of the list has an associated 32-bit value, usually indicated by the `TObject` type. This value can be used as a tag for each list item, and it's very useful for storing additional information along with each item. This approach is connected to a specific feature of the native Windows list box control, which offers four bytes of extra storage for each list box item. We'll use this feature in the ODList example later on in this chapter.

The ListView and TreeView Components

If you want an even more sophisticated list, you can use the ListView Win32 common control, which will make the user interface of your application look very modern. This component is slightly more complex to use, as described toward the end of this chapter. Other alternatives for listing values are the TreeView common control, which shows items in a hierarchical output, and the StringGrid control, which shows multiple elements for each line. The string grid control is described in Chapter 22, "Graphics in Delphi".[157]

If you use the common controls in your application, users will already know how to interact with them, and they will regard the user interface of your program as up to

157 This was originally a bonus chapter available as a separate download on the publisher web site, but it's now part of this ebook.

date. TreeView and ListView are the two key components of Windows Explorer, and you can assume that many users will be familiar with them, even more than with the traditional Windows controls.

Ranges

Finally, there are a few components you can use to select values in a range. Ranges can be used for numeric input and for selecting an element in a list.

The ScrollBar Component

The stand-alone ScrollBar control is the original component of this group, but it is seldom used by itself. Scroll bars are usually associated with other components, such as list boxes and memo fields, or are associated directly with forms. In all these cases, the scroll bar can be considered part of the surface of the other components. For example, a form with a scroll bar is actually a form that has an area resembling a scroll bar painted on its border, a feature governed by a specific Windows style of the form window. By *resembling,* I mean that it is not technically a separate window of the ScrollBar component type. These “fake” scroll bars are usually controlled in Delphi using specific properties of the form and the other components hosting them.

The TrackBar and ProgressBar Components

Direct use of the ScrollBar component is quite rare, especially with the TrackBar component introduced with Windows 95, which is used to let a user select a value in a range. Among Win32 common controls there is the companion ProgressBar control, which allows the program to output a value in a range, showing the progress of a lengthy operation.

The UpDown Component

Another related control is the UpDown component, which is usually connected to an edit box so that the user can either type a number in it or increase and decrease the number using the two small arrow buttons. To connect the two controls, you set the `Associate` property of the UpDown component. Nothing prevents you from using the UpDown component as a stand-alone control, displaying the current value in a label or in some other way.

The PageScroller Component

The Win32 PageScroller control is a container allowing you to scroll the internal control. For example, if you place a toolbar in the page scroller and the toolbar is larger than the available area, the PageScroller will display two small arrows on the side. Pressing these arrows will scroll the internal area. This component can be used as a scrollbar, but it also partially replaces the ScrollBox control.

The ScrollBox Component

The ScrollBox control represents a region of a form, which can scroll independently from the rest of the surface. For this reason the ScrollBox has two scrollbars used to move the embedded components. You can easily place other components inside a ScrollBox, as you do with a panel. In fact, a ScrollBox is basically a panel with scroll bars to move its internal surface, an interface element used in many Windows applications. When you have a form with many controls and a toolbar or status bar, you might use a ScrollBox to cover the central area of the form, leaving its toolbars and status bars outside of the scrolling region. By relying on the scrollbars of the form, in fact, you might allow the user to move the toolbar or status bar out of view, a very odd situation.

Dragging from One Component to Another

Now that you've been introduced to the standard controls, we'll examine a couple of general techniques: dragging and focus handling. Let me start with a simple example of dragging, called DragList. The form of this example, shown in Figure 5.3 at run time, contains a ListBox and a CheckListBox. You can drag items from one control to the other. It also has an edit box you can use to enter new items and drag them to either list. If you run the program, you'll see that there is also a rule: Lists cannot have duplicated items. This means we have to check whether the item is already in the list before inserting it.

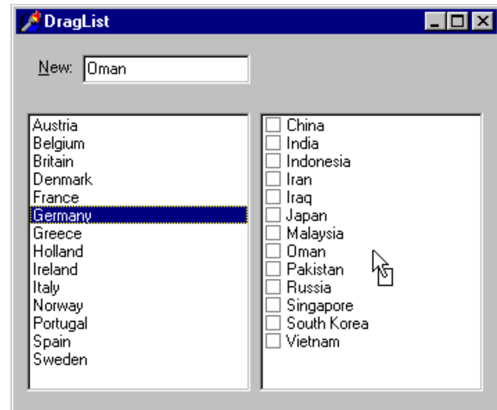


Figure 5.3: The form of the DragList example at run time, during a dragging operation. Image from the original book.

The two list boxes use the `dmAutomatic` value for the `DragMode` property (with the `DragKind` property left to the default value `dkDrag`). For the edit box, by contrast, we have to use manual dragging to let the edit box behave as usual when a user clicks on it. For this reason, as a user presses the mouse button over the edit box, we must initiate the dragging operation, delaying it as indicated by the first parameter of the `BeginDrag` method:

```
procedure TDragForm.Edit1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Edit1.BeginDrag (False, 10);
end;
```

The two lists share the same handler for the `OnDragOver` event, which is used to determine whether the control accepts dragging from a given source. In this handler, when the user is dragging from the edit box, the program checks to see whether the text is already in the list and disallows the dragging operation if it is. We can easily write a single event handler for both controls because they inherit from the same base class, `TCustomListBox`:

```
procedure TDragForm.ListDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
  // if the source is the edit and the items
  // is already in the destination list, reject it
  if (Source = Edit1) and
    ((Sender as TCustomListBox).Items.IndexOf (Edit1.Text) >= 0) then
    Accept := False;
end;
```

The handlers of the `OnDragDrop` events, however, are quite different, so I've decided to separate them. The list box allows only a single item to be selected, while the list check box can have multiple selected items; this makes the code quite different in the two cases. What can be shared is the code to add an item to a list only if it is not already there. I've added this shared code to a method of the form, which is called by both event handlers:

```
function TDragForm.AddNotDup (List: TCustomListBox;
  Text: string): Boolean;
begin
  // return if the string was not already in the list
  Result := List.Items.IndexOf (Text) < 0;
  if Result then
    List.Items.Add (Text);
end;
```

The code for the two drag-drop methods is quite simple. For the check list box, the program copies the text of the edit box or that of the selected list box item and removes it from the source:

```
procedure TDragForm.CheckListBox1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  nItem: Integer;
begin
  if Source = Edit1 then
    // copy the text of the edit box
    CheckListBox1.Items.Add (Edit1.Text)
  else if Source = ListBox1 then
  begin
    // copy if not duplicate
    nItem := ListBox1.ItemIndex;
    if AddNotDup (CheckListBox1, ListBox1.Items [nItem]) then
      // remove source item
      ListBox1.Items.Delete (nItem);
  end;
end;
```

For the list box, we have to scan all the items of the check list box to see which one is selected. Since we want to delete the items we copy, we must do this operation in reverse order, because deleting an item changes the position of the items that follow it:

```
procedure TDragForm.ListBox1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  I: Integer;
begin
  if Source = Edit1 then
    // copy the text of the edit box
```

```
    ListBox1.Items.Add (Edit1.Text)
  else if Source = CheckListBox1 then
  begin
    // copy all the selected items (unless duplicate)
    // and delete them (using reverse order!)
    for I := CheckListBox1.Items.Count - 1 downto 0 do
      if CheckListBox1.Checked [I] then
      begin
        if AddNotDup (ListBox1, CheckListBox1.Items [I]) then
          CheckListBox1.Items.Delete (I);
      end;
  end;
end;
```

note We'll see an example of dragging operations within a TreeView control at the end of this chapter.

Handling the Input Focus

Using the `TabStop` and `TabOrder` properties available in most controls, you can specify the order in which controls will receive the input focus when the user presses the Tab key. Instead of setting the tab order property of each component of a form manually, you can use the shortcut menu of the Form Designer to activate the Edit Tab Order dialog box, as shown in Figure 5.4.

Besides these basics settings, it is important to know that each time a component receives or loses the input focus, it receives a corresponding `OnEnter` or `OnExit` event. This allows you to fine-tune and customize the order of the user operations. Some of these techniques are demonstrated by the InFocus example, which creates a fairly typical password-login window. Its form has three edit boxes with labels indicating their meaning, as shown in Figure 5.5. At the bottom of the window is a status area with prompts guiding the user. Each item needs to be entered in sequence.

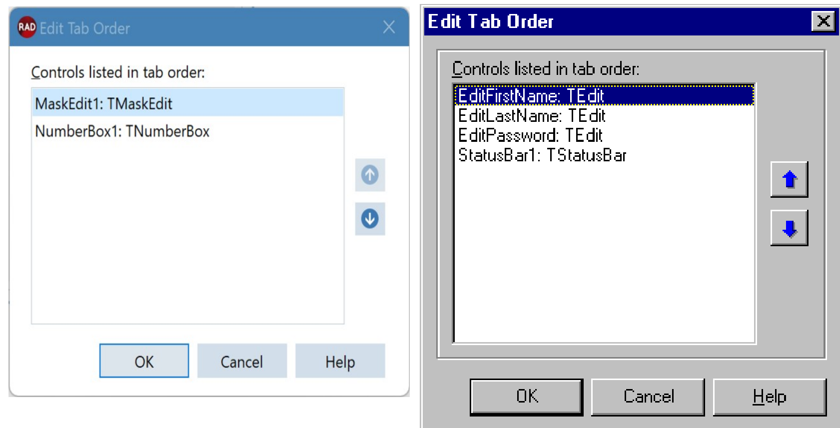


Figure 5.4: The Edit Tab Order dialog box. Images captured in Delphi 5 and Delphi 12.

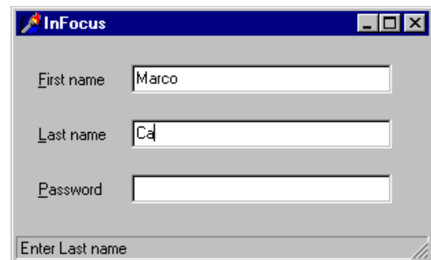


Figure 5.5: The InFocus example at run time. Image from the original book.

For the output of the status information I've used the StatusBar component, with a single output area (obtained by setting its `SimplePanel` property to `True`). Here is a summary of the properties for this example. Notice the `&` character in the labels, indicating a shortcut key, and the connection of these labels with corresponding edit boxes (using the `FocusControl` property):

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
  object Label1: TLabel
    Caption = '&First name'
    FocusControl = EditFirstName
  end
  object EditFirstName: TEdit
    OnEnter = GlobalEnter
    OnExit = EditFirstNameExit
  end
  object Label2: TLabel
```

```
    Caption = '&Last name'
    FocusControl = EditLastName
  end
  object EditLastName: TEdit
    OnEnter = GlobalEnter
  end
  object Label3: TLabel
    Caption = '&Password'
    FocusControl = EditPassword
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    OnEnter = GlobalEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
end
```

The program is very simple and does only two operations. The first is to identify, in the status bar, the edit control that has the focus. It does this by handling the controls' `OnEnter` event, possibly using a single generic event handler to avoid repetitive code. In the example, instead of storing some extra information for each edit box, I've checked each control of the form to determine which label is connected to the current edit box (indicated by the `Sender` parameter):

```
procedure TFocusForm.GlobalEnter(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ControlCount - 1 do
    // if the control is a label
    if (Controls [I] is TLabel) and
      // and the label is connected to the current edit box
      (TLabel(Controls[I]).FocusControl = Sender) then
    // copy the text leaving off the initial & character
    StatusBar1.SimpleText := 'Enter ' +
      Copy (TLabel(Controls[I]).Caption, 2, 1000);
end;
```

The second event handler of the form relates to the `OnExit` event of the first edit box. If the control is left empty, it refuses to release the input focus and sets it back before showing a message to the user. The methods also look for a given input value, automatically filling the second edit box and moving the focus directly to the third one:

```
procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
  if EditFirstName.Text = '' then
  begin
```

```
    // don't let the user get out
    EditFirstName.SetFocus;
    MessageDlg ('First name is required',
      mtError, [mbOK], 0);
  end
  else if EditFirstName.Text = 'Admin' then
  begin
    // fill the second edit and jump to the third
    EditLastName.Text := 'Admin';
    EditPassword.SetFocus;
  end;
end;
```

Working with Menus

Working with menus and menu items is generally quite simple. This section offers only some very brief notes and a few more advanced examples. The first thing to keep in mind about menu items is that they can serve different purposes:

- **Commands** are menu items used to execute an action.
- **State-setters** are menu items used to toggle an option on and off, to change the state of a particular element. These commands usually have a check mark on the left to indicate they are active.
- **Radio items** have a round check mark and are grouped to represent alternative selections, like radio buttons. To obtain radio menu items, simply set the `RadioItem` property to `True` and set the `GroupIndex` property for the alternative menu items to the same value.
- **Dialog menu items** cause a dialog box to appear and are usually indicated by an ellipsis (three dots) after the text.

As you enter new elements in the Menu Designer, Delphi creates a new component for each menu item and lists it in the Object Inspector (although nothing is added to the form). To name each component, Delphi uses the caption you enter and appends a number (so that *Open* becomes `Open1`). Because Delphi removes spaces and other special characters in the caption when it creates the name, and the menu item separators are set up using a hyphen as caption, these items would have an empty name. For this reason Delphi adds the letter N to the name, appending the number and generating items called `N1`, `N2`, and so on.

note Do not use the `Break` property, which is used to lay out a pull-down menu on multiple columns. The `mbMenuBarBreak` value indicates that this item will be displayed in a second or subsequent line, the `mbMenuBreak` value that this item will be added to a second or subsequent column of the pull-down.

Accelerator Keys in Delphi 5

In Delphi 5 you don't need to enter the `&` character in the `Caption` of a menu item; it provides an automatic accelerator key if you omit one. The Delphi 5 automatic accelerator key system can also figure out if you have entered conflicting accelerator keys and fix them on the fly. This doesn't mean you should stop adding custom accelerator keys with the `&` character, because the automatic system simply uses the first available letter, and it doesn't follow the default standards. You might also find better mnemonic keys than those chosen by the automatic system.

This new Delphi 5 feature is controlled by the `AutoHotkeys` property, which is available in the main menu component and in each of the pull-down menus and menu items. In the main menu, this property defaults to `maAutomatic`, while in the pull-downs and menu items it defaults to `maParent`, so that the value you set for the main menu component will be used automatically by all the subitems, unless they have a specific value of `maAutomatic` or `maManual`.

The engine behind this system is the `RethinkHotkeys` method of the `TMenuItem` class, and the companion `InternalRethinkHotkeys`. There is also a `RethinkLines` method, which checks whether a pull-down has two consecutive separators, or begins or ends with a separator. In all these cases the separator is automatically removed.

One of the reasons Delphi includes this new feature is the new ITE (Integrated Translation Environment)[158]. When you need to translate the menu of an application, it is convenient if you don't have to deal with the accelerator keys, or at least if you don't have to worry about whether two items on the same menu conflict. Having a system that can automatically resolve similar problems is definitely an advantage. Another motivation was Delphi's IDE itself. With all the dynamically loaded packages that install menu items in the IDE main menu or in pop-up menus, and with different packages loaded in different versions of the product, it's next to impossible

158 The VCL translation support has recently been removed as an official feature of the product and is only available as an additional download in the GetIt Package Manager. The foundations of the concepts remain valid and can be applicable to other, third-party, translation tools.

to get non-conflicting accelerator-key selections in each menu. That is why this mechanism isn't a wizard that does static analysis of your menus at design time; it was created to deal with the real problem of managing menus created dynamically at run time.

note This new feature is certainly very handy, but because it is active by default, it can break existing code. I had to modify two of this chapter's program examples from the previous edition of the book, just to avoid run-time errors caused by this change. As we'll see later, the problem is that I use the caption in the code, and the extra & broke my code. The change was quite simple, though, as all I had to do was to set the `AutoHotkeys` property of the main menu component to `maManual`.

Pop-Up Menus and the OnContextPopup Event

Besides the MainMenu component, you can use the similar PopupMenu component. This is typically displayed when the user right-clicks a component that uses the given pop-up menu as the value for its `PopupMenu` property.

However, besides connecting the pop-up menu to a component with the corresponding property, you can call its `Popup` method, which requires the position of the pop-up in screen coordinates. The proper values can be obtained by converting a local point to a screen point with the `ClientToScreen` method of the local component, in this code fragment a label:

```
procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (
      Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;
```

An alternative approach provided by Delphi 5 is the use of the `OnContextMenu` event. This brand-new event fires when a user right-clicks on a component, exactly what we've traced above with the test `if Button = mbRight`. The advantage is that the same event is also fired in response to a Shift+F10 key combination, as well as by any other user input methods defined by Windows Accessibility options or hard-

ware (including the shortcut menu key of some Windows-compatible keyboards). We can use this event to fire a pop-up menu with little code:

```
procedure TFormPopup.Label1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now),
    0, False, True, nil, 0, ‘’));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;
```

This example adds some dynamic behavior to the shortcut menu, adding a temporary item indicating when the pop-up menu is displayed. This is not particularly useful, but I’ve done it to highlight that if you need to display a plain pop-up menu, you can easily use the PopupMenu property of the control in question or one of its parent controls. Handling the `OnContextMenu` event makes sense only when you want to do some extra processing.

The `Handled` parameter is initialized to `False`, so that if you do nothing in the event handler, the normal pop-up menu processing will occur. If you do something in your event handler to replace the normal pop-up menu processing (such as popping up a dialog or a customized menu, as in this case), you should set `Handled` to `True` and the system will stop processing the message. Setting `Handled` to `True` should be fairly rare, as you’ll generally handle the `OnContextPopup` to dynamically create or customize the pop-up menu, but then you can let the default handler actually show the menu.

The handler of an `OnContextPopup` event isn’t limited to displaying a pop-up menu. It can do any other operation, such as directly display a dialog box. Here is an example of a right-click operation used to change the color of the control:

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
  ColorDialog1.Color := Label2.Color;
  if ColorDialog1.Execute then
    Label2.Color := ColorDialog1.Color;
  Handled := True;
```

```
end;
```

All the code snippets of this section are available in the simple CustPop example.

Creating Menu Items Dynamically

Besides defining the structure of a menu with the Menu Designer and modifying the status of the items using the `Checked`, `Visible`, and `Caption` properties, you can create an entire menu or portions of one at run time. This makes sense, for example, when you have many repetitive items, or when the menu items depend on some system configuration or user permissions.

The basic idea is that each object of the `TMenuItem` class—which Delphi uses for both menu items and pull-down menus—contains a list of menu items. Each of these items has the same structure, in a kind of recursive way. A pull-down menu has a list of submenus, and each sub-menu has a list of sub-menus, each with its own list of submenus, and so on. The properties you can use to explore the structure of an existing menu are `Items`, which contains the actual list of menu items, and `Count`, which contains the number of subitems. Adding new menu items or entire pull-down menus to an existing menu is fairly easy, particularly if you can write a single event handler for all of them.

This is demonstrated by the DynaMenu example, which also illustrates the use of menu check marks, radio items, and many other features of menus that aren't described in detail in the text. As soon as you start this program, it creates a new pull-down with menu items used to change the font size of a big label hosted by the form. Instead of creating a bunch of menu items with captions indicating sizes ranging from 8 to 48, you can let the program do this repetitive work for you.

The new pull-down menu should be inserted in the `Items` property of the `MainMenu1` component. You can calculate the position by asking the MainMenu component for the previous pull-down menu:

```
procedure TFormColorText.FormCreate(Sender: TObject);
var
  PullDown, Item: TMenuItem;
  Position, I: Integer;
begin
  // create the new pull-down menu
  PullDown := TMenuItem.Create (Self);
  PullDown.AutoHotkeys := maManual;
  PullDown.Caption := '&Size';
  PullDown.OnClick := SizeClick;
  // compute the position and add it
```

```
  Position := MainMenu1.Items.IndexOf (Options1);
  MainMenu1.Items.Insert (Position + 1, PullDown);
  // create menu items for various sizes
  I := 8;
  while I <= 48 do
  begin
    // create the new item
    Item := TMenuItem.Create (Self);
    Item.Caption := IntToStr (I);
    // make it a radio item
    Item.GroupIndex := 1;
    Item.RadioItem := True;
    // handle click and insert
    Item.OnClick := SizeItemClick;
    PullDown.Insert (PullDown.Count, Item);
    I := I + 4;
  end;
  // add extra item at the end
  Item := TMenuItem.Create (Self);
  Item.Caption := 'More...';
  // make it a radio item
  Item.GroupIndex := 1;
  Item.RadioItem := True;
  // handle it by showing the font selection dialog
  Item.OnClick := Font1Click;
  PullDown.Insert (PullDown.Count, Item);
end;
```

As you can see in the code above, the menu items are created in a `while` loop, setting the radio item style and calling the `Insert` method with the number of items as a parameter to add each item at the end of the pull-down. At the end, the program adds one extra item, which is used to set a different size than those listed. The `OnClick` event of this last menu item is handled by the `Font1Click` method (also connected to a specific menu item), which displays the font selection dialog box. You can see the dynamic menu in Figure 5.6.

note Because the program uses the `Caption` of the new items dynamically, we should either disable the `AutoHotkeys` property of the main menu component, or disable this feature for the pull-down menu we are going to add (and thus automatically disable it for the menu items). This is what I've done in the code above by setting the `AutoHotkeys` property of the dynamically created pull-down component to `maManual`. An alternative approach is to let the menu display the automatic captions and then call the new `StripHotkeys` function before converting then caption to a number. There is also a new `GetHotkey` function, which returns the *active* character of the caption.

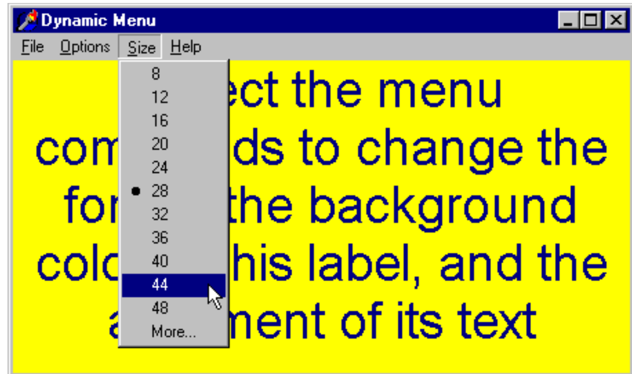


Figure 5.6:
The Size pull-down menu of the DynaMenu example is created at run time, along with all of its menu items. Image from the original book.

The handler for the `OnClick` event of these dynamically created menu items uses the caption of the `Sender` menu item to set the size of the font:

```
procedure TFormColorText.SizeItemClick(Sender: TObject);
begin
  with Sender as TMenuItem do
    Label1.Font.Size := StrToInt (Caption);
end;
```

This code doesn't set the proper radio item mark next to the selected item, because the user can select a new size also by changing the font. The proper radio item is checked in the `OnClick` event handler of the entire pull-down menu, which is connected just after the pull-down is created and activated just before showing the pull-down. The code scans the items of the pull-down menu (the `Sender` object) and checks whether the caption matches the current `Size` of the font. If no match is found, the program checks the last menu item, to indicate that a different size is active:

```
procedure TFormColorText.SizeClick (Sender: TObject);
var
  I: Integer;
  Found: Boolean;
begin
  Found := False;
  with Sender as TMenuItem do
  begin
    // look for a match, skipping the last item
    for I := 0 to Count - 2 do
      if StrToInt (Items [I].Caption) =
        Label1.Font.Size then
      begin
        Items [I].Checked := True;
        Found := True;
```

```
        System.Break; // skip the rest of the loop
      end;
    if not Found then
      Items [Count - 1].Checked := True;
  end;
end;
```

When you want to create a menu or a menu item dynamically, you can use the corresponding components, as I've done in the DynaMenu example. As an alternative, you can also use some global functions available in the Menus unit: `NewMenu`, `NewPopupMenu`, `NewSubMenu`, `NewItem`, and `NewLine`.

Using Menu Images

In Delphi it is very easy to improve a program's user interface by adding images to menu items. This is becoming common in Windows applications and it is very nice that Borland has added all the required support, making the development of graphical menu items trivial.

All you have to do is add an image list control to the form, add a series of bitmaps to the image list, connect the image list to the menu using its `Images` property, and set the proper `ImageIndex` property for the menu items. You can see the effect of these simple operations in Figure 5.7. (You can also associate a bitmap with the menu item directly, using the `Bitmap` property.)

note Delphi 5 makes the definition of images for menus more flexible, by allowing you to associate an image list with any specific pull-down menu (and even a specific menu item) using the new `SubMenuImages` property. Having a specific and smaller image list for each pull-down menu, instead of one single huge image list for the entire menu, allows for more run-time customization of an application.

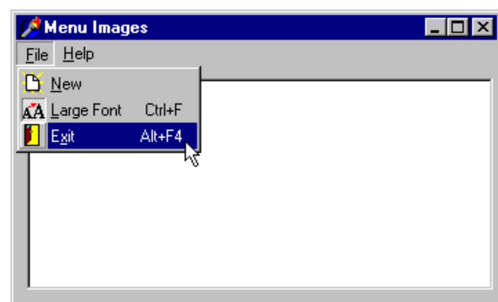
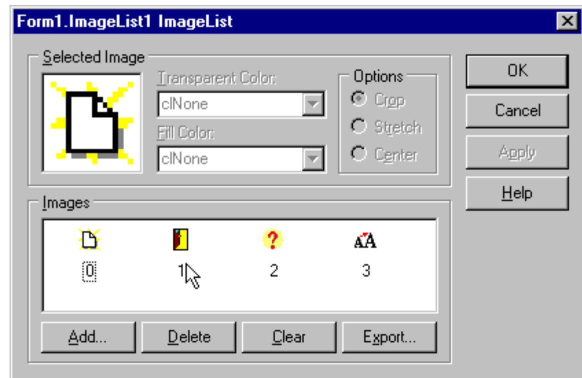


Figure 5.7: The simple graphical menu of the MenuImg example. Image from the original book.

To create the image list you can double-click on the component, activating the corresponding editor (shown in Figure 5.8), and then import existing bitmap or icon files. You can actually prepare a single large bitmap and let the image editor divide it according to the `Height` and `Width` properties of the ImageList component, which refer to the size of the individual bitmaps in the list.

Figure 5.8:
The Image List editor, with the bitmaps of the MenuImg example. Image from the original book.



note As an alternative, you can use the series of images that ship with Delphi[159] and are stored by default in the `Program Files/Common Files/Borland Shared/Images/Buttons` directory. Each bitmap contains both an “enabled” and a “disabled” image. As you import them, the Image List editor will ask you whether to split them in two, a suggestion you should accept. This operation adds to the image list a normal image and a disabled one, which is not generally used (as it can be built automatically when needed). For this reason I generally delete the disabled part of the bitmap from the Image List.

The program’s code is very simple. The only element I want to emphasize is that if you set the `Checked` property of a menu item with an image instead of displaying a check mark, the item paints its image as sunken. You can see this in the Large Font menu of the MenuImg example in Figure 5.7. Here is the code for that menu item selection:

```
procedure TForm1.LargeFont1Click(Sender: TObject);
begin
  if Memo1.Font.Size = 8 then
    Memo1.Font.Size := 12
  else
```

159 These images are no longer available. The GetIt Package manager offers a nice collection of images, called Icons8 (licensed under Creating Commons), but you can find many others available online.

```
    Memo1.Font.Size := 8;
  // changes the image style near the item
  LargeFont1.Checked := not LargeFont1.Checked;
end;
```

Customizing the System Menu

In some circumstances, it is interesting to add menu commands to the system menu itself, instead of (or besides) having a menu bar. This might be useful for secondary windows, toolboxes, windows requiring a large area on the screen, and "quick-and-dirty" applications. Adding a single menu item to the system menu is straightforward:

```
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_SEPARATOR, 0, '');
AppendMenu (GetSystemMenu (Handle, FALSE),
  MF_STRING, idSysAbout, '&About...');
```

This code fragment (extracted from the `OnCreate` event handler of the SysMenu example) adds a separator and a new item to the system menu item. The `GetSystemMenu` API function, which requires as a parameter the handle of the form, returns a handle to the system menu. The `AppendMenu` API function is a general-purpose function you can use to add menu items or complete pull-down menus to any menu (the menu bar, the system menu, or an existing pull-down menu). When adding a menu item, you have to specify its text and a numeric identifier. In the example I've defined this identifier as:

```
const
  idSysAbout = 100;
```

Adding a menu item to the system menu is easy, but how can we handle its selection? Selecting a normal menu generates the `wm_Command` Windows message. This is handled internally by Delphi, which activates the `OnClick` event of the corresponding menu item component. The selection of system menu commands, instead, generates a `wm_SysCommand` message, which is passed by Delphi to the default handler. Windows usually needs to do something in response to a system menu command.

We can intercept this command and check to see whether the command identifier (passed in the `CmdType` field of the `TWmSysCommand` parameter) of the menu item is our `idSysAbout`. Since there isn't a corresponding event in Delphi, we have to define a new message-response method for the form class:

```
public
  procedure WMSysCommand (var Msg: TMessage);
    message wm_SysCommand;
```

The code of this procedure is not very complex. We just need to check whether the command is our own and call the default handler:

```
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu example');
  inherited;
end;
```

To build a more complex system menu, instead of adding and handling each menu item as we have just done, we can follow a different approach. Just add a MainMenu component to the form, create its structure (any structure will do), and write the proper event handlers. Then reset the value of the `Menu` property of the form, removing the menu bar.

Now we can add some code to the SysMenu example to add each of the items from the hidden menu to the system menu. This operation takes place when the button of the form is pressed. The corresponding handler uses generic code that doesn't depend on the structure of the menu we are appending to the system menu:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  // add a separator
  AppendMenu (GetSystemMenu (Handle, FALSE), MF_SEPARATOR, 0, '');
  // add the main menu to the system menu
  with MainMenu1 do
    for I := 0 to Items.Count - 1 do
      AppendMenu (GetSystemMenu (Self.Handle, FALSE),
        mf_Popup, Items[I].Handle, PChar (Items[I].Caption));
  // disable the button
  Button1.Enabled := False;
end;
```

note This code uses the expression `Self.Handle` to access the handle of the form. This is required because we are currently working on the `MainMenu1` component, as specified by the `with` statement.[160]

160 This is, in fact, a very good reason to avoid the use of the with statement in the first place. In retrospective, I don't like the fact I was encouraging this and I really don't like this code snippet. I decided to keep it offers me a good opportunity to explain this is not good code.

The menu flag used in this case, `mf_Popup`, indicates that we are adding a pull-down menu. In this function call the fourth parameter is interpreted as the handle of the pull-down menu we are adding (in the previous example we passed the identifier of the menu, instead). Since we are adding to the system menu items with sub-menus, the final structure of the system menu will have two levels, as you can see in Figure 5.9.

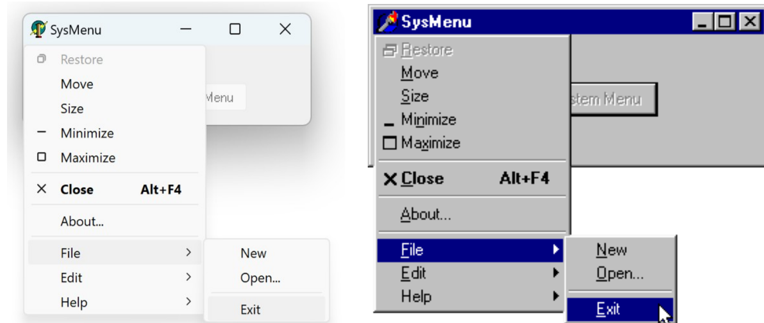


Figure 5.9: The second-level system menu items of the SysMenu example are the result of copying a complete main menu to the system menu. Images captured in Delphi 5 and Delphi 12.

note The Windows API uses the terms *pop-up menu* and *pull-down menu* interchangeably. This is really odd, because most of us use the terms to mean different things. Pop-up menus are shortcut menus, and pull-down menus are the secondary menus of the menu bar. Apparently, Microsoft uses the terms in this way because the two elements are implemented with the same kind of internal windows; and the fact that they are two distinct user-interface elements is probably something that was later conceptually built over a single basic internal structure.

Once you have added the menu items to the system menu, you need to handle them. Of course, you can check for each menu item in the `WMSysCommand` method, or you can try building a smarter approach. Since in Delphi it is easier to write a handler for the `OnClick` event of each item, we can look for the item corresponding to the given identifier in the menu structure. Delphi helps us by providing a `FindItem` method.

When (and if) we have found a main menu item that corresponds to the item selected in the system menu, we can call its `Click` method (which invokes the `OnClick` handler). Here is the code I've added to the `WMSysCommand` method:

```
var
  Item: TMenuItem;
begin
  ...
  Item := MainMenu1.FindItem (Msg.CmdType, fkCommand);
```

```
if Item <> nil then
  Item.Click;
```

In this code, the `CmdType` field of the message structure that is passed to the `WMSysCommand` procedure holds the command of the menu item being called.

note You can also use a simple `if` or `case` statement to handle one of the system menu's predefined menu items that have special codes for this identifier, such as `sc_Close`, `sc_Minimize`, `sc_Maximize`, and so on. For more information, you can see the description of the `wm_SysCommand` message in the Windows API Help file.

This application works but has one glitch. If you click the right mouse button over the Taskbar icon representing the application, you get a plain system menu (actually different from the default one). The reason is that this system menu belongs to a different window, the window of the `Application` global object. I'll discuss the `Application` object, and update this example to make it work with the Taskbar button, in Chapter 6.

The ActionList Component[161]

As explained in the previous chapter, Delphi's event architecture is very open: You can write a single event handler and connect it to the `OnClick` events of a toolbar button and a menu. You can also connect the same event handler to different buttons or menu items, as the event handler can use the `Sender` parameter to refer to the object that fired the event by using the `Sender` parameter. It's a little more difficult to synchronize the status of toolbar buttons and menu items. If you have a menu item and a toolbar button that both toggle the same option, every time the option is toggled, you must both add the check mark to the menu item and change the status of the button to show it pressed.

To overcome this problem, Delphi 4 introduced an event-handling architecture based on actions. An action (or command) both indicates the operation to do when a menu item or button is clicked and determines the status of all the elements connected to the action. The connection of the action with the user interface of the

161 This is a fundamental feature of the VCL architecture, still incredibly modern and still largely unused by Delphi developers. I want to underline the fact this was a great idea and it remains very important today to move form a pure RAD visual development to a much more flexible architecture based on visual design, but separating the UI from the application logic.

linked controls is very important and should not be underestimated, because it is where you can get the real advantages of this architecture.

note If you have ever written code using the MFC class library of Visual C++, you'll recognize that a Delphi action maps to both a command and a `CCommandUpdateUI` object. The Delphi architecture is more flexible, though, because it can be extended by sub-classing the action classes.

There are many players in this event-handling architecture. The central role is certainly played by the action objects. Action objects have a name, like any other component, and they have other properties that will be applied to the linked controls (called action clients). These properties include the `Caption`, the graphical representation (`ImageIndex`), the status (`Checked`, `Enabled`, and `Visible`), and the user feedback (`Hint` and `HelpContext`). The base class for an action object is `TBasicAction`. There is a `TAction` class, but it inherits from `TCustomAction`, which derives from `TContainedAction`, which in turn descends from `TBasicAction`, a `TComponent` subclass.

Each action object is connected to one or more client objects through an ActionLink object. Multiple controls, possibly of different types, can share the same action object, as indicated by their `Action` property. Technically, the ActionLink objects maintain a bidirectional connection between the client object and the action. The ActionLink object is required because the connection works in both directions. An operation on the object (such as a click) is forwarded to the action object and results in a call to its `OnExecute` event; an update to the status of the action object is reflected in the connected client controls. In other words, one or more client controls can create an ActionLink, which registers itself with the action object.

You should not set the properties of the client controls you connect with an action, because the action will override the property values of the client controls. For this reason you should generally write the actions first and then create the menu items and buttons you want to connect with them. Note also that when an action has no `OnExecute` handler, the client control is automatically disabled (or grayed), unless the `DisableIfNoHandler` property is set to `False`.

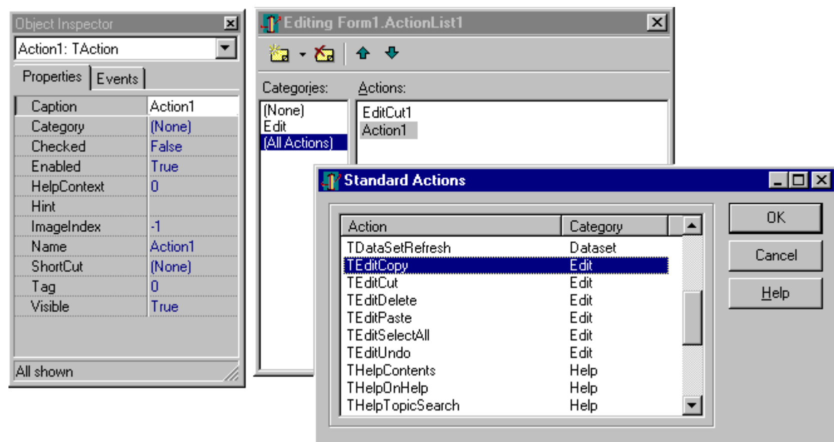
The client controls connected to actions are usually menu items and various types of buttons (push buttons, check boxes, radio buttons, speed buttons, toolbar buttons, and the like), but nothing prevents you from creating new components that hook into this architecture. Component writers can even define new actions and new link action objects.

Besides a client control, some actions can also have a target component. Some predefined actions hook to a specific target component (for examples, see the coverage of the DataSet components in the Chapter 9 section "Looking for Records in a

Table”). Other actions automatically look for a target component in the form that supports the given action, starting with the active control.

Finally, the action objects are held by an ActionList component, the only class of this architecture that shows up on the Component Palette. The action list receives the execute actions that aren’t handled by the specific action objects, firing the `OnExecuteAction`. If even the action list doesn’t handle the action, Delphi calls the `OnExecuteAction` event of the `Application` object. The ActionList component has a special editor you can use to create a number of actions, as you can see in Figure 5.10.

Figure 5.10: The ActionList component editor, with a list of predefined actions you can use. Image from the original book.



In the editor, actions are displayed in different groups, as indicated by their `Category` property. By simply setting this property to a brand-new value, you instruct the editor to introduce a new category. These categories are basically logical groups, although in some cases a group of actions can work only on a specific type of target component. You might want to define a category for every pull-down menu or group them in some other logical way.

With the action list editor, you can create a brand new action or choose one of the existing actions registered in the system. These are listed in a secondary dialog box, as shown in Figure 5.10. There are many predefined actions, which can be divided into logical groups[162]:

162 There are many additional groups added to the predefined list of actions, including DataSnap Client, Dialog, File, Format (for RichEdit operations), Internet, Search, Tab, and Tools. There are almost 70 predefined actions in Delphi 12.

- **Edit actions**, illustrated in the next example. They include Cut, Copy, and Paste actions.
- **MDI window actions**, which will be demonstrated in Chapter 8, as we examine the Multiple Document Interface approach. They include all the most common MDI operations: Arrange, Cascade, Close, Tile, and Minimize all.
- **Dataset actions**, which relate to database tables and queries and will be discussed in Chapter 11. There are many dataset actions, representing all the main operations you can perform on a dataset.
- **Help actions,** which allow you to activate the contents page or index of the Help file attached to the application.

---

note You can also define new custom actions and register them in Delphi's IDE, as we'll see in Chapter 13.

Besides handling the `OnExecute` event of the action and changing the status of the action to affect the user interface of the client controls, an action can also handle the `OnUpdate` event, which is activated when the application is idle. This gives you the opportunity to check the status of the application or the system and change the user interface of the controls accordingly. For example, the standard PasteEdit action enables the client controls only when there is some text in the Clipboard.

Actions in Practice

Now that you understand the main ideas behind this very important Delphi feature, let's try out an example. The program is called Actions and demonstrates a number of features of the action architecture.

I began building it by placing a new ActionList component in its form and adding the three standard edit actions and a few custom ones. The form also has a panel with some speed buttons, a main menu, and a Memo control (the automatic target of the edit actions). This is the list of the actions, extracted from the DFM file:

```
object ActionList1: TActionList
  Images = ImageList1
  object ActionCopy: TEditCopy
    Category = 'Edit'
    Caption = '&Copy'
    Hint = 'Copy'
    ImageIndex = 1
```

```
  ShortCut = <Ctrl+C>
end
object ActionCut: TEditCut
  Category = 'Edit'
  Caption = 'Cu&t'
  Hint = 'Cut'
  ImageIndex = 0
  ShortCut = <Ctrl+X>
end
object ActionPaste: TEditPaste
  Category = 'Edit'
  Caption = '&Paste'
  Hint = 'Paste'
  ImageIndex = 2
  ShortCut = <Ctrl+V>
end
object ActionNew: TAction
  Category = 'File'
  Caption = '&New'
  Hint = 'New'
  ImageIndex = 3
  ShortCut = <Ctrl+N>
  OnExecute = ActionNewExecute
end
object ActionExit: TAction
  Category = 'File'
  Caption = 'E&xit'
  Hint = 'Exit'
  ImageIndex = 5
  ShortCut = <Alt+F4>
  OnExecute = ActionExitExecute
end
object NoAction: TAction
  Category = 'Test'
  Caption = '&No Action'
  Hint = 'No Action'
end
object ActionCount: TAction
  Category = 'Test'
  Caption = '&Count Chars'
  Hint = 'Count Characters'
  ImageIndex = 6
  OnExecute = ActionCountExecute
  OnUpdate = ActionCountUpdate
end
object ActionBold: TAction
  Category = 'Edit'
  Caption = '&Bold'
  Hint = 'Bold'
  ImageIndex = 4
  ShortCut = <Ctrl+B>
  OnExecute = ActionBoldExecute
end
```

```
  object ActionEnable: TAction
    Category = 'Test'
    Caption = '&Enable NoAction'
    Hint = 'Enable No Action'
    OnExecute = ActionEnableExecute
  end
  object ActionSender: TAction
    Category = 'Test'
    Caption = 'Test &Sender'
    Hint = 'Test Sender'
    OnExecute = ActionSenderExecute
  end
end
```

note The shortcut keys are stored in the DFM files using virtual key numbers, which also include values for the Ctrl and Alt keys. In this and other listings throughout the book I've replaced the numbers with the literal values, enclosing them in angle brackets.

All of these actions are connected with the items of a MainMenu component and some of them also with the buttons of a Toolbar control (more on the Toolbar control in Chapter 7). Notice that the images selected in the ActionList control affect the actions in the editor only, as you can see in Figure 5.11. For the images of the ImageList to show up also in the menu items and in the toolbar buttons, you must also select the image list in the MainMenu and in the Toolbar components.

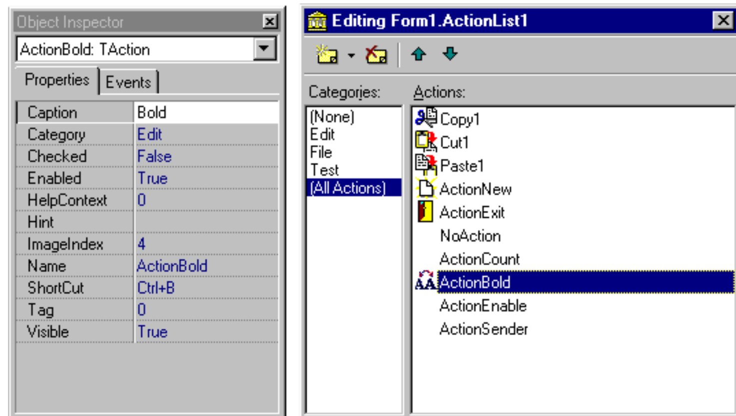


Figure 5.11: The ActionList editor of the Actions example. Image from the original book.

The three predefined actions for the Edit menu don't have associated handlers, but these special objects have internal code to perform the related action on the active edit or memo control. These actions also enable and disable themselves, depending on the content of the Clipboard and on the existence of selected text in the active

edit control. Most other actions have custom code, except for the `NoAction` object. Having no code, the menu item and the button connected with this command are disabled, even if the `Enabled` property of the action is set to `True`.

I've added to the example, and to the Test menu, another action that enables the menu item connected to the `NoAction` object:

```
procedure TForm1.ActionEnableExecute(Sender: TObject);
begin
  NoAction.Enabled := True;
  NoAction.DisableIfNoHandler := False;
  ActionEnable.Enabled := False;
end;
```

Simply setting `Enabled` to `True` will produce the effect for only a very short time, unless you set the `DisableIfNoHandler` property, as discussed in the previous section. Once this operation is done, I disable the current action, since there is no need to issue the same command again.

This is different from an action you can toggle, such as the Edit ➢ Bold menu item and the corresponding speed button. Here is the code of the Bold action:

```
procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
  with Memo1.Font do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
  // toggle status
  ActionBold.Checked := not ActionBold.Checked;
end;
```

The `ActionCount` object has very simple code, but it demonstrates an `OnUpdate` handler; when the memo control is empty, it is automatically disabled. We could have obtained the same effect by handling the `OnChange` event of the memo control itself, but in general it might not always be possible or easy to determine the status of a control simply by handling one of its events. Here is the code of the two handlers of this action:

```
procedure TForm1.ActionCountExecute(Sender: TObject);
begin
  ShowMessage ('Characters: ' + IntToStr (
    Length (Memo1.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
  ActionCount.Enabled := Memo1.Text <> '';
```

```
end;
```

Finally, I've added a special action to test the sender object of the action event handler and get some other system information. Besides showing the object class and name, I've added code that accesses the action list object. I've done this mainly to show that you can access this information and how to do it:

```
procedure TForm1.ActionSenderExecute(Sender: TObject);
begin
  Memo1.Lines.Add (
    'Sender class: ' + Sender.ClassName);
  Memo1.Lines.Add (
    'Sender name: ' + (Sender as TComponent).Name);
  Memo1.Lines.Add (
    'Category: ' + (Sender as TAction).Category);
  Memo1.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name );
  end;
```

You can see the output of this code in Figure 5.12, along with the user interface of the example. Notice that the `Sender` is not the menu item you've selected, even if the event handler is connected to it. The `Sender` object, which fires the event, is the action, which intercepts the user operation.

Finally, keep in mind that you can also write handlers for the events of the ActionList object itself, which play the role of global handlers for all the actions of the list (something I haven't done in the example).

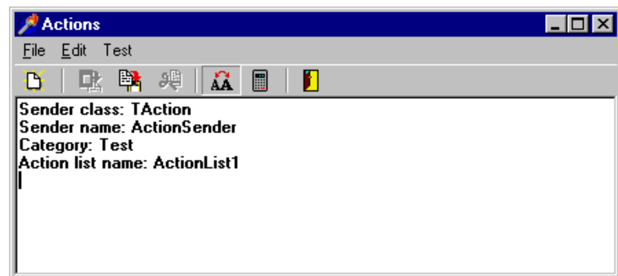


Figure 5.12: The Actions example, with a detailed description of the Sender of an Action object's OnExecute event. Image from the original book.

Owner-Draw Controls

Let's return briefly to menu graphics. Besides using an ImageList to add glyphs to the menu items, you can turn a menu into a completely graphical element, using the owner-draw technique. The same technique also works for other controls, such as list boxes. In Windows, the system is usually responsible for painting buttons, list boxes, edit boxes, menu items, and similar elements. Basically these controls know how to paint themselves. As an alternative, however, the system allows the owner of these controls, generally a form, to paint them. This technique, available for buttons, list boxes, combo boxes, and menu items, is called *owner-draw*.

In Delphi the situation is slightly more complex. The components can take care of painting themselves in this case (as in the `TBitBtn` class for bitmap buttons) and possibly activate corresponding events. Basically, the system sends the request for painting to the owner (usually the form), and the form forwards the event back to the proper control, firing its event handlers.

note Most of the Win32 common controls have support for the owner-draw technique, generally called custom drawing. You can fully customize the appearance of a ListView, a TreeView, a TabControl, a PageControl, a HeaderControl, a StatusBar, and a ToolBar. In Delphi 5 the ToolBar, ListView and TreeView controls also support *advanced* custom drawing, a more fine-tuned drawing capability introduced by Microsoft in the latest versions of the Win32 common controls library. The downside to owner-draw is that when the Windows user interface style changes in the future (and it always does), your owner-draw controls that fit in perfectly with the current user interface styles will look outdated and out of place. Since you are creating a custom user interface, you'll need to keep it updated yourself. By contrast, if you use the standard output of the controls, your applications will automatically adapt to a new version of such controls.

Owner-Draw Menu Items

Delphi makes the development of graphical menu items quite simple compared to the traditional approach of the Windows API. You set the `OwnerDraw` property of a menu item component to `True` and handle its `OnMeasureItem` and `OnDrawItem` events.

In the `OnMeasureItem` event you can determine the size of the menu items. This event handler is activated once for each menu item when the pull-down menu is displayed and has two reference parameters you can set:

```
procedure ColorMeasureItem (Sender: TObject;
```

```
  ACanvas: TCanvas; var Width, Height: Integer);
```

The other parameter, `ACanvas`, is typically used to determine the height of the current font.

In the `OnDrawItem` event you paint the actual image. This event handler is activated every time the item has to be repainted. This happens when Windows first displays the items and each time the status changes; for example, when the mouse moves over an item, it should become highlighted. In fact, to paint the menu items, we have to consider all the possibilities, including drawing the highlighted items with specific colors, drawing the check mark if required, and so on. Luckily enough the Delphi event passes to the handler the `Canvas` where it should paint, the output rectangle, and the status of the item (selected or not):

```
procedure ColorDrawItem(Sender: TObject;
  ACanvas: TCanvas; ARect: TRect; Selected: Boolean);
```

In the ODMenu example I'll handle the highlighted color, but skip other advanced aspects (such as the check marks). I've set the `OwnerDraw` property of the menu and written handlers for some of the menu items. To write a single handler for each event of the three color-related menu items, I've set their `Tag` property to the value of the actual color in the `OnCreate` event handler of the form:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Blue1.Tag := clBlue;
  Red1.Tag := clRed;
  Green1.Tag := clGreen;
end;
```

This makes the handler of the actual `OnClick` event of the items quite straightforward:

```
procedure TForm1.ColorClick(Sender: TObject);
begin
  ShapeDemo.Brush.Color :=
    (Sender as TComponent).Tag
end;
```

The handler of the `OnMeasureItem` event doesn't depend on the actual items, but uses a fixed value (different from the handler of the other pull-down):

```
procedure TForm1.ColorMeasureItem(Sender: TObject;
  ACanvas: TCanvas; var Width, Height: Integer);
begin
  Width := 80;
  Height := 30;
end;
```

The most important portion of the code is in the handlers of the `OnDrawItem` events. For the color, we use the value of the tag to paint a rectangle of the given color, as you can see in Figure 5.13. Before doing this, however, we have to fill the background of the menu items (the rectangular area passed as a parameter) with the standard color for the menu (`clMenu`) or the selected menu items (`clHighlight`):

```
procedure TForm1.ColorDrawItem(Sender: TObject;
  ACanvas: TCanvas; ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top,
    ARect.Right, ARect.Bottom);
end;
```

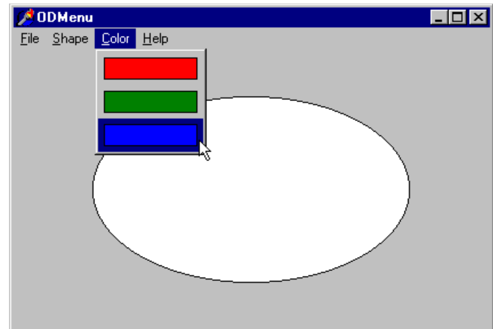


Figure 5.13: The owner-draw menu of the ODMenu example. Image from the original book.

The three handlers for this event of the Shape pull-down menu items are all different, although they use similar code:

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // draw the ellipse
```

```
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top,
    ARect.Right, ARect.Bottom);
end;
```

note To accommodate the increasing number of states in the Windows 2000 user interface style, Delphi 5 includes a new `OnAdvancedDrawItem` event for menus.

A ListBox of Colors

As we have just seen for menus, list boxes have an owner-draw capability, which means a program can paint the items of a list box. The same support is provided for combo boxes. To create an owner-draw list box, we set its `Style` property to `lbOwnerDrawFixed` or `lbOwnerDrawVariable`. The first value indicates that we are going to set the height of the items of the list box by specifying the `ItemHeight` property and that this will be the height of each and every item. The second owner-draw style indicates a list box with items of different heights. In this case the component will trigger the `OnMeasureItem` event for each item, to ask the program for their heights.

In the ODList example, I'll stick with the first, simpler, approach. The example stores color information along with the items of the list box and then draws the items in colors (instead of using a single color for the whole list). Here are the properties of the components of the main form of this example:

```
object ODListForm: TODListForm
  Caption = 'Owner-draw Listbox'
  OnCreate = FormCreate
  object ListBox1: TListBox
    Align = alClient
    Font.Charset = ANSI_CHARSET
    Font.Color = clBlack
    Font.Height = -32
    Font.Name = 'Arial'
    Font.Style = [fsBold]
    ItemHeight = 16
    ParentFont = False
    Sorted = True
    Style = lbOwnerDrawFixed
    OnDblClick = ListBox1DblClick
    OnDrawItem = ListBox1DrawItem
  end
  object ColorDialog1: TColorDialog...
```

```
end
```

Notice the value of the `TextHeight` attribute of the form, which indicates the number of pixels required to display text. This is the value we should use for the `ItemHeight` property of the list box. An alternative solution is to compute this value at run time, so that if we later change the font at design time we don't have to remember to set the height of the items accordingly.

note I've just described `TextHeight` as an *attribute* of the form, not a property. And in fact it isn't a property but a local value of the form. If it is not a property, you might ask, how does Delphi save it in the `DFM` file? Well, the answer is that Delphi's streaming mechanism is based on properties plus special *property-clones* created by the `DefineProperties` method. You can refer to the Delphi Help file or to *Delphi Developer's Handbook* for information about this advanced topic.

Since `TextHeight` is *not* a property, although it is listed in the form description, we cannot access it directly. Studying the VCL source code, I found that this value is computed by calling a private method of the form, `GetTextHeight`. Since it is private, we cannot call this function. What we can do is to duplicate its code (which is actually quite simple) in the `FormCreate` method of the form, after selecting the font of the list box:

```
Canvas.Font := ListBox1.Font;
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

The next thing we have to do is add some items to the list box. Since this is a list box of colors, we want to add color names to the `Items` of the list box and the corresponding color values to the `Objects` data storage related to each item of the list. Instead of adding the two values separately, I've written a procedure to add new items to the list:

```
procedure TODListForm.AddColors (Colors: array of TColor);
var
  I: Integer;
begin
  for I := Low (Colors) to High (Colors) do
    ListBox1.Items.AddObject (
      ColorToString (Colors[I]),
      TObject(Colors[I]));
end;
```

This method uses an open-array parameter, an array of an undetermined number of elements of the same type. (See the online tutorial *Essential Pascal* at `www.marcocantu.com` if you are unfamiliar with this language construct.) For each item passed as a parameter, we add the name of the color to the list, and we add its value to the related data, by calling the `AddObject` method. To obtain the string cor-

responding to the color, we call the Delphi `ColorToString` function. This returns a string containing either the corresponding color constant, if any, or the hexadecimal value of the color. The color data is added to the list box after casting its value to the `TObject` data type (a four-byte reference), as required by the `AddObject` method.

note Besides `ColorToString`, which converts a color value into the corresponding string with the identifier or the hexadecimal value, there is also a Delphi function to convert a properly formatted string into a color, `StringToColor`.

In the ODList example this method is called in the `OnCreate` event handler of the form (after previously setting the height of the items):

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime,
  clGray, RGB (213, 23, 123), RGB (0, 0, 0),
  clAqua, clNavy, clOlive, clPurple, clTeal]);
```

The code used to draw the items is not particularly complex. We simply retrieve the color associated with the item, set it as the color of the font, and then draw the text:

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TListbox do
  begin
    // erase
    Canvas.FillRect(Rect);
    // draw item
    Canvas.Font.Color := TColor (Items.Objects [Index]);
    Canvas.TextOut(Rect.Left, Rect.Top, Listbox1.Items[Index]);
   end;
end;
```

The system already sets the proper background color, so the selected item is displayed properly even without any extra code on our part. You can see an example of the output of this program at startup in Figure 5.14. The example also allows you to add new items, by double-clicking on the list box:

```
procedure TODListForm.ListBox1DblClick(Sender: TObject);
begin
  if ColorDialog1.Execute then
    AddColors ([ColorDialog1.Color]);
end;
```

If you try using this capability, you'll notice that some colors you add are turned into color names (one of the Delphi color constants) while others are converted into hexadecimal numbers.

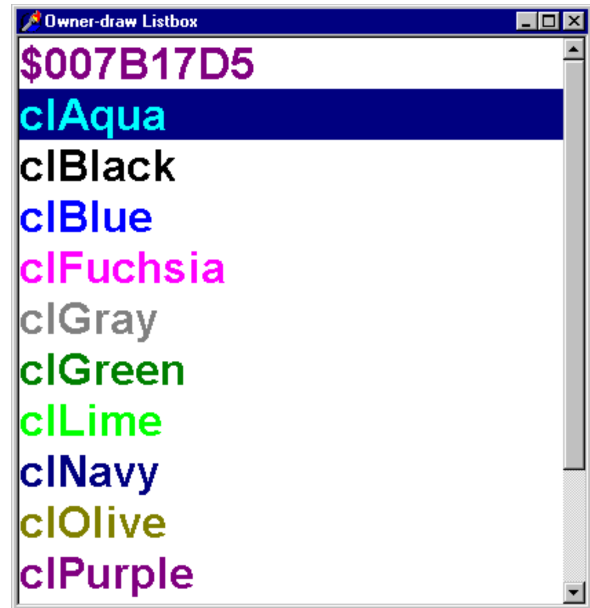


Figure 5.14: The output of the ODList example, with a colored owner-draw list box. Image from the original book.

ListView and TreeView

Although using an owner-draw list box is quite simple, this kind of list box is often replaced by the more powerful ListView and TreeView controls. Again, these two controls are part of the Win32 common controls, stored in the ComCtl32.DLL library.

Microsoft has kept expanding this library over the last two years, adding new controls such as the calendar and the coolbar, all available since Delphi 4, and extending the existing ones. Some of the versions of the library (distributed in particular along with the numerous versions of Microsoft Internet Explorer) have created compatibility problems with the controls, although the situation has apparently become more stable over the last year.

Some of these controls are complex, can be customized in a number of ways, and even support custom drawing features. Here I'll show you a couple of simple examples of the use of the TreeView and ListView components. In Chapters 7 and 8 we'll

use other common controls. In any case, I cannot provide extensive coverage of all of the features of these controls, which would require too much space.

A Graphical Reference List

When you use a ListView component, you can provide bitmaps both indicating the status of the element (for example, the selected item) and describing the contents of the item in a graphical way.

How do we connect the images to a list or tree? We need to refer to the ImageList component we've already used for the images of the menu. A ListView can actually have three image lists, one for the large icons (the `LargeImages` property), one for the small icons (the `SmallImages` property), and one used for the state of the items (the `StateImages` property).

To define the images of the RefList example, however, I used an alternative approach: I created a single big bitmap (16 x 80 pixels for five small images and 32 x 160 pixels for five large images) with all the images inside. Figure 5.15 shows these two bitmaps in the Delphi Image Editor[163]. Then I added the bitmap to a resource file and wrote some code to load it all at once (not one image at a time).

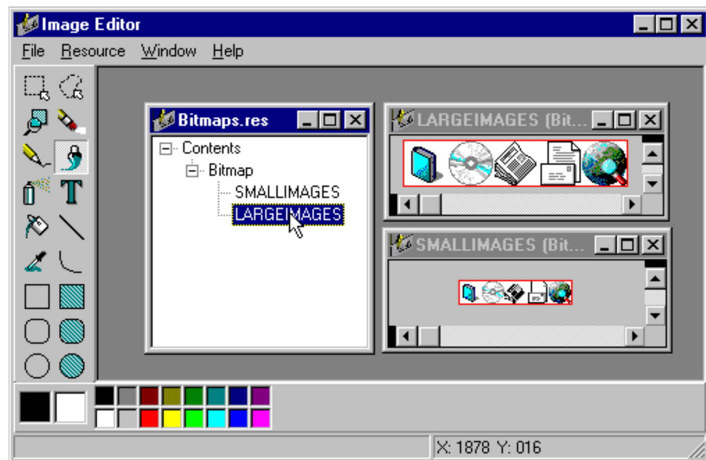


Figure 5.15: All the images of the ListView of the RefList example are stored in two bitmaps. Image from the original book (the Image Editor isn't available with Delphi any more).

I created two ImageList components at run time. As you can see in the parameter of the `Create` constructor, I assigned the form as their owner, so that I don't have to

163 This image editing tool is not available any more.

manually destroy them at the end. Here is the code of the handler for the first part of the form's OnCreate event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ImageList1, ImageList2: TImageList;
begin
  // load the large images
  ImageList1 := TImageList.Create (self);
  ImageList1.Height := 32;
  ImageList1.Width := 32;
  ImageList1.ResourceLoad (rtBitmap,
    'LargeImages', clWhite);
  ListView1.LargeImages := ImageList1;

  // load the small images
  ImageList2 := TImageList.Create (self);
  ImageList2.ResourceLoad (rtBitmap,
    'SmallImages', clWhite);
  ListView1.SmallImages := ImageList2;
```

Each of the items of the ListView has an ImageIndex, which refers to its image in the list. For this to work properly, the elements in the two image lists should follow the same order. When you have a fixed image list, you can add items to it using Delphi's ListView Item Editor, which is connected to the Items property. You can see an example of the use of this editor in Figure 5.16. In this editor you can define items and so-called subitems. The subitems are displayed only in the detailed view (when you set the vsReport value of the ViewStyle property) and are connected with the titles set in the Columns property.

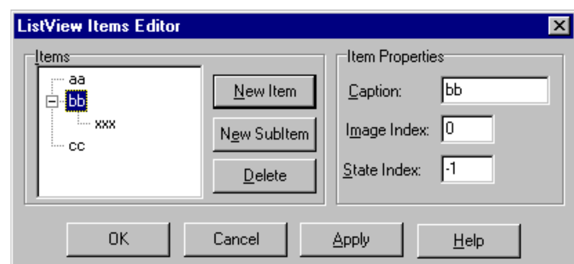


Figure 5.16: The ListView Item Editor. Image from the original book.

In my RefList example (a simple list of references to books, magazines, CD-ROMs, and Web sites) the items are stored to a file, since users of the program can edit the content of the list, which are automatically saved as the program exits. This way, edits made by the user become persistent.

Saving and loading the contents of a ListView is not trivial, since the `TListItems` type doesn't have an automatic mechanism to save the data. As an alternative simple approach, I've copied the data to and from a string list, using a custom format. The string list can then be saved to a file and reloaded with a single command.

The file format is simple, as you can see in the following saving code. For each item of the list, the program saves the caption on one line, the image index on another line (prefixed by the `@` character), and the subitems on the following lines, indented with a tab character:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
  try
    for I := 0 to ListView1.Items.Count - 1 do
    begin
      // save the caption
      List.Add (ListView1.Items[I].Caption);
      // save the index
      List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
      // save the subitems (indented)
      for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
        List.Add (#9 + ListView1.Items[I].SubItems [J]);
    end;
    List.SaveToFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
  finally
    List.Free;
  end;
end;
```

The items are then reloaded in the second part of the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  List: TStringList;
  NewItem: TListItem;
  I: Integer;
begin
  ...
  // load the items
  ListView1.Items.Clear;
  List := TStringList.Create;
  try
    List.LoadFromFile (
      ExtractFilePath (Application.ExeName) + 'Items.txt');
    for I := 0 to List.Count - 1 do
```

```
      if List [I][1] = #9 then
        NewItem.SubItems.Add (Trim (List [I]))
      else if List [I][1] = '@' then
        NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
      else
      begin
        // a new item
        NewItem := ListView1.Items.Add;
        NewItem.Caption := List [I];
      end;
  finally
    List.Free;
  end;
end;
```

The program has a menu you can use to choose one of the different views supported by the ListView control, and to add check boxes to the items, as in a CheckListBox. You can see some of the various combinations of these styles in Figure 5.17[164].

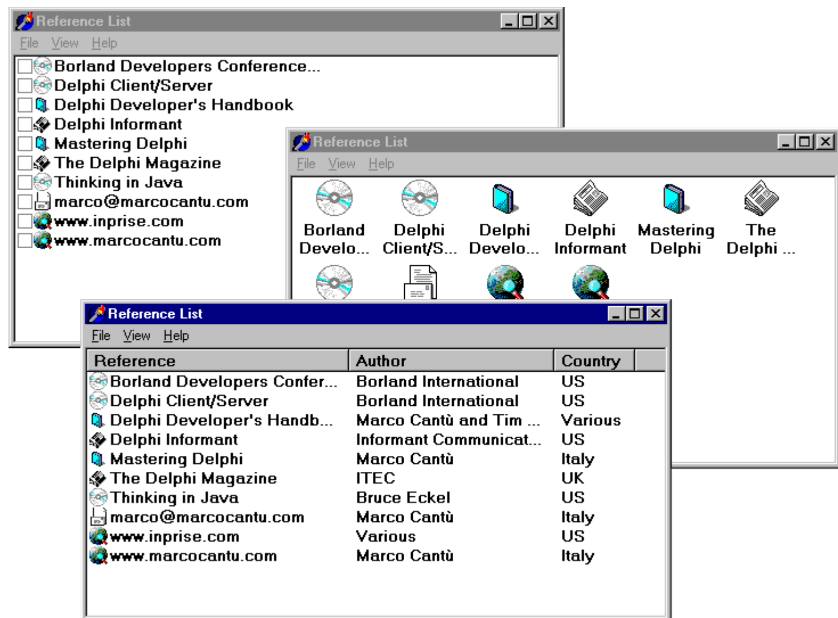


Figure 5.17: Different examples of the output of a ListView component of the RefList program, obtained by changing the ViewStyle property and adding the check boxes. Image from the original book.

Another important feature, which is common in the detailed or report view of the control, is to let a user sort the items on one of the columns. To accomplish this

---

164 The content of this demo, that is the list of books and magazine, it really a blast form the past. You can notice even the company web site, inprise.com!

requires three operations. The first is to set the `SortType` property of the ListView to `stBoth` or `stData`. In this way, the ListView will operate the sorting not based on the captions, but calling the `OnCompare` event for each two items it has to sort. Since we want to do the sorting on each of the columns of the detailed view, we also handle the `OnColumnClick` event (which takes place when the user clicks on the column titles in the detailed view, but only if the `ShowColumnHeaders` property is set to `True`). Each time a column is clicked, the program saves the number of that column in the `nSortCol` private field of the form class:

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

Then, in the third step, the sorting code uses either the caption or one of the subitems according to the current sort column:

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem;
  Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
       Item2.SubItems [nSortCol - 1]);
end;
```

The final features I've added to the program relate to mouse operations. When the user left-clicks an item, the RefList program shows a description of the selected item. Right-clicking the selected item sets it in edit mode, and a user can change it (keep in mind that the changes will automatically be saved when the program terminates). Here is the code for both operations, in the `OnMouseDown` event handler of the ListView control:

```
procedure TForm1.ListView1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
    begin
      // create and show a description
      strDescr := ListView1.Columns [0].Caption + #9 +
```

```
        ListView1.Selected.Caption + #13;
      for I := 1 to ListView1.Selected.SubItems.Count do
        strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
          ListView1.Selected.SubItems [I-1] + #13;
      ShowMessage (strDescr);
    end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
end;
```

Although it is not feature-complete, this example shows some of the potential of the ListView control. I've also activated the Windows 98 "hot-tracking" feature, which lets the list view highlight and underline the item under the mouse, as Figure 5.18 demonstrates. The relevant properties of the ListView can be seen in its textual description:

```
object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
    item
      Caption = 'Country'
      Width = 80
    end>
  Font.Height = -13
  Font.Name = 'MS Sans Serif'
  Font.Style = [fsBold]
  FullDrag = True
  HideSelection = False
  HotTrack = True
  HotTrackStyles = [htHandPoint, htUnderlineHot]
  SortType = stBoth
  ViewStyle = vsList
  OnColumnClick = ListView1ColumnClick
  OnCompare = ListView1Compare
  OnMouseDown = ListView1MouseDown
end
```

This program is actually quite interesting, and I'll further extend it in Chapter 8, adding a dialog box to it.

Figure 5.18: The new hot-tracking feature of the ListView control. Notice that the items are sorted by author. Image from the original book.

A Tree of Data

Now that we've seen an example based on the ListView, we can close the chapter by looking at the TreeView control. The TreeView has a user interface that is flexible and powerful (with support for editing and dragging elements). It is also standard, because it is the user interface of the Windows Explorer. There are a number of properties and various ways to customize the bitmap of each line or of each type of line.

To define the structure of the nodes of the TreeView at design time, you can use the TreeView `Items` property editor (see Figure 5.19). In this case, however, I've decided to load it in the TreeView data at startup, in a way similar to the last example.

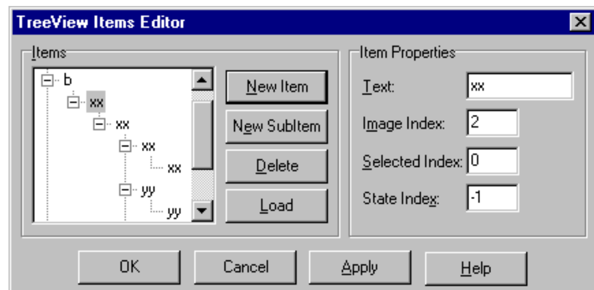


Figure 5.19: The TreeView Items property editor. Image from the original book.

The Items property of the TreeView component has many member functions you can use to alter the hierarchy of strings. For example, we can build a two-level tree with the following lines:

```
var
  Node: TTreeNode;
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

Using these two methods (Add and AddChild) we can build a complex structure at run time. But how do we load the information? Again, you can use a StringList at run time, load a text file with the information, and parse it.

However, since the TreeView control has a LoadFromFile method, the DragTree example uses the following simpler code:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (
    ExtractFilePath (Application.ExeName) + 'TreeText.txt');
end;
```

The LoadFromFile method basically loads the data in a string list and checks the level of each item by looking at the number of tab characters. (If you are curious, see the TTreeStrings.GetBufStart method, which you can find in the ComCtrls unit in the VCL source code included in Delphi.) By the way, the data I've prepared for the TreeView is the organizational chart of a multinational company.

Besides loading the data, the program saves it when it terminates, making the changes persistent. It also has a few menu items to customize the font of the TreeView control and change some other simple settings. The specific feature I've implemented in this example is support for dragging items and entire subtrees. I've set the DragMode property of the component to dmAutomatic and written the event handlers for the OnDragOver and OnDragDrop events.

In the first of the two handlers, the program makes sure the user is not trying to drag an item over a child item (which would be moved along with the item, leading to an infinite recursion):

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
```

```
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and
        (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
  end
  else
    Accept := False;
end;
```

The effect of this code is that (except for the particular case we need to disallow) a user can drag an item of the TreeView over another one, as shown in Figure 5.20. Writing the actual code for moving the items is simple, because the TreeView control provides the support for this operation, through the `MoveTo` method of the `TTreeNode` class:

```
procedure TForm1.TreeView1DragDrop(Sender,
  Source: TObject; X, Y: Integer);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  if TargetNode <> nil then
  begin
    SourceNode := TreeView1.Selected;
    SourceNode.MoveTo (TargetNode, naAddChildFirst);
    TargetNode.Expand (False);
    TreeView1.Selected := TargetNode;
  end;
end;
```

note Among the Demos shipping with Delphi, there is an interesting one showing a custom-draw TreeView control. The example is in the CustomDraw subdirectory.[165]

165 The CustomDraw example is no longer available as part of the official Delphi demos.

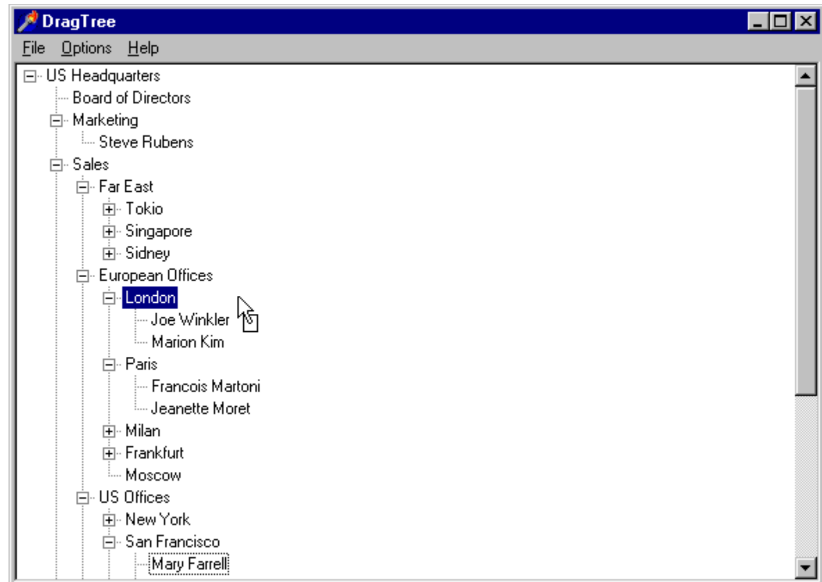


Figure 5.20: The DragTree example during a dragging operation

What's Next?

In this chapter, we have started to explore some of the basic components available in Delphi. These components correspond to the standard Windows controls and some of the Win32 common controls, and they are extremely common in applications. We have also seen how to create main menus and pop-up menus, and we've seen how to add extra graphics to some of these controls.

We also explored a very important and still little-used component, the ActionList, and its architecture for handling menu-item and toolbar-button events. We'll get back to this topic in other examples, and we'll cover the standard MDI and dataset actions in the related chapters.

The next step, however, is to explore in depth one of the most common elements of Delphi programming: forms. We've already used forms many times, but there are still plenty of new features to discuss, including some quite important ones.

Chapter 6: Forms, Windows, And Applications

If you've read the previous chapter, you should now be able to use Delphi's standard components in your applications. So let's turn our attention to the central element of development in Delphi: the form. We have used forms since the initial chapters, but I've never described in detail what you can do with a form, which properties you can use, or which methods of the `TForm` class are particularly interesting.

This chapter looks at some of the properties and styles of forms and at sizing and positioning them. I'll also introduce applications with multiple forms and cover the global VCL objects that handle the interaction among them, `Screen` and `Application`. I'll also devote some time to input on a form, both from the keyboard and the mouse. Let me start this chapter with a general, theoretical discussion of forms and windows.

Forms versus Windows

In Windows, most elements of the user interface are windows. For this reason, in Delphi most components are also based on windows—most of them, but not all. Of course, this is not what a user perceives. The distinction is not obvious, so you should consider the following definitions carefully. Then we can make some further observations.

- From a user standpoint, a window is a portion of the screen surrounded by a border, having a caption and usually a system menu, that can be moved on the screen, closed, and at times also minimized and maximized. Windows can be moved on the screen or inside other windows, as in MDI (Multiple Document Interface) applications. These user windows can be divided into two general categories: main windows and dialog boxes.
- Technically speaking, a window is an entry in an internal memory area of the Windows system, often corresponding to an element visible on the screen, that has some associated code. One of the Windows system libraries contains a list of all the windows that have been built by every application and assigns to each of them a unique number (usually known as a *handle*). Some of these windows are perceived as such by users (see the first definition above), others have the role of controls or visual components, others are temporarily created by the system (for example, to show a pull-down menu), and still others are created by the application but remain hidden from the user.

The common denominator of all windows is that they are known by the Windows system and refer to a function for their behavior; each time something happens in the system, a notification message is sent to the proper window, which responds by executing some code. Each window of the system, in fact, has an associated function (generally called its *window procedure*), which handles the various messages the window is interested in.

In a Delphi application, the system converts these lower-level messages into events. But at times, as we have already seen in some examples, we handle low-level messages directly in a form. Delphi allows us to work at a higher level than the system, making application development much easier.

note The memory area of the Windows system allocated to listing all the windows that have been built is limited. Building too many windows reduces the so-called *system resources*. Windows 3.1 had a severe limit to the number of windows available in the system. In Windows 95 and 98, this limit has been greatly enlarged, and in Windows NT it doesn't even exist. Once there are too many windows in the system (including all the controls and hidden windows), you cannot create even one more window, something that will block most applications. This is why, in Delphi, there are a number of non-windowed components, including labels. This approach lets you save a lot of this system memory without having to worry (or even know) about it. As already mentioned in Chapter 4, graphical non-windowed controls have also other advantages, including faster creation and redraw and less overhead overall[166].

With these general definitions in mind, we can now move back to Delphi and try to understand the role of forms. Forms represent windows from a user standpoint and can be used to build main windows, MDI windows, and dialog boxes. Their behavior is defined mostly by the code written for them but also by a couple of very important properties, `FormStyle` and `BorderStyle`, which we'll explore shortly. Many other components are based on windows, but only forms appear to be windows from a user's point of view. The other windowed components, or controls, can be considered windows only according to the technical definition.

As an example, simply create a new application and place a button in it, save the files in a directory with default names, and run the program. Using the WinSight tool supplied with Delphi[167], you can see the list of the windows of the system; notice in particular the windows created by the application, as shown in Figure 6.1. These include the following windows:

- A main window, the form, with the title *Form1*. It is an overlapped window of class `TForm1`.
- A child window, the button inside the form, with the title *Button1*. This is a child window of class `TButton`.
- A hidden main window, the application window, entitled *Project1*. This is a pop-up window of class `TApplication`.

Notice that the names in brackets in WinSight, which are internal names of the system, correspond to the names of the classes of the Delphi components.

166 If if the Windows handle limit is not relevant today, using graphical controls with no handle still offers advantages, for controls with limited user interaction and not mapped to platform controls.

167 WinSight is not available as part of the product, and I haven't been able to find it online. A newer, similar tools is Spy++ by Microsoft.

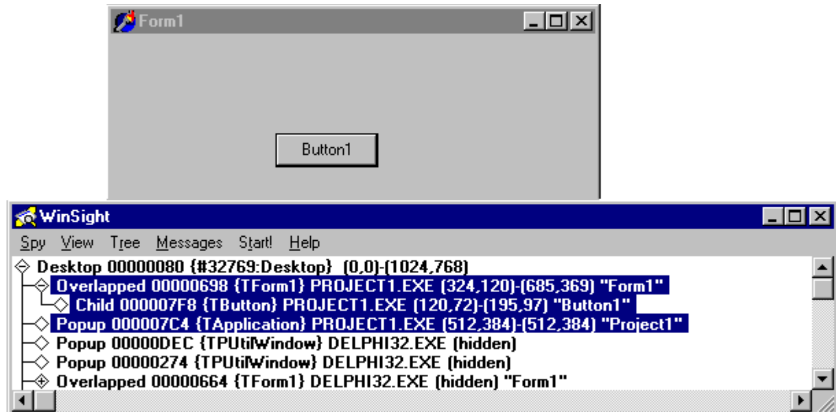


Figure 6.1: The windows of a simple application as they appear in WinSight. Image from the original book.

Overlapped, Pop-Up, and Child Windows

To understand the role of the various windows of this program, we need to look at some technical elements related to the Windows environment. These are not simple concepts, but they are worth knowing about.

1. Each window you create has one of three general styles that determine its behavior. These styles are overlapped, pop-up, and child:
Overlapped windows are main windows of the application, which behave as you would probably expect.

2. *Pop-up* windows are often used for dialog boxes and message boxes and can be considered a holdover from older versions of the system. In fact, in Windows 1, the windows were not overlapped but tiled, and only the pop-up windows could cover other windows. Pop-up windows are generally very similar to overlapped windows.

3. The third group, *child* windows, was originally used for controls inside a dialog box. You can use this style for any window that cannot move outside the client area of the parent window. The obvious extension is to use child windows to build MDI applications; but this behavior is not automatic.

It is important to note that, technically speaking, only child windows can have a parent. Any other window, however, can have an owner. An *owner* is a window that has a continuous message exchange with the windows it owns—for example, when the window is reduced to an icon, when it is activated, and so on. Usually a parent is also the owner, but it forces its child to live inside its client area. The child windows don't use screen coordinates; instead, they use the

client area coordinates of their parent window—to display themselves they borrow pixels not from the screen but from their parent window.

Notice that the Windows API uses the same term (*Parent*) to indicate both the parent and the owner. Even the `GetParent` API function can return both items. Within the system, however, the two handles (that of the parent window and that of the owner window) are stored separately. This is indeed very odd, and it causes a lot of confusion.

In Delphi, all forms are overlapped windows, including dialog boxes, and the form owns all the windowed components (the controls) you place inside it. However, their parent can be either the form or one of the special *container* components, such as the GroupBox or the Panel. When you place a radio button inside a group box, the group box is its parent, but the form is its owner. What about pop-up windows? In Delphi, they are used for the hidden application window, the drop-down list of custom combo boxes, and hint windows. In the system, they are used for message boxes and pop-up or pull-down menus, just to mention two examples.

The `Parent` property of a control indicates what is responsible for displaying it. When you drop a component into a form in the designer, the form will become both parent and owner. When you create the control at run time, you'll need to set the owner (using the Create constructor parameter), but you must also set the `Parent` property, or the control won't be visible.

The Application Is a Window

From the analysis of the WinSight information, you might have noticed that the program has an extra window for the application. Similarly, in the last chapter, we saw that items added to the system menu of the main form were not added to the Taskbar icon, as well. The application window, in fact, is hidden from sight but appears on the Taskbar. This is why Delphi names the window *Form1* and the corresponding Taskbar icon *Project1*[168].

The window related to the `Application` object—the application window—serves to keep together all the windows of an application. The fact that all the top-level forms of a program have this hidden owner window, for example, is fundamental when the

168 There is a a significant change in the VCL pertaining to what its displayed in the task bar. You have the option to use the main form rather than the Application hidden window. This is important today, since the taskbar button offers a form preview and other related features. The default project source code adds by default the line "*Application.MainFormOnTaskbar := True;*" which does what the name implies: It displays the main form rather then the Application handle in the taskbar. When you open an old Delphi application (like those in the original source code of this book), that code is missing and you get the old behavior.

application is activated. In fact, when the windows of your program are behind other windows, clicking on one window in your application will bring all of your application's windows to the front. In other words, the hidden application window is used to connect the different forms of the application. Actually the application window is not hidden, because that would affect its behavior; it simply has zero height and width, and therefore it is not visible.

When you create a new, blank application, Delphi generates a code for the project file, which includes the following[169]:

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

This code uses the global object `Application`, which is of class `TApplication` and is defined by the VCL in the Forms unit. This object is indeed a component, although you cannot set its properties using the Object Inspector. The properties include the name of the executable file (`ExeName`), the `Title` of the application (by default, the name of the executable file without the extension), and the `Icon` displayed in the Taskbar. You can see the application's `Title` in the Windows Taskbar[170]. The same name appears when you scan the running applications with the Alt+Tab keys. To avoid a discrepancy between the two titles, you can change the application's title at design time, in the Application page of the Project Options dialog box. Or at run time, you can copy the form's caption to the title of the application with this code[171]:

```
Application.Title := Form1.Caption;
```

You can also set other properties of the global `Application` object using the same dialog box. To handle the events of the `Application` object, until Delphi 4 you had to write the code manually. Delphi 5, instead, includes the new ApplicationEvents component, specifically intended to handle events of the `Application` object. Beside the easier connection of event handlers at design time, the advantage of using this new component lies in the fact it allows for multiple handlers. If you simply place two instances of the ApplicationEvents component in two different forms, each of them can handle the same event, and both event handlers will be executed. In other words, multiple ApplicationEvents components can chain the handlers.

---

169 As mentioned in the previous note, there is now an extra line, "*Application.MainFormOnTaskbar := True;*".

170 That is, unless you set *MainFormOnTaskbar* to True.

171 This isn't recommended any more, given the better alternative available.

Some of these application-wide events, including OnActivate, OnDeactivate, OnMinimize, and OnRestore, allow you to keep track of the status of the application. Other events are forwarded to the application by the controls receiving them, as OnActionExecute, OnActionUpdate, OnHelp, OnHint, OnShortCut, and OnShowHint. Finally, there is the OnException global exception handler we've used in Chapter 3, the OnIdle event used for background computing and the OnMessage event, which fires whenever a message is posted to any of the windows or windowed controls of the application.

In most applications, you don't care about the application window, apart from setting its Title and icon and handling some of its events. There are some simple operations you can do anyway. Setting the ShowMainForm property to False in the project source code indicates that the main form should not be displayed at startup. Inside a program, instead, you can use the MainForm property of the Application object to access the main form, which is the first form created in the program.

Displaying the Application Window

There is no better proof that there is indeed a window for the Application object than to display it. Actually, we don't need to show it—we just need to resize it and set a couple of window attributes, such as the presence of a caption and a border. We can perform these operations by using Windows API functions on the window indicated by the Handle property of the Application object:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (
    Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle,
    0, 0, 0, 200, 100, swp_NoMove or swp_NoZOrder);
end;
```

The two GetWindowLong and SetWindowLong API functions are used to access the system information related to the window. In this case, we are using the gwl_Style parameter to read or write the styles of the window, which include its border, title, system menu, border icons, and so on. The code above gets the current styles and adds (using an or statement) a standard border and a caption to the form. As we'll

see later in this chapter, you seldom need to use these low-level API functions in Delphi, because there are properties of the `TForm` class that have the same effect. We need this code here because the application window is not a form.

Executing this code displays the project window, as you can see in Figure 6.2[172]. Although there's no need to implement something like this in your own programs, running this program will reveal the relationship between the application window and the main window of a Delphi program. This is a very important starting point if you want to understand the internal structure of Delphi applications.

Figure 6.2: The hidden application window revealed by the ShowApp program. Image from the original book.

note In Windows, the minimize and maximize operations are associated by default with system sounds and a visual animated effect. Applications built with Delphi up to version 4 didn't play the sounds or show the visual effect (unless you write some specific code). Delphi 5 applications, instead, produce the sound and display the visual effect by default. Simply recompile your programs and they'll exhibit this extra feature! Technically, the reason this didn't happen in earlier releases is that the main form's minimize and maximize system messages were not being passed to the default window procedure, where Windows implements system sound behavior, was to avoid an unwanted animation effect in the Taskbar. Having found a fix for this problem in Delphi 5, the default behavior has been restored by passing the messages to the operating system.[173]

---

172 This still happens today with Delphi 12 and Windows 11. If you try it, the application window on screen looks really weird. The point, of course, is just to make you see it exists, it has no role in an actual application.

173 Even if for different reasons, VCL main forms minimize and maximize without using the most common Windows APIs calls for this operations, therefore missing some of the standard effects. I don't think this is a significant issue.

The Application System Menu

Unless you write a very odd program like the example we've just looked at, users will only see the application window in the Taskbar. There, they can activate the window's system menu by right-clicking on it. As I mentioned, when discussing the system menu in the last chapter, an application's menu is not the same as that of the main form. In the SysMenu example in Chapter 5, I added custom items to the system menu of the main form. Now in the SysMenu2 example, I want to customize the system menu of the application window in the Taskbar.

First we have to add the new items to the system menu of the application window when the program starts. Here is the updated code of the `FormCreate` method:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // add a separator and a menu item to the system menu
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');
  // add the same items to the application system menu
  AppendMenu (GetSystemMenu (Application.Handle, FALSE),
    MF_SEPARATOR, 0, '');
  AppendMenu (GetSystemMenu (Application.Handle, FALSE),
    MF_STRING, idSysAbout, '&About...');
end;
```

The first part of the code adds the new separator and item to the system menu of the main form. The other two calls add the same two items to the application's system menu, simply by referring to `Application.Handle`. This is enough to display the updated system menu, as you can see by running this program. The next step is to handle the selection of the new menu item.

To handle form messages, we can simply write new event handlers or message-handling methods. We cannot do the same with the application window, simply because inheriting from the `TApplication` class is quite a complex issue. Most of the time we can simply handle the `OnMessage` event of this class, which is activated for every message the application retrieves from the message queue.

To handle the `OnMessage` event of the global `Application` object, simply add an ApplicationEvents component to the main form, and define a handler for the `OnMessage` event of this component. In this case, we simply need to handle the `wm_SysCommand` message, and we only need to do that if the `wParam` parameter indicates that the user has selected the menu item we've just added, `idSysAbout`:

```
procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
```

```
  var Handled: Boolean);
begin
  if (Msg.Message = wm_SysCommand) and
    (Msg.wParam = idSysAbout) then
  begin
    ShowMessage ('Mastering Delphi: SysMenu2 example');
    Handled := True;
  end;
end;
```

This method is very similar to the one used to handle the corresponding system menu item of the main form:

```
procedure WMSysCommand (var Msg: TWMSysCommand);
  message wm_SysCommand;
...
procedure TForm1.WMSysCommand (var Msg: TWMSysCommand);
begin
  // handle a specific command
  if Msg.CmdType = idSysAbout then
    ShowMessage ('Mastering Delphi: SysMenu2 example');
  inherited;
end;
```

Activating Applications and Forms

To show how the activation of forms and applications works, I've written a simple, self-explanatory example called ActivApp. This example has two forms. Each form has a Label component (`LabelForm`) used to display the status of the form. The program uses text and color for this, as the handlers of the `OnActivate` and `OnDeactivate` events of the first form demonstrate:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Active';
  LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
  LabelForm.Caption := 'Form2 Not Active';
  LabelForm.Color := clBtnFace;
end;
```

The second form has a similar label and similar code. The main form also displays the status of the entire application. It uses an ApplicationEvents component to handle the `OnActivate` and `OnDeactivate` events of the Application object. These two

event handlers are similar to the two listed previously, with the only difference being that they modify the text and color of a second label of the form.

If you try running this program, you'll see whether this application is the active one and, if so, which of its forms is the active one. By looking at the output (see Figure 6.3) and listening for the beep, you can understand how each of the activation events is triggered by Delphi. Run this program and play with it for a while to understand how it works. We'll get back to other events related to the activation of forms in a while.

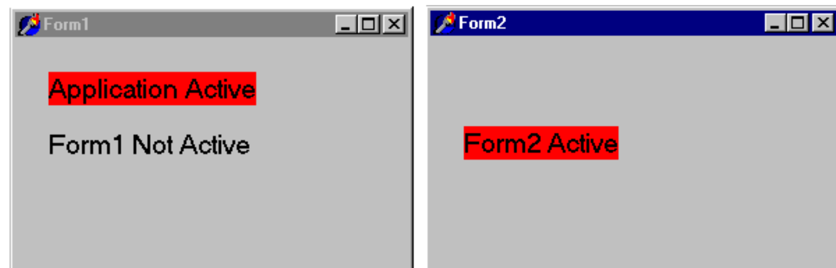


Figure 6.3: The ActivApp example shows whether the application is active and which of the application's forms is active. Image from the original book.

Setting Form and Border Styles

Among the properties of a form, two of them determine the fundamental rules of its behavior: `FormStyle` and `BorderStyle`. The first of these two special properties allows you to choose between a normal SDI (Single Document Interface) and one of the windows that make up an MDI (Multiple Document Interface) application[174].

These are the possible values of the `FormStyle` property:

- `fsNormal`: The form is a normal SDI window or a dialog box.
- `fsMDIChild`: The form is an MDI child window.
- `fsMDIForm`: The form is an MDI parent window—that is, the frame window of the MDI application.

174 MDI has long been deprecated by Microsoft, but some VCL developers still use it. That's why Delphi in version 12 overhauled MDI with quality and features improvements .

- `fsStayOnTop`: The form is an SDI window, but it always remains on top of all other windows except for any that also happen to be *stay-on-top* windows.

Because an application that conforms to the Multiple Document Interface standard needs windows of two different kinds (frame and child), two values of the `FormStyle` property are involved. To build an MDI application, you can use the standard application template or look at Chapter 8, which focuses on the MDI in detail. For now, though, it might be interesting to explore the use of the `fsStayOnTop` style.

To create a topmost form (a form whose window is always on top), you need only set the `FormStyle` property, as indicated above. This property has two different effects, depending on the kind of form you apply it to:

- The main form of an application will remain in front of every other application (unless other applications have the same topmost style, too).
- A secondary form will remain in front of any other form of the application it belongs to. The windows of other applications are not affected, though.

The Border Style

The second property of a form is `BorderStyle`. This property refers to a visual element of the form, but it has a much more profound influence on the behavior of the window, as you can see in Figure 6.4.

At design time, the form is always shown using the default value of the `BorderStyle` property, `bsSizeable`. This corresponds to a Windows style known as *thick frame*. When a main window has a thick frame around it, a user can resize it by dragging its border. This is made clear by the special *resize* cursors (with the shape of a double-pointer arrow) displayed when the user moves the mouse onto this thick window border.[175]

175 The user interface of windows borders has changed a lot in Windows over recent editions. While the technical foundations described here are still the same, the UI and behavior are significantly different.

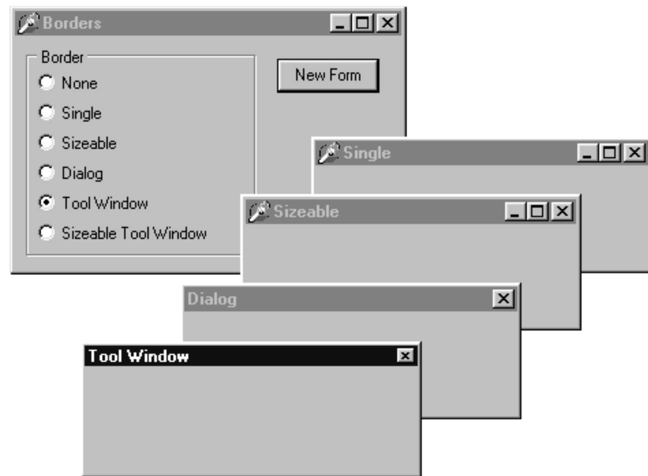


Figure 6.4: Sample forms with the different border styles, created by the Borders example. Image from the original book.

A second important choice for this property is `bsDialog`. If you select it, the form uses as its border the typical dialog-box frame—a thick frame that doesn't allow resizing. In addition to this graphical element, note that if you select the `bsDialog` value, the form becomes a dialog box. This involves a number of changes. For example, the items on its system menu are different, and the form will ignore some of the elements of the `BorderIcons` set property.

note Setting the `BorderStyle` property at design time produces no visible effect. In fact, several component properties do not take effect at design time, because they would prevent you from working on the component while developing the program. For example, how could you resize the form with the mouse if it were turned into a dialog box? When you run the application, though, the form will have the border you requested.

There are four more values we can assign to the `BorderStyle` property. The style `bsSingle` can be used to create a main window that's not resizable. Many games and applications based on windows with controls (such as data-entry forms) use this value, simply because resizing these forms makes no sense. Enlarging a form to see an empty area or reducing its size to make some components less visible often doesn't help a program's user (although Delphi's automatic scroll bars partially solve the last problem). The value `fsNone` is used only in very special situations and inside other forms. You'll never see an application with a main window that has no border or caption (except maybe as an example in a programming book to show you that it makes no sense).

The last two values, bsToolWindow and bsSizeToolWin, are related to the specific Win32 extended style ws_ex_ToolWindow. This style turns the window into a floating toolbox, with a small title font and close button. This style should not be used for the main window of an application.

To test the effect and behavior of the different values of the BorderStyle property, I've written a simple program called Borders. You've already seen its output, in Figure 6.4. However, I suggest you run this example and experiment with it for a while to understand all the differences in the forms.

The main form of this program contains only a radio group and a button. There is also a secondary form, with no components and the Position property set to poDefaultPosOnly. This affects the initial position of the secondary form we'll create by pressing the button. (I'll discuss the Position property later in this chapter.)

The code of the program is very simple. When you press the button, a new form is dynamically created, depending on the selected item of the radio group:

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
  NewForm: TForm2;
begin
  NewForm := TForm2.Create (Application);
  NewForm.BorderStyle := TFormBorderStyle (
    BorderRadioGroup.ItemIndex);
  NewForm.Caption := BorderRadioGroup.Items[
    BorderRadioGroup.ItemIndex];
  NewForm.Show;
end;
```

This code actually uses a trick: it casts the number of the selected item into the TFormBorderStyle enumeration. This works because I've given the radio buttons the same order as the values of this enumeration:

```
type
  TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
    bsDialog, bsToolWindow, bsSizeToolWin);
```

The BtnNewFormClick method then copies the text of the radio button to the caption of the secondary form. This program refers to TForm2, the secondary form defined in a secondary unit of the program, saved as SECOND.PAS. For this reason, to compile the example, you must add the following lines to the implementation section of the unit of the main form:

```
uses
  Second;
```

note Whenever you need to refer to another unit of a program, place the corresponding `uses` statement in the `implementation` portion instead of the `interface` portion if possible. This speeds up the compilation process, results in cleaner code (because the units you include are separate from those included by Delphi), and never generates circular references between different units. To accomplish this, you can also use the File ➢ Use Unit menu command.

The Border Icons

Another important element of a form is the presence of icons on its border[176]. By default, a window has a small icon connected to the system menu, a Minimize button, a Maximize button, and a Close button on the far right. You can set different options using the `BorderIcons` property, a set with four possible values: `biSystemMenu`, `biMinimize`, `biMaximize`, and `biHelp`.

note The `biHelp` border icon enables the "What's this?" Help. When this style is included and the `biMinimize` and `biMaximize` styles are excluded, a question mark appears in the form's title bar. If you click on this question mark and then click on a component inside the form (but not on the form itself!), Delphi activates the Help about that object inside a pop-up window. This is demonstrated by the BIcons example, which has a simple Help file with a page connected to the `HelpContext` property of the button in the middle of the form.

The BIcons example demonstrates the behavior of a form with different border icons and shows how to change this property at run time. The form of this example is very simple: it has only a menu, with a pull-down containing four menu items, one for each of the possible elements of the set of border icons. I've written a single method, connected with the four commands, that reads the check marks on the menu items to determine the value of the `BorderIcons` property. This code is therefore also a good exercise in working with sets:

```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
begin
  (Sender as TMenuItem).Checked :=
    not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
```

176 The ability of customizing the border has been largely expanded over time. The VCL library includes a TitleBarPanel component offering complete control on the title bar, as you can place other controls on it (effectively displaying the in the title bar). This is the component the Delphi IDE also uses to host buttons, a combo boxe, and a search box it its title bar.

```
    else
      BorIco := [];
    if MaximizeBox1.Checked then
      Include (BorIco, biMaximize);
    if MinimizeBox1.Checked then
      Include (BorIco, biMinimize);
    if Help1.Checked then
      Include (BorIco, biHelp);
    BorderIcons := BorIco;
  end;
```

While running the BIcons example, you can easily set and remove the various visual elements of the form's border. You'll immediately see that some of these elements are closely related: if you remove the system menu, all of the border icons will disappear; if you remove either the Minimize or the Maximize button, it will be grayed; if you remove both these buttons, they will disappear. Notice also that in these last two cases, the corresponding items of the system menu are automatically disabled. This is the standard behavior for any Windows application. When the Maximize and Minimize buttons have been disabled, you can activate the Help button. As a shortcut to obtain this effect, you can press the button inside the form. Also, you can click on the button after pressing the Help Menu icon to see a Help message, as you can see in Figure 6.5.

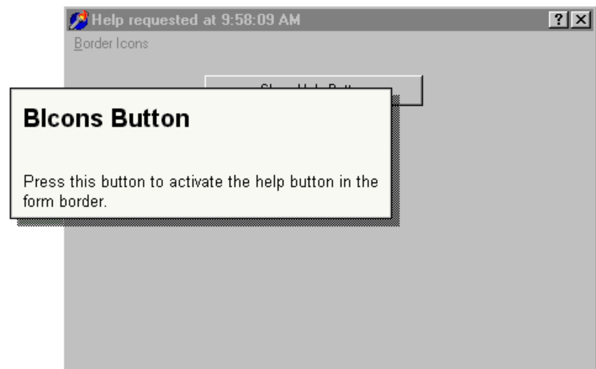


Figure 6.5: The Help button displayed by the BIcons example. By dragging the Help cursor over the button you get the Help displayed in the figure. Image from the original book.

As an extra feature, the program also displays the time that the Help was invoked in the caption, by handling the `OnHelp` event of the form. This effect is visible in the figure.

Setting More Window Styles

The border style and border icons are indicated by two different Delphi properties, which can be used to set the initial value of the corresponding user interface elements. We have seen that besides changing the user interface, these properties affect the behavior of a window. It is important to know that these border-related properties and the `FormStyle` property mainly correspond to different settings in the *style* and *extended style* of a window. These two terms reflect two parameters of the `CreateWindowEx` API function Delphi uses to create forms.

It is important to acknowledge this, because Delphi allows you to modify these two parameters freely by overriding the `CreateParams` virtual method:

```
public
  procedure CreateParams (
    var Params: TCreateParams); override;
```

This is the only way to use some of the peculiar window styles that are not directly available through form properties. For a list of window styles and extended styles, see the API Help under the topics *CreateWindow* and *CreateWindowEx*. You'll notice that the Win32 API has a number of styles for these functions, including those related to tool windows.

To show how to use this approach, I've written the NoTitle example, which lets you create a program with a custom caption. First we have to remove the standard caption but keep the resizing frame by setting the corresponding styles:

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
  inherited CreateParams (Params);
  Params.Style := (Params.Style or ws_Popup) and
    not ws_Caption;
end;
```

note Besides changing the style and other features of a window when it is created, you can change them at run time, although some of the settings do not take effect. To change most of the creation parameters at run time, you can use the `SetWindowLong` API function, which allows you to change the internal information of a window. The companion `GetWindowLong` function can be used to read the current status. Two more functions, `GetClassLong` and `SetClassLong`, can be used to read and modify class styles (the information of the `WindowClass` structure of `TCreateParams`). You'll seldom need to use these low-level Windows API functions in Delphi, unless you write advanced components.

To remove the caption, we need to change the overlapped style to a pop-up style; otherwise, the caption will simply stick. Now how do we add a custom caption? I've

placed a label aligned to the upper border of the form and a small button on the far end. You can see this effect at run time in Figure 6.6.

Figure 6.6: The NoTitle example has no real caption but a fake one made with a label. Image from the original book.

To make the fake caption work, we have to tell the system that a mouse operation on this area corresponds to a mouse operation on the caption. This can be done by intercepting the `wm_NCHitTest` Windows message, which is frequently sent to Windows to determine where the mouse currently is. When the hit is in the client area and on the label, we can pretend the mouse is on the caption by setting the proper result:

```
procedure TForm1.HitTest (var Msg: TWmNCHitTest);
  // message wm_NCHitTest
begin
  inherited;
  if (Msg.Result = htClient) and (Msg.YPos <
      Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then
    Msg.Result := htCaption;
end;
```

The `GetSystemMetrics` API function used in the listing above is used to query the operating system about the size of the various visual elements[177]. It is important to make this request every time (and not cache the result) because users can customize most of these elements by using the Appearance tab of the Desktop options (in Control Panel) and other Windows settings. The small button, instead, has a call to the `Close` method in its `OnClick` event handler. The button is kept in its position even when the window is resized by using the `[akTop,akRight]` value for the `Anchors`

177 These days the VCL intercepts the *GetSystemMetrics* API to make is DPI aware.

property. The form also has size constraints, so that a user cannot make it too small, as described in the "Form Constraints" section later in this chapter.

Scaling Forms

When you create a form with a number of components, it is common to make the form nonresizable to avoid having some of the components fall outside the visible portions of the form. This is not a big problem, because Delphi automatically adds scroll bars to the form so you can reach every control easily. (Form scrolling is one of the subjects of Chapter 7.)

Be aware of this problem when you create a big form: if you build a form on a high-resolution screen, it might be bigger than the available screen size on your end-user's systems. This is a pity, and it is more common that you might expect. If you can, never build a form larger than 640 x 480 pixels[178].

If you have to build a bigger form and using scroll bars is not a solution, Delphi has some nice scaling features. There are two basic techniques:

- The form's `ScaleBy` method allows you to scale the form and each of its components. You can use this method at startup, after you've determined the screen resolution, or in response to a specific request by the user.
- The `PixelsPerInch` and `Scaled` properties allow Delphi to resize an application automatically when the application is run with a different system font size, often because of a different screen resolution[179]. Of course, you can change the values of these properties manually, as described in the next section, and let the system scale the form only when you want.

note Form scaling is calculated based on the difference between the font height at run time and the font height at design time. Scaling ensures that edit and other controls are large enough to display their text using the user's font preferences without clipping the text. The form scales as well, as we will see later on, but the main point is to make edit and other controls readable.

178 This size makes little sense with today's screens. Also this entire sections ignores the issues with HiDPI Windows configurations, system scaling, and all of the options that have been dded to the operating system and the VCL to handle these scenarios. I won't call out these differences for each of the references in this section that are subject to these changes, or I could add a footnote for almost each line!

179 This is mostly true for HiDPI configurations.

In both cases, to make the form scale its window, be sure to also set the `AutoScroll` property to `False`. Otherwise, the contents of the form will be scaled, but the form border itself will not.

Manual Form Scaling

Any time you want to scale a form, including its components, you can use the `ScaleBy` method, which has two integer parameters, a multiplier and a divisor—it's a fraction. You can apply the same method to a single component. For example, with this statement

```
ScaleBy (3, 4);
```

the size of the current form is divided by 4 and multiplied by 3; that is, the form is reduced to three-quarters of its original size. Generally, it is easier to use percentage values. The same statement can be written this way:

```
ScaleBy (75, 100);
```

When you scale a form, all the proportions are maintained, but if you go below or above certain limits, the text strings can alter their proportions slightly. If you reduce the size of a form too much, most of the components will become unusable or even disappear completely. The problem is that in Windows, components can be placed and sized only in whole pixels, while scaling almost always involves multiplying by fractional numbers. So any fractional portion of a component's origin or size will be truncated.

To avoid similar problems, you should let the user perform only a limited number of scaling operations or re-create the form from scratch before each new scaling so that round-off errors do not accumulate.

---

note If you apply the `ScaleBy` method to a form, the form won't actually be scaled. Only the components inside the form will change their size. As I mentioned before, to overcome this problem, you should disable the form's `AutoScroll` property. What is the relationship between scaling and scrolling? My guess is that if scrolling is enabled, the component can be moved outside the form's visible area without many problems; otherwise, the form is resized, too.

I've built a simple example, Scale, to show how you can scale a form manually, responding to a request by the user. The form of this application (see Figure 6.7) has two buttons, a label, an edit box, and an UpDown control.

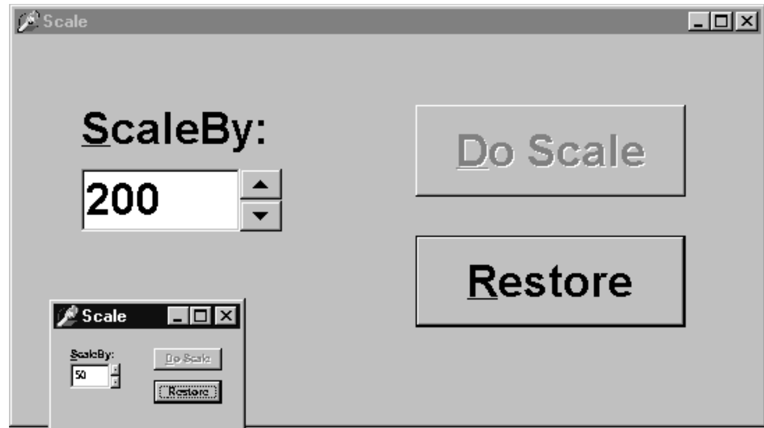


Figure 6.7: The form of the Scale example after a scaling with 50 and 200. Image from the original book.

The UpDown component connects to the edit box, using its `Associate` property. With this setting, a user can type numbers in the box or click on the two small arrows to increase or decrease the number in the edit box by a fixed amount (indicated by the `Increment` property of the UpDown component). To extract the input value, you can use the `Text` property of the edit box or the `Position` of the UpDown control. You can also prevent input errors by setting the Min and Max properties of the UpDown, as I've done in this example:

```
object UpDown1: TUpDown
  Associate = Edit1
  Min = 30
  Max = 300
  Increment = 10
  Position = 100
  Wrap = False
end
```

When you press the ScaleButton button, the current input value is used to determine the scaling percentage of the form:

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
  AmountScaled := UpDown1.Position;
  ScaleBy (AmountScaled, 100);
  UpDown1.Height := Edit1.Height;
  ScaleButton.Enabled := False;
  RestoreButton.Enabled := True;
end;
```

This method stores the current input value in the form's `AmountScaled` private field and enables the Restore button, disabling the one that was pressed. Later, when the

user presses the Restore button, the opposite scaling takes place, by calling `ScaleBy (100, AmountScaled)`. In both cases, I've added a line of code to set the `Height` of the UpDown component to the same `Height` as the edit box it is attached to. This prevents small differences between the two.

note If you want to scale the text of the form properly, including the captions of components, the items in list boxes, and so on, you should use TrueType fonts exclusively. The system font (MS Sans Serif) doesn't scale well. The font issue is important because the size of many components depends on the text height of their captions, and if the caption does not scale well, the component might not work properly. For this reason, in the Scale example I've used an Arial font.[180]

Automatic Form Scaling

Instead of playing with the `ScaleBy` method, you can ask Delphi to do the work for you. When Delphi starts, it asks the system for the display configuration and stores the value in the `PixelsPerInch` property of the `Screen` object, a special global object of the VCL, available in any application.

`PixelsPerInch` sounds like it has something to do with the pixel resolution of the screen, but unfortunately, it doesn't. If you change your screen resolution from 640 x 480 to 800 x 600 to 1024 x 768 or even 1600 x 1280[181], you will find that Windows reports the same `PixelsPerInch` value in all cases, unless you change the system font. What `PixelsPerInch` really refers to is the screen pixel resolution that the currently installed system font was designed for. When the end user changes the system font scale, usually to make menus and other text easier to read, the user will expect all applications to honor those settings. An application that does not reflect the user desktop preferences will look out of place and, in extreme cases, may be unusable to visually impaired users who rely on very large fonts and high-contrast color schemes.

The most common `PixelPerInch` values are 96 (small fonts) and 120 (large fonts), but other values are possible. Windows 98 even allows the user to set the system font size to an arbitrary scale[182]. At design time, the `PixelsPerInch` value of the screen, which is a read-only property, is copied to every form of the application.

---

180 I'd say this is now hardly the case any more, as the Windows OS had many improvements in this area.

181 Which is still very small compared to a 4K monitor… again, some of the core concepts still apply, but a lot has changed in Windows and in the VCL in this area.

182 This has now become a very commonly used feature, with many user setting their display at 150% or 200% scalcing.

Delphi then uses the value of `PixelsPerInch`, if the `Scaled` property is set to `True`, to resize the form when the application starts.

As I've already mentioned, both automatic scaling and the scaling performed by the `ScaleBy` method operate on components by changing the size of the font. The size of each control, in fact, depends on the font it uses. With automatic scaling, the value of the form's `PixelsPerInch` property (the design-time value) is compared to the current system value (indicated by the corresponding property of the `Screen` object), and the result is used to change the font of the components on the form. Actually, to improve the accuracy of this code, the final height of the text is compared to the design-time height of the text, and its size is adjusted if they do not match.

Thanks to Delphi automatic support, the same application running on a system with a different system font size automatically scales itself, without any specific code. The application's edit controls will be the correct size to display their text in the user's preferred font size, and the form will be the correct size to contain those controls. Although automatic scaling has problems in some special cases, if you comply with the following rules, you should get good results[183]:

- Set the `Scaled` property of forms to `True`. (This is the default.)
- Use only TrueType fonts.
- Use Windows small fonts (96dpi) on the computer you use to develop the forms.
- Set the `AutoScroll` property to `False`, if you want to scale the form and not just the controls inside it. (AutoScroll defaults to True, so don't forget to do this step.)
- Set the form position either near the upper-left corner or in the center of the screen (with the `poScreenCenter` value) to avoid having an out-of-screen form. Form position is discussed in the next section.

Setting the Form's Position and Size

In addition to `PixelsPerInch`, there are more run-time properties you can set to control the appearance of a form. The `Position` property indicates the initial position of the form on the screen when it is first created. The default `poDesigned` value

183 Add to this using PerMonivotrv2 configuration as a must have, along with enabling themes. There is a lot more to be said, but covering modern HiDPI and proper forms and controls scaling and positioning will require an entire chapter, not a footnote.

indicates that the form will appear where you designed it and use the positional and size properties of the form. Some of its other choices (`poDefault`, `poDefaultPosOnly`, and `poDefaultSizeOnly`) depend on a feature of the system: using a specific flag, Windows can position and/or size new windows using a cascade layout. Finally, with the `poScreenCenter` value, the form is displayed in the center of the screen, with the size you set at design time.

note The default positions are ignored when the form has a dialog border style.

The second parameter that affects the initial size and position of a window is its *state*. You can use the `WindowState` property at design time to display a maximized or minimized window at startup. This property, in fact, can have only three values: `wsNormal`, `wsMinimized`, and `wsMaximized`. The meaning of this property is intuitive. If you set a minimized window state, it will be properly displayed in the Windows Taskbar.

Of course, you can maximize or minimize a window at run time, too. Simply changing the value of the `WindowState` property to `wsMaximized` or to `wsNormal` produces the expected effect. Setting the property to `wsMinimized`, however, creates a minimized window that is placed over the Taskbar, not within it. This is not the expected action for a main form, but that for a secondary form! The simple solution to this problem is to call the `Minimize` method of the `Application` object. There is also a `Restore` method in the `TApplication` class that you can use when you need to restore a form, although most often the user will do this using the Restore command of the system menu.

The Size of a Form and Its Client Area

At design time, there are two ways to set the size of a form: by setting the value of the `Width` and `Height` properties or by dragging its borders. At run time, if the form has a resizable border, the user can resize it (producing the `OnResize` event).

However, if you look at a form's properties in source code or in the online Help, you can see that there are two properties referring to its width and two referring to its height. `Height` and `Width` refer to the size of the form, including the borders; `ClientHeight` and `ClientWidth` refer to the size of the internal area of the form, excluding the borders, the caption, scroll bars (if any), and the menu bar. The client area of the form is the surface you can use to place components on the form, to create output, and to receive user input.

Since you might be interested in having a certain available area, at times it makes sense to set the client size of a form instead of its global size. This is straightforward, because as you set one of the two client properties, the corresponding form property changes accordingly. When you modify the value of `ClientHeight`, the value of `Height` immediately changes.

note In Windows, it is also possible to create output and receive input from the nonclient area of the form—that is, its border. Painting on the border and getting input when you click on it are complex issues. If you are interested, look in the Help file at the description of such Windows messages as `wm_NCPaint`, `wm_NCCalcSize`, and `wm_NCHitTest` and the series of nonclient messages related to the mouse input, such as `wm_NCLButtonDown`. The difficulty of this approach is in combining your code with the default Windows behavior. However, Delphi lets you process these low-level Windows messages without any problem, something that most visual programming environments do not allow at all.

Form Constraints

When you choose a re-sizable border for a form, users can generally resize the form as they like and also maximize it to full screen. Windows informs you that the form's size has changed with the `wm_Size` message, which generates the `OnResize` event. `OnResize` takes place after the size of the form has already been changed. Modifying the size again in this event (if the user has reduced or enlarged the form too much) would be silly. A preventive approach is better suited to this problem.

Delphi provides a specific property for forms and also for all controls: the `Constraints` property. Simply setting the sub-properties of the `Constraints` property to the proper maximum and minimum values creates a form that cannot be resized beyond those limits. Here is an example:

```
object Form1: TForm1
  Width = 242
  Height = 162
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

Notice that as you set up the `Constraints` property, it has an immediate effect even at design time, changing the size of the form if it is outside the permitted area.

Delphi also uses the maximum constraints for maximized windows, producing an awkward effect. For this reason, you should generally disable the Maximize button

of a window that has a maximum size. There are cases in which maximized windows with a limited size make sense—this is the behavior of Delphi's main window.

note The `Constraints` property plays an even more important role for controls and for their docking operations, as we'll see in Chapter 7.

In case you need to change constraints at run time, you can also consider using two specific events, `OnCanResize` and `OnContrainedResize`. The first of the two can also be used to disable resizing a form or control in given circumstances.

Creating Forms

Up to now we have ignored the issue of form creation. We know that when the form is created, we receive the `OnCreate` event and can change or test some of the initial form's properties or fields. The statement responsible for creating the form is in this project's source file (or `DPR` file, available through the Project menu command):

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

To skip the automatic form creation, you can either modify this code or use the Forms page of the Project Options dialog box (see Figure 6.8). In this dialog box, you can decide whether the form should be automatically created. If you disable the automatic creation, the project's initialization code becomes the following:

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

If you now run this program, nothing happens. It terminates immediately because no main window is created. So what is the effect of the call to the application's `CreateForm` method? It creates a new instance of the form class passed as the first parameter and assigns it to the variable passed as the second parameter.

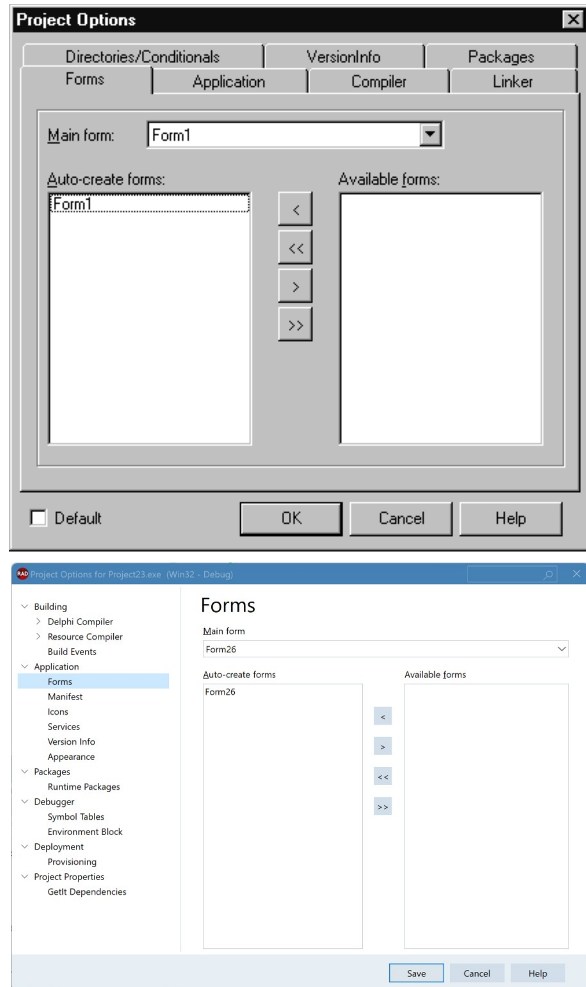


Figure 6.8: The Forms page of the Delphi Project Options dialog box. Images captured in Delphi 5 and Delphi 12.

Something else happens behind the scenes. When `CreateForm` is called, if there is currently no main form, the current form is assigned to the application's `MainForm` property. For this reason, the form indicated as *Main form* in the dialog box shown in Figure 6.8 corresponds to the first call to the application's `CreateForm` method (that is, when several forms are created at start-up).

The same holds for closing the application. Closing the main form terminates the application, regardless of the other forms. If you want to perform this operation from the program's code, simply call the `Close` method of the main form, as we've done several times in past examples.

note In Delphi 5, you can (finally) control the automatic creation of secondary forms by using the Auto Create Forms checkbox on the Preferences page of the Environment Options dialog box[184].

Delphi Form Creation Order

Regardless of the manual or automatic creation of forms, when a form is created, there are many events you can intercept. Form-creation events are fired in the following order:

1. `OnCreate` indicates that the form is being created.
2. `OnShow` indicates that the form is being displayed. Besides main forms, this event happens after you set the `Visible` property of the form to `True` or call the `Show` or `ShowModal` methods. This event is fired again if the form is hidden and then displayed again.
3. `OnActivate` indicates that the form becomes the active form within the application. This event is fired every time you move from another form of the application to the current one, as we saw in the section "Activating Applications and Forms."
4. Other events, including `OnResize` and `OnPaint`, indicate operations always done at start-up but then repeated many times.

As you can see in the list above, every event has a specific role apart from form initialization, except for the `OnCreate` event, which is guaranteed to be called only once as the form is created.

However, there is an alternative approach to adding initialization code to a form: overriding the constructor. This is usually done as follows:

```
constructor TForm1.Create(AOwner: TComponent);
begin
  inherited Create (AOwner);
  // extra initialization code
end;
```

Before the call to the `Create` method of the base class, the properties of the form are still not loaded, and the internal components are not available. For this reason the standard approach is to call the base class constructor first and then do the custom operations.

Now the question is whether these custom operations are executed before or after the `OnCreate` event is fired. The answer depends on the value of the `OldCreateOrder`

184 This is in the Tools | Options dialog box under User Interface | Form Designer.

property of the form, introduced in Delphi 4 for backward compatibility with past versions of Delphi[185]. (This property is part of the Legacy category, which in Delphi 5 is hidden by default.) By default, for a new project, all of the code in the constructor is executed before the `OnCreate` event handler. In fact, this event handler is not activated by the base class constructor but by its `AfterConstruction` method, a sort of constructor introduced for compatibility with C++Builder.

note To study the creation order and the potential problems, you can examine the CreatOrd program. This program has an `OnCreate` event handler, which creates a list box control dynamically. The constructor of the form can access to this list box or not depending on the value of the `OldCreateOder` property.

Tracking Forms with the Screen Object

We have already explored some of the properties and events of the `Application` object. Other interesting global information about an application is available through the `Screen` object, whose base class is `TScreen`. This object holds information about the system display (the screen size and the screen fonts) and also about the current set of forms in a running application. For example, you can display the screen size and the list of fonts by writing:

```
Label1.Caption := IntToStr (Screen.Width) + 'x' +
  IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;
```

`TScreen` also reports the number and resolution of monitors in a multimonitor system. What I want to focus on now, however, is the list of forms held by the `Forms` property of the `Screen` object, the topmost form indicated by the `ActiveForm` property, and the related `OnActiveFormChange` event. Note that the forms the `Screen` object references are the forms of the application and not those of the system.

These features are demonstrated by the Screen example, which maintains a list of the current forms in a list box. This list must be updated each time a new form is created, an existing form is destroyed, or the active form of the program changes. To see how this works, you can create a number of secondary forms by clicking on the button labeled New:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
```

185 This *OldCreateOrder* property was recently removed, after having been deprecated for a very long time. As indicated here, it was added for Delphi 4 migration, and after another 20 versions of the product, the team felt it was time to remove it.

```
var
  NewForm: TSecondForm;
begin
  // create a new form, set its caption, and run it
  NewForm := TSecondForm.Create (Self);
  Inc (nForms);
  NewForm.Caption := 'Second ' + IntToStr (nForms);
  NewForm.Show;
end;
```

One of the key portions of the program is the `OnCreate` event handler of the form, which fills the list a first time and then connects a handler to the `OnActiveFormChange` event:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FillFormsList (Self);
  // set the secondary forms counter to 0
  nForms := 0;
  // set an event handler on the screen object
  Screen.OnActiveFormChange := FillFormsList;
end;
```

The code used to fill the Forms list box is inside a second procedure, `FillFormsList`, which is also installed as an event handler for the `OnActiveFormChange` event of the `Screen` object:

```
procedure TMainForm.FillFormsList (Sender: TObject);
var
  I: Integer;
begin
  FormsLabel.Caption := 'Forms: ' +
    IntToStr (Screen.FormCount);
  FormsListBox.Clear;
  // write class name and form title to the list box
  for I := 0 to Screen.FormCount - 1 do
    FormsListBox.Items.Add (Screen.Forms[I].ClassName +
      ' - ' + Screen.Forms[I].Caption);
  ActiveLabel.Caption := 'Active Form : ' +
    Screen.ActiveForm.Caption;
end;
```

note It is very important that you remove the `OnActiveFormChange` event handler before exiting the application; that is, before the main form is destroyed. Otherwise, the code will be executed when no list box exists, and you'll get a system error. The solution is to handle the `OnClose` event of the main form and assign `nil` to `Screen.OnActiveFormChange`.

The `FillFormsList` method fills the list box and sets a value for the two labels above it to show the number of forms and the name of the active one. When you click on

the New button, the program creates an instance of the secondary form, gives it a new title, and displays it. The Forms list box is updated automatically because of the handler we have installed for the `OnActiveFormChange` event. Figure 6.9 shows the output of this program when some secondary windows have been created.

note The program always updates the text of the `ActiveLabel` above the list box to show the currently active form, which is always the same as the first one in the list box.

The secondary forms each have a Close button you can select to remove them. The program handles the `OnClose` event, setting the `Action` parameter to `caFree`, so that the form is actually destroyed when it is closed. This code closes the form, but it doesn't update the list of the windows properly. The system moves the focus to another window first, firing the event that updates the list, and destroys the old form only after this operation.

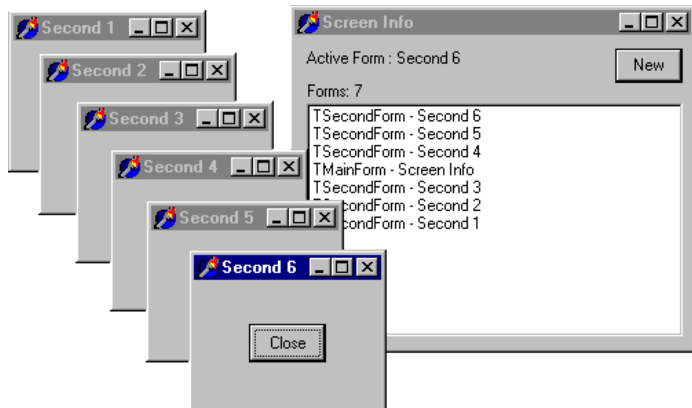


Figure 6.9: The output of the Screen example with a number of secondary forms. Image from the original book.

The first idea I had to update the windows list properly is to introduce a delay, posting a user-defined Windows message. Because the posted message is queued and not handled immediately, if we send it at the last possible moment of life of the secondary form, the main form will receive it when the other form is destroyed.

The trick is to post the message in the `OnDestroy` event handler of the secondary form. To accomplish this, we need to refer to the `MainForm` object, by adding a `uses` statement in the implementation portion of this unit. I've posted a `wm_User` message, which is handled by a specific `message` method of the main form, as shown here:

```
public
```

```
  procedure ChildClosed (var Message: TMessage);
    message wm_User;
```

Here is the code for this method:

```
procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (self);
end;
```

The problem here is that if you close the main window before closing the secondary forms, the main form exists, but its code cannot be executed anymore. To avoid another system error (an Access Violation Fault), you need to post the message only if the main form is not closing. But how do you know that? One way is to add a flag to the `TMainForm` class and change its value when the main form is closing, so that you can test the flag from the code of the secondary window.

This is a good solution—so good that the VCL already provides something similar. There is a barely documented `ComponentState` property. It is a Pascal set that includes (among other flags) a `csDestroying` flag, which is set when the form is closing. Therefore, we can write the following code:

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

With this code, the list box always lists all of the forms in the application. Note that you need to disable the automatic creation of the secondary form by using the Forms page of the Project Options dialog box.

After giving it some thought, however, I found an alternative and much more Delphi-oriented solution. Every time a component is destroyed, it tells its owner about the event by calling the `Notification` method defined in the `TComponent` class. Because the secondary forms are owned by the main one, as specified in the code of the `NewButtonClick` method, we can override this method and simplify the code. In the form class, simply write

```
protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
```

Here is the code of the method:

```
procedure TMainForm.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
```

```
  inherited Notification(AComponent, Operation);
  if Showing and (AComponent is TForm) then
      FillFormsList;
end;
```

You'll find the complete code of this version in the Screen2 directory.

note In case the secondary forms were not owned by the main one, we could have used the `FreeNotification` method to get the secondary form to notify the main form when they are destroyed. `FreeNotification` receives as parameter the component to notify when the current component is destroyed. The effect is a call to the `Notification` method coming from a component other than the owned ones. `FreeNotification` is generally used by component writers to safely connect components on different forms or data modules.

The last feature I've added to both versions of the program is a simple one. When you click on an item in the list box, the corresponding form is activated, using the `BringToFront` method:

```
procedure TMainForm.FormsListBoxClick(Sender: TObject);
begin
  Screen.Forms [FormsListBox.ItemIndex].BringToFront;
end;
```

Nice—well, almost nice. If you click on the list box of an inactive form, the main form is activated first, and the list box is rearranged, so you might end up selecting a different form than you were expecting. If you experiment with the program, you'll soon realize what I mean. This minor glitch in the program is an example of the risks you face when you dynamically update some information and let the user work on it at the same time.

Closing a Form

When you close the form using the `Close` method or by the usual means (Alt+F4, the system menu, or the Close button), the `OnCloseQuery` event is called. In this event, you can ask the user to confirm the action, particularly if there is unsaved data in the form. Here is a simple scheme of the code you can write:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if MessageDlg ('Are you sure you want to exit?',
      mtConfirmation, [mbYes, mbNo], 0) = idNo then
    CanClose := False;
end;
```

If `OnCloseQuery` indicates that the form should still be closed, the `OnClose` event is called. The third step is to call the `OnDestroy` event, which is the opposite of the `OnCreate` event and is generally used to deallocate objects related to the form and free the corresponding memory.

note To be more precise, the `BeforeDestruction` method generates an `OnDestroy` event before the `Destroy` destructor is called. That is, unless you have set the `OldCreateOrder` property to `True`, in which case Delphi uses a different closing sequence.

So what is the use of the intermediate `OnClose` event? In this method, you have another chance to avoid closing the application, or you can specify alternative "close actions." The method, in fact, has an `Action` parameter passed by reference. You can assign the following values to this parameter:

- `caNone`: The form is not allowed to close. This corresponds to setting the `CanClose` parameter of the `OnCloseQuery` method to `False`.
- `caHide`: The form is not closed, just hidden. This makes sense if there are other forms in the application; otherwise, the program terminates. This is the default for secondary forms, and it's the reason I had to handle the `OnClose` event in the previous example to actually close the secondary forms.
- `caFree`: The form is closed, freeing its memory, and the application eventually terminates if this was the main form. This is the default action for the main form and the action you should use when you create multiple forms dynamically (if you want to remove the Windows and destroy the corresponding Delphi object as the form closes).
- `caMinimize`: The form is not closed but only minimized. This is the default action for MDI child forms, as we'll see in Chapter 8.

note When a user shuts down Windows, the `OnCloseQuery` event is activated, and a program can use it to stop the shut-down process. In this case, the `OnClose` event is not called even if `OnCloseQuery` sets the `CanClose` parameter to `True`.

Form Input

Having discussed some special capabilities of forms, I'll now move to a very important topic: user input in a form. If you decide to make limited use of components,

you might write complex programs as well, receiving input from the mouse and the keyboard. In this chapter, I'll only introduce this topic. More about graphics can be found in Chapter 22, "Graphics in Delphi."

Supervising Keyboard Input

Generally, forms don't handle keyboard input directly. If a user has to type something, your form should include an edit component or one of the other input components. If you want to handle keyboard shortcuts, you can use those connected with menus (possibly using a hidden pop-up menu).

At other times, however, you might want to handle keyboard input in particular ways for a specific purpose. What you can do in these cases is turn on the `KeyPreview` property of the form. Then, even if you have some input controls, the form's `OnKeyPress` event will always be activated for any keyboard-input operation. The keyboard input will then reach the destination component, unless you stop it in the form by setting the character value to zero (not the character *0,* but the value 0 of the character set, indicated as `#0`).

The example I've built to demonstrate this, KPreview, has a form with no special properties (not even `KeyPreview`), a radio group with four options, and some edit boxes, as you can see in Figure 6.10.

By default the program does nothing special, except when the different radio buttons are used to enable the key preview:

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

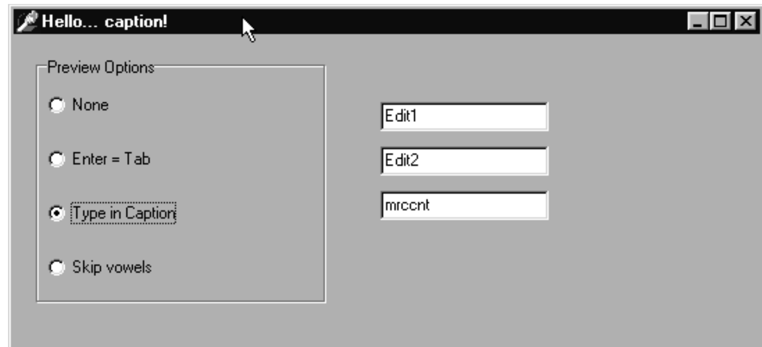


Figure 6.10: The KPreview program allows you to type into the caption of the form (among other things). Image from the original book.

Now we'll start receiving the `OnKeyPress` events, and we can do one of the three actions requested by the three special buttons of the radio group. The action depends on the value of the `ItemIndex` property of the radio group component. This is the reason the event handler is based on a `case` statement:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

In the first case, if the value of the `Key` parameter is `#13`, which corresponds to the Enter key, we disable the operation (setting `Key` to zero) and then mimic the activation of the Tab key. There are many ways to accomplish this, but the one I've chosen is quite particular. I send the `CM_DialogKey` message to the form, passing the code for the Tab key (`VK_TAB`):

```
    1: // Enter = Tab
      if Key = #13 then
      begin
        Key := #0;
        Perform (CM_DialogKey, VK_TAB, 0);
      end;
```

note The `CM_DialogKey` message is an internal undocumented Delphi message, something that is really beyond the scope of this book but is discussed in other texts, including my own *Delphi Developer's Handbook* (Sybex, 1998).[186]

186 Not an easy book to find, thee days, but I have to say it has a lot of advanced content still valid and interesting today, along with areas of the libraries that have been significantly modified.

To type in the caption of the form, the program simply adds the character to the current `Caption`, as you can see in Figure 6.10. There are two special cases. When the Backspace key is pressed, the last character of the string is removed (by copying to the `Caption` all the characters of the current `Caption` but the last one). When the Enter key is pressed, the program stops the operation, by resetting the `ItemIndex` property of the radio group control. Here is the code:

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1,
      Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
    Caption := Caption + Key;
  Key := #0;
end;
```

Finally, if the last radio item is selected, the code checks whether the character is a vowel (by testing for its inclusion in a constant *vowel set*). In this case, the character is skipped altogether:

```
3: // skip vowels
  if Key in ['a', 'e', 'i', 'o', 'u',
      'A', 'E', 'I', 'O', 'U'] then
    Key := #0;
```

Getting Mouse Input

When a user presses one of the mouse buttons over a form (or over a component, by the way), Windows sends the application some messages. Delphi defines some events you can use to write code that responds to these messages. The two basic events are as follows:

- `OnMouseDown` is received when one of the mouse buttons is pressed.
- `OnMouseUp` is received when one of the buttons is released.

Another fundamental system message is related to mouse movement. The event is `OnMouseMove`. Although it should be easy to understand the meaning of the three messages—down, up, and move—the question that might arise is, how do they relate to the `OnClick` event we have often used up to now?

We have used the `OnClick` event for components, but it is also available for the form. Its general meaning is that the left mouse button has been pressed and

released on the same window or component. However, between these two actions, the cursor might have been moved outside the area of the window or component, while the left mouse button was held down. If you press the mouse button at a certain position and then move it away and release it, no click is involved. In this case, the window receives *only* a down message, some move messages, and an up message. Another difference is that the click event relates only to the left mouse button.

The Mouse Buttons

Most of the mouse types connected to a Windows PC have two mouse buttons, and some even have three. Usually we refer to these buttons as the left mouse button, which is the most used; the right mouse button; and the middle mouse button:

- The left mouse button is *the* mouse button. It is used to select elements on screen, to give menu commands, to click buttons, to select and move elements (*dragging*), to select and activate (usually with a double-click), and so on.
- The right mouse button is used for local pop-up menus. Many applications used this approach in the past, but Windows 95 has made local menus the standard effect of right-clicking.
- The middle button is seldom used because most users either don't have it or don't have a proper software driver. Some CAD programs use the middle button. If you want to support this button, it should be optional (or else you should be ready to provide your customers with a free three-button mouse and the corresponding driver).

Keep in mind that users can customize their mouse buttons, switching the left and right buttons and turning a single click on the middle button into a double-click of the left button. When you refer to events related to a mouse button in your code, what matters is not the physical button but rather its meaning.

note Beyond the three traditional mouse buttons, there are now some mouse devices with a "button wheel" instead of the middle button. Users typically use the wheel for scrolling (causing an `OnMouseWheel` event), but they can also press it (generating the `OnMouseWheelDown` and `OnMouseWheelUp` events). The up and down messages are similar to the mouse button messages, whereas the `OnMouseWheel` event is devoted to handling the scrolling operations. Mouse wheel events are automatically converted into scrolling events.

Using Windows without a Mouse

A user should always be able to use any Windows application without the mouse. This is not an option; it is a Windows programming rule. Of course, an application

might be easier to use with a mouse, but that should never be mandatory. In fact, there are users who for various reasons might not have a mouse connected, such as travelers with a small laptop and no space, workers in industrial environments, and bank clerks with a number of other peripherals around.

There is another reason, already mentioned in this chapter in respect to the menu, to support the keyboard: Using the mouse is nice, but it tends to be slower. If you are a skilled touch typist, you won't use the mouse to drag a word of text; you'll use shortcut keys to copy and paste it, without moving your hands from the keyboard.

For all these reasons, you should always set up a proper tab order for a form's components, remember to add keys for buttons and menu items for keyboard selection, use shortcut keys on menu commands, and so on. An exception to this rule might be a graphics program. However, be aware that you can use even a program such as Microsoft Paint without the mouse—although I don't recommend it.

The Parameters of the Mouse Events

Since I'm going to build a graphics program, I will focus only on the use of the mouse. The first event we need to consider for the first minimal version of the MouseOne program is `OnMouseDown`. The related method has a number of parameters, as shown in the following declaration:

```
procedure TShapesForm.FormMouseDown (
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

In addition to the usual `Sender` parameter, there are four more parameters:

- `Button` indicates which of the three mouse buttons has been pressed. Possible values are `mbRight`, `mbLeft`, and `mbCenter`. These are exclusive values because the purpose of this parameter is to determine which button generated the message.
- `Shift` indicates which *mouse-related keys* were pressed when the event occurred. These mouse-related keys are Alt, Ctrl, and Shift, plus the mouse buttons themselves. This parameter is of a set type since several keys (and mouse buttons) might be pressed at the same time. This means you should test for a condition using the `in` expression, not for equality.
- `X` and `Y` indicate the coordinates of the position of the mouse, in *client area* coordinates of the current window (a form or a control). The origin of the *x*-and *y*-axes of these coordinates is the upper-left corner of the client area of the window receiving the event (again, a form or a control).

Using this information, it is very simple to draw a small circle in the position of a left mouse button-down event:

```
procedure TForm1.FormMouseDown(
  Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

note To draw on the form, we use a very special property: `Canvas`. A `TCanvas` object has two distinctive features: it holds a collection of drawing tools (such as a pen, a brush, and a font) and it has a number of drawing methods, which use the current tools. This kind of direct drawing code in this example is not correct, because the on-screen image is not persistent: moving another window over the current one will clear its output. The next example demonstrates the Windows “store-and-draw” approach.

Dragging and Drawing with the Mouse

To demonstrate a few of the mouse techniques discussed so far, I’ve built a simple example based on a form without any component and called MouseOne. The first feature of this program is that it displays in the `Caption` of the form the current position of the mouse:

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format (‘Mouse in x=%d, y=%d’, [X, Y]);
end;
```

You can use this simple feature of the program to better understand how the mouse works. Make this test: run the program (this simple version or the complete one) and resize the windows on the desktop so that the form of the MouseOne program is behind another window and inactive but with the title visible. Now move the mouse over the form, and you’ll see that the coordinates change. This means that the `OnMouseMove` event is sent to the application even if its window is not active, and it proves what I have already mentioned: mouse messages are always directed to the window under the mouse. The only exception is the mouse capture operation I’ll discuss in this same example.

Besides showing the position in the title of the window, the MouseOne example can track mouse movements by painting small pixels on the form if the user keeps the Shift key pressed. (Again this direct painting code produces nonpersistent output.)

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if ssShift in Shift then
    // mark points in yellow
    Canvas.Pixels [X, Y] := clYellow;
end;
```

The real feature of this example, however, is the direct mouse dragging support. Contrary to what you might think, Windows has no system support for dragging, which is implemented in the VCL by means of lower-level mouse events and operations. (An example of dragging from one control to another was discussed in the last chapter.) In the VCL, forms cannot originate dragging operations, so in this case we are obliged to use the low-level approach. The aim of this example is to draw a rectangle from the initial position of the dragging operation to the final one, giving the users some visual clue of the operation they are doing.

The idea behind dragging is quite simple. The program receives a sequence of button-down, mouse-move, and button-up messages. When the button is pressed, dragging begins, although the real actions take place only when the user moves the mouse (without releasing the mouse button) and when dragging terminates (when the button-up message arrives).

The problem with this basic approach is that it is not reliable. A window usually receives mouse events only when the mouse is over its client area; so if the user presses the mouse button, moves the mouse onto another window, and then releases the button, the second window will receive the button-up message.

There are two solutions to this problem. One (seldom used) is mouse clipping. Using a Windows API function (namely `ClipCursor`), you can force the mouse not to leave a certain area of the screen. When you move it outside the specified area, it stumbles against an invisible barrier. The second and more common solution is to capture the mouse. When a window captures the mouse, all the subsequent mouse input is sent to that window. This is the approach we will use for the MouseOne example.

The code of the example is built around three methods: `FormMouseDown`, `FormMouseMove`, and `FormMouseUp`. Pressing the left mouse button over the form starts the process, setting the `fDragging` Boolean field of the form (which indicates that dragging is in action in the other two methods). The method also uses a `TRect`

variable used to keep track of the initial and current position of the dragging. Here is the code:

```
procedure TMouseForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    fDragging := True;
    SetCapture (Handle);
    fRect.Left := X;
    fRect.Top := Y;
    fRect.BottomRight := fRect.TopLeft;
    Canvas.DrawFocusRect (fRect);
  end;
end;
```

An important action of this method is the call to the `SetCapture` API function. Now even if a user moves the mouse outside of the client area, the form still receives all mouse-related messages. You can see that for yourself by moving the mouse toward the upper-left corner of the screen; the program shows negative coordinates in the caption.

When dragging is active and the user moves the mouse, the program draws a dotted rectangle corresponding to the actual position. Actually, the program calls the `DrawFocusRect` method twice. The first time this method is called, it deletes the current image, thanks to the fact that two consecutive calls to `DrawFocusRect` simply reset the original situation. After updating the position of the rectangle, the program calls the method a second time:

```
procedure TMouseForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
  begin
    // remove and redraw the dragging rectangle
    Canvas.DrawFocusRect (fRect);
    fRect.Right := X;
    fRect.Bottom := Y;
    Canvas.DrawFocusRect (fRect);
  end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
end;
```

When the mouse button is released, the program terminates the dragging operation by calling the `ReleaseCapture` API function and by setting the value of the `fDragging` field to `False`:

```
procedure TMouseForm.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
  begin
    ReleaseCapture;
    fDragging := False;
    Invalidate;
  end;
end;
```

The final call, `Invalidate`, triggers a painting operation and executes the following `OnPaint` event handler:

```
procedure TMouseForm.FormPaint(Sender: TObject);
begin
  Canvas.Rectangle (fRect.Left, fRect.Top,
    fRect.Right, fRect.Bottom);
end;
```

This makes the output of the form persistent, even if you hide it behind another form. Figure 6.11 shows a previous version of the rectangle and a dragging operation in action.

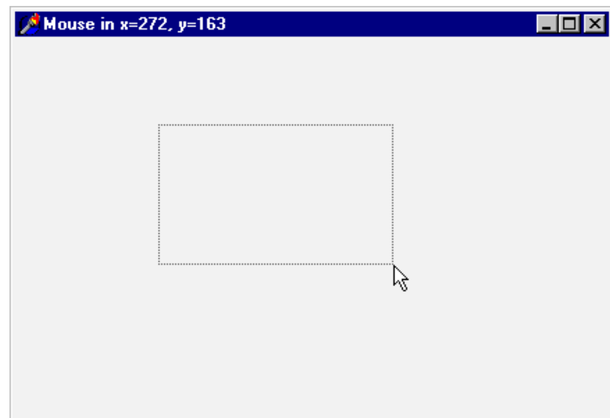


Figure 6.11: The MouseOne example uses a dotted line to indicate, during a dragging operation, the final area of a rectangle. Image from the original book.

Painting in Windows

Why do we need to handle the `OnPaint` event to produce a proper output, and why can we not paint directly over the form canvas? It depends on Windows' default behavior. As you draw on a window, Windows does *not* store the resulting image. When the window is covered, its contents are usually lost[187].

The reason for this behavior is simple: to save memory. Windows assumes it's "cheaper" in the long run to redraw the screen using code than to dedicate system memory to preserving the display state of a window. It's a classic memory versus CPU cycles trade-off. A color bitmap for a 300 x 400 image at 256 colors requires about 120KB. By increasing the color count or the number of pixels, you can easily have full-screen bitmaps of about 1MB and reach 4MB of memory for a 1280 x 1024 resolution at 16 million colors. If storing the bitmap was the default choice, running half a dozen simple applications would require at least 8MB of memory, if not 16MB, just for remembering their current output[188].

In the general case you want to have a consistent output for your applications, there are two techniques you can use. The general solution is to store enough data about the output to be able to reproduce it when the systems sends a *painting* requested. An alternative approach is to save the output of the form in a bitmap while you produce it, by placing an Image component over the form and drawing on the canvas of this image component.

The first technique, painting, is the common approach to handling output in Windows, aside from particular graphics-oriented programs that store the form's whole image in a bitmap. The approach used to implement painting has a very descriptive name: *store and paint*. In fact, when the user presses a mouse button or performs any other operation, we need to store the position and other elements; then, in the painting method, we use this information to actually paint the corresponding image.

The idea of this approach is to let the application repaint its whole surface under any of the possible conditions. If we provide a method to redraw the contents of the form, and if this method is automatically called when a portion of the form has been hidden and needs repainting, we will be able to re-create the output properly.

---

187 Core Windows painting concepts haven't changed at all over the years.

188 Needless to say some of the memory usage observations makes little sense in today's world, although we get inquiries about issues with multi-gigabytes bitmaps, which make me wonder if developers have an idea of the fact memory is finite anyway, even if so much larger.

Since this approach takes two steps, we must be able to execute these two operations in a row, asking the system to repaint the window—without waiting for the system to ask for this. You can use several methods to invoke repainting: `Invalidate`, `Update`, `Repaint`, and `Refresh`. The first two correspond to the Windows API functions, while the latter two have been introduced by Delphi.

- The `Invalidate` method informs Windows that the entire surface of the form should be repainted. The most important thing is that `Invalidate` does *not* enforce a painting operation immediately. Windows simply stores the request and will respond to it only after the current procedure has been completely executed and as soon as there are no other events pending in the system. Windows deliberately delays the painting operation because it is one of the most time-consuming operations. At times, with this delay, it is possible to paint the form only after a number of changes have taken place, avoiding a number of consecutive calls to the (slow) paint method.
- The `Update` method asks Windows to update the contents of the form, repainting it immediately. However, remember that this operation will take place only if there is an *invalid area*. This happens if the `Invalidate` method has just been called or as the result of an operation by the user. If there is no invalid area, a call to `Update` has no effect at all. For this reason, it is common to see a call to `Update` just after a call to `Invalidate`. This is exactly what is done by the two Delphi methods, `Repaint` and `Refresh`.
- The `Repaint` method calls `Invalidate` and `Update` in sequence. As a result, it activates the `OnPaint` event immediately. There is a slightly different version of this method called `Refresh`. For a form the effect is the same; for components it might be slightly different.

When you need to ask the form for a repaint operation, you should generally call `Invalidate`, following the standard Windows approach. This is particularly important when you need to request this operation frequently, because if Windows takes too much time to update the screen, the requests for repainting can be accumulated into a simple repaint action. The `wm_Paint` message in Windows is a sort of low-priority message. To be more precise, if a request for repainting is pending but other messages are waiting, the other messages are handled before the system actually performs the paint action.

On the other hand, if you call `Repaint` several times, the screen must be repainted each time before Windows can process other messages, and because paint operations are computationally intensive, this can actually make your application less responsive. There are times, however, when you want the application to repaint a

surface as quickly as possible. In these less-frequent cases, calling `Repaint` is the way to go.

---

note Another important consideration is that during a paint operation Windows redraws only the so-called *update region*, to speed up the operation. For this reason if you invalidate only a portion of a window, only that area will be repainted. To accomplish this you can use the `InvalidateRect` and `InvalidateRegion` functions. Actually, this feature is a double-edged sword. It is a very powerful technique, which can improve speed and reduce the flickering caused by frequent repaint operations. On the other hand, it can also produce incorrect output. A typical problem is when only some of the areas affected by the user operations are actually modified, while others remain as they were even if the system executes the source code that is supposed to update them. In fact, if a painting operation falls outside the update region, the system ignores it, as if it were outside the visible area of a window.

What's Next?

In this chapter we've explored some important form properties. Now we know how to handle the size and position of a form, how to resize it, and how to get mouse input and paint over it. We've also discussed in detail the role of two global objects, `Application` and `Screen`, and we've built applications with multiple forms. In Chapter 8, we'll extend this to cover dialog boxes in more detail.

Other chapters in the book will describe topics related to forms. In particular, Chapter 22, which was originally a bonus chapter available as a separate download; Chapter 7, the use of toolbars, status bars, and scrolling forms; Chapter 8, building a dialog box, forms with multiple pages, and MDI applications. As you can see from this list, forms play a central role in Delphi programming, and we still have to explore a number of topics related to them.

Chapter 7: Building A User Interface

One of the distinctive features of many Windows applications is the presence of a toolbar at the top of the window and a status bar at its bottom[189]. The toolbar usually contains a number of small buttons the user can click to give commands or to toggle options on and off. At times, a toolbar can also contain combo boxes, edit boxes, or other controls. The toolbars of the current generation of big applications usually can be moved to the left or right of the window, or even hidden and turned into a *toolbox,* a small floating window with an array of buttons.

189 This is still a common UI, although many alternatives emerged and became popular over the years. Modern apps tend to reduce the visible UI elements, which makes them nicer aesthetically, but often not so easy to use.

More complex applications tend to have multiple toolbars the user can configure. In Delphi you can use either the native ControlBar component or the Win32 CoolBar control[190], originally introduced by Microsoft Internet Explorer, for this purpose.

Toolbars account only for the first part of this chapter, which also covers the docking support that was introduced in Delphi 4 and presents examples of splitting forms, resizing controls dynamically, and scrolling the content of a form. These topics are not particularly complex, but it is worth examining their key concepts briefly.

The Toolbar Control

In early versions of Delphi, toolbars had to be created using panels and speed buttons, as briefly described in "Building a Toolbar with a Panel" later in this chapter. Starting with version 3, Delphi introduced a specific Toolbar component, which encapsulates the corresponding Win32 common control. This component provides a toolbar, with its own buttons, and it has some extended capabilities.

You've already seen examples of the Toolbar component in the Chapter 5 discussion of actions. To use this component, you place it on a form and then use the component editor (the shortcut menu activated by a right mouse button click) to create a few buttons and separators. You can see an example of a Toolbar component under construction in Figure 7.1.

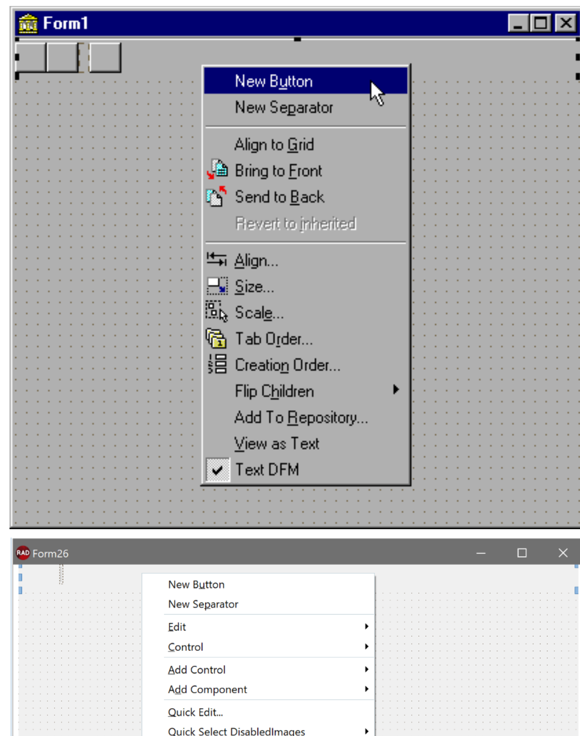
The Toolbar is populated with objects of the `TToolButton` class. These are *internal* objects, just as a `TMenuItem` is an internal object of a MainMenu component. These objects have a fundamental property, `Style`, which determines their behavior[191]:

- The `tbsButton` style indicates a standard push button.
- The `tbsCheck` style indicates a button with the behavior of a check box, or that of a radio button if the button is `Grouped` with the others in its block (determined by the presence of separators).

190 The Coolbar controls, while still available, is rarely used these days. I've kept the coverage, though.

191 There are now two further toolbar button styles, `tbsTextButton` and `tbsWholeDropDown`.

Figure 7.1: To create a toolbar, you can place the corresponding component on a form and then use its shortcut menu to add buttons and separators. Images captured in Delphi 5 and Delphi 12.



- The `tbsDropDown` style indicates a drop-down button, a sort of combo box. The drop-down portion can be easily implemented in Delphi by connecting a PopupMenu control to the `DropdownMenu` property of the control.
- The `tbsSeparator` and `tbsDivider` styles indicate separators with no or different vertical lines (depending on the `Flat` property of the toolbar).

To create a graphic toolbar, you can add an ImageList component to the form[192], load some bitmaps into it, and then connect the ImageList with the `Images` property of the toolbar. By default the images will be assigned to the buttons in the order they appear, but you can change this quite easily by setting the `ImageIndex` property of each toolbar button. You can prepare further ImageLists for special conditions of

192 Instead of using an image list, it's now recommended to use an ImageCollection and a VirtualImageList. The combination of these controls allows your application to select the correct image depending on the HighDPI resolution the application is running on. In the ImageCollection you can provide multiple set of images for different resolutions, or the component can create those automatically for you by resizing the available ones. By using the old approach explained here, you can end up with toolbars having very small images on HighDPI.

the buttons and assign them to the `DisabledImages` and `HotImages` properties of the toolbar. The first group is used for the disabled buttons, the second for the button currently under the mouse. This is the effect introduced by Microsoft Internet Explorer.

In a nontrivial application, you would generally add an ActionList component, particularly if you plan to have a menu with options that duplicate toolbar buttons (for example, a File ➢ Save menu option and a Save button). In this case you'll attach very little behavior to the toolbar buttons, as their properties and events will be managed by the action components. For example, you can obtain a toolbar button that toggles between a "selected" and an "unselected" state, like a check box. You obtain this effect by toggling the value of the `Checked` property of the action every time this is executed. In this case there is no need to set up the toolbar button with the `tbsCheck` style, as the code will determine the requested behavior.

The Toolbar and the ActionList of an Editor

In the MdEdit1 example, I've built a menu and a toolbar around a RichEdit control, providing the first step of an RTF (Rich Text File) editor I'll expand further in this and future chapters.

The application is based on an ActionList component, which includes actions for file handling and Clipboard support, and handles font and paragraph attributes. My aim is not to build a full-featured editor, or to investigate each and every feature of the RichEdit common control. I simply want to show how to build the user interface of a program, for which purpose it is valuable to work with a useful example. Rather than discuss all of the features of the program, I'll only highlight the points related to the current discussion. For a more detailed description of the code, you can open the "MdEdit Basics" RTF document available with the source code of the project.

The toolbar of the MdEdit1 example has most of its buttons connected to actions, which are available in a single ActionList component used to handle also all of the menu items. Only the last button, which has the `tbsDropDown` style, is handled directly and not through an action. Here is the structure of the toolbar:

```
object ToolBar1: TToolBar
  AutoSize = True
  Flat = True
  Images = Images
  object ToolButton1: TToolButton
    Action = acNew
  end
  object ToolButton2: TToolButton
```

```
    Action = acOpen
  end
  ...
  object ToolButton17: TToolButton
    DropdownMenu = SizeMenu
    ImageIndex = 13
    Style = tbsDropDown
    OnClick = ToolButton17Click
  end
end
```

The last button is connected to a PopupMenu component (called `SizeMenu`). This is all you have to do to make it display the list of items when the down arrow is selected, as you can see in Figure 7.2. Because the button can also be clicked, I've provided an event handler, which increases the size of the selected text.

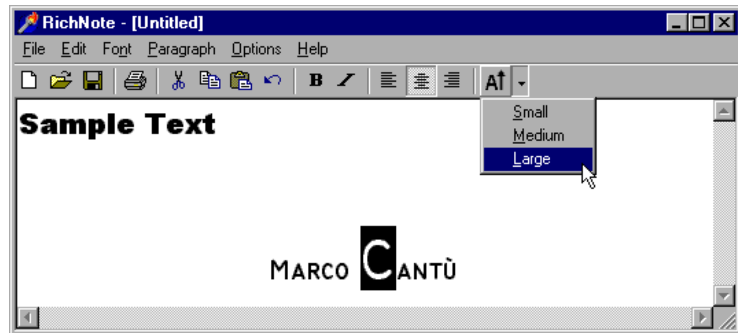


Figure 7.2: The toolbar of the MdEdit1 example has a drop-down button connected to a pop-up menu. Image from the original book.

The three paragraph-alignment buttons have their `Grouped` property set to `True`, forming a group (as they are enclosed between two separators). This is required because the program checks the action corresponding to the current style, in the `OnUpdate` event of the action list, but it fails to uncheck the other two actions. The user interface behavior of the menu items is determined by their `RadioItem` style and that of the toolbar buttons with the grouping and the `AllowAllUp` property.

Building a Toolbar with a Panel

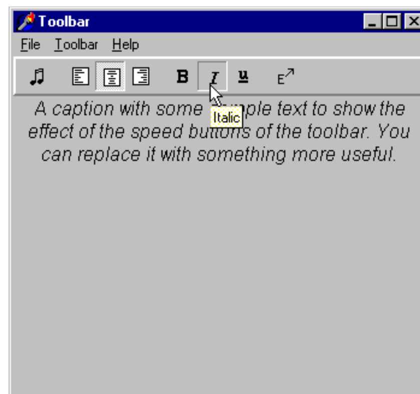
Before the toolbar control was available in Delphi, the standard approach for building a toolbar was to use a panel aligned to the top of the form and place a number of SpeedButton components inside it. A *speed button* is a lightweight graphical control (consuming no Windows resources); it cannot receive the

input focus, it has no tab order, and it is faster to create and paint than a bitmap button.

Speed buttons can behave like push buttons, check boxes, or radio buttons, and they can have different bitmaps depending on their status. To make a group of speed buttons work like radio buttons, just place some speed buttons on the panel, select all of them, and give the same value to each one's `GroupIndex` property. All the buttons having the same `GroupIndex` become mutually exclusive selections. One of these buttons should always be selected, so remember to set the `Down` property to `True` for one of them at design time or as soon as the program starts.

By setting the `AllowAllUp` property, you can create a group of mutually exclusive buttons, each of which can be *up*—that is, a group from which the user can select one option or leave them all unselected. As a special case, you can make a speed button work as a check box, simply by defining a group (the `GroupIndex` property) that has only one button and that allows it to be deselected (the `AllowAllUp` property).

Finally, you can set the `Flat` property of all the SpeedButton components to `True`, obtaining a more modern user interface. If you are interested in this approach, you can look at the PanelBar example, illustrated here:



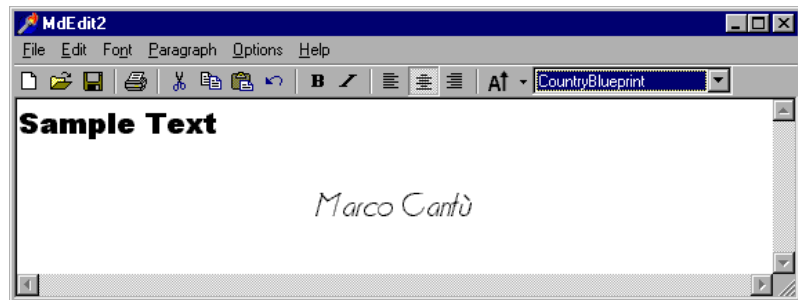
The use of SpeedButton controls is becoming less common. Besides the fact that the Toolbar control is very handy and definitely more standard, speed buttons have two big problems. First, each of them requires a specific bitmap and cannot use one from an image list (unless you write some complex code).

> Second, speed buttons don't work very well with actions, because some properties such as the `Down` state do not map directly.

A Combo Box in a Toolbar

We can extend this example by adding a combo box to the toolbar. A number of common applications use combo boxes in toolbars to show lists of styles, fonts, font sizes, and so on. Because we've used a drop-down button for the font size, we can add a combo box to allow rapid selection of the font family. This is simple to accomplish, as the Toolbar control is a full-featured control container; you can directly take an edit box, a combo box, and other controls and place them inside the toolbar. Figure 7.3 shows the MdEdit2 application, with its font-selection combo box.

Figure 7.3: The MdEdit2 example at run time. Image from the original book.



The combo box in the toolbar is initialized in the `FormCreate` method, which extracts the screen fonts available in the system:

```
ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (
  RichEdit.Font.Name)
```

The combo box initially displays the name of the default font used in the RichEdit control, which is set at design time. This value is recomputed each time the current selection changes, using the font of the selected text:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ComboFont.ItemIndex :=
    ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name)
end;
```

When a new font is selected from the combo box, the reverse action takes place. The text of the current combo box item is assigned as the name of the font for any text selected in the RichEdit control:

```
procedure TFormRichNote.ComboFontClick(Sender: TObject);
begin
  RichEdit.SelAttributes.Name :=
    ComboFont.Text;
end;
```

Toolbar Hints

Another common element in toolbars is the *fly-by hint*, also called *balloon help*—some text that briefly describes the button currently under the cursor. This text is usually displayed in a yellow box after the mouse cursor has remained steady over a button for a set amount of time. To add hints to an application's toolbar, simply set its `ShowHints` property to `True`.

I want to use the `Caption` of each action as its hint, so I could simply copy them all at run time, instead of setting each at design time. The problem is that the captions include the ampersand character used for the menu shortcuts. We can solve this by removing those extra characters with the new `StripHotKey` function in the Menus unit. Here is the code:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  ...
  // move captions to hints, removing the &
  for I := 0 to ActionList.ActionCount - 1 do
    (ActionList.Actions[I] as TAction).Hint :=
      StripHotKey ((ActionList.Actions[I] as TAction).Caption);
end;
```

As you can see in Figure 7.4, the hints also include a string showing the shortcut associated with each menu item, as a reminder to the user. This is a default behavior you can disable by setting the `HintShortCuts` property of the `Application` object. This global object controls the hints with other properties and some methods and events. For example, you can change the `HintColor`, `HintPause`, `HintHidePause`, and `HintShortPause` properties. The MdEdit2 example allows a user to customize the hint background color by selecting a specific menu item (Options ➢ Hint Color), with the following event handler:

```
procedure TFormRichNote.acHintColorExecute (Sender: TObject);
begin
  ColorDialog.Color := Application.HintColor;
  if ColorDialog.Execute then
    Application.HintColor := ColorDialog.Color;
end;
```

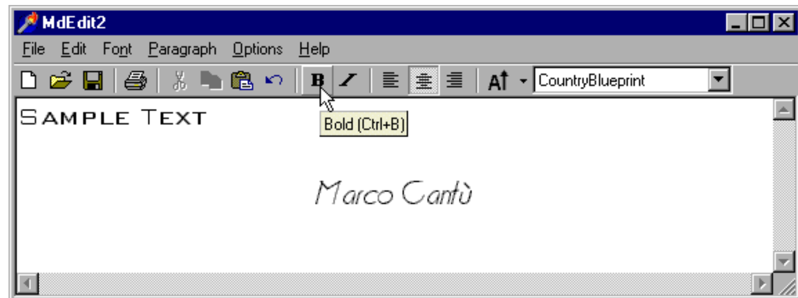


Figure 7.4: The hints displayed by the MdEdit2 example. Image from the original book.

note As an alternative, you can change the hint color by handling the `OnShowHint` property of the `Application` object. This handler can change the color of the hint just for specific controls. The `OnShowHint` event is used in the following CustHint example.

Customizing the Hints

Just as we have added hints to an application's toolbar, we can add hints to forms or to the components of a form. For a large control, the hint will show up near the mouse cursor. In some cases, it is important to know that a program can freely customize how hints are displayed[193].

The simplest thing you can do is change the value of the `HintColor` property of the `Application` object (as in the previous example) and the three properties related to the hint pause: `HintPause`, `HintHidePause`, and `HintShortPause`. The first defines how long the cursor should remain on a component before hints are displayed, the

193 There is now also a BalloonHint component you can use to create hints with a more complex UI structure and including more information. These new hints require Windows themes to be active.

second how long the hint will be displayed, and the third how long the system should wait to display a hint if another hint has just been displayed.

To obtain more control over hints, you can customize them even further by assigning a method to the application's `OnShowHint` event. You need to either hook them up manually or—better—add an ApplicationEvents component to the form and handle its `OnShowHint` event.

The method you have to define has some interesting parameters, such as a string with the text of the hint, a `Boolean` flag for its activation, and a structure with further information:

```
TShowHintEvent = procedure (
  var HintStr: string;
  var CanShow: Boolean;
  var HintInfo: THintInfo) of object;
```

Each of the parameters is passed by reference, so you have a chance to change it. The last parameter is a structure, containing a reference to the control, the position of the hint, its color, and other information:

```
THintInfo = record
  HintControl: TControl;
  HintPos: TPoint;
  HintMaxWidth: Integer;
  HintColor: TColor;
  CursorRect: TRect;
  CursorPos: TPoint;
end;
```

You can modify the values of this structure; for example, you can change the position of the hint window before it is displayed. This is what I've done in the CustHint example, which shows the hint of the label at the center of its area. Here is what you can write to show the hint for the big label in the center of its surface:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
begin
  with HintInfo do
    if HintControl = Label1 then
      HintPos := HintControl.ClientToScreen (Point (
        HintControl.Width div 2, HintControl.Height div 2));
end;
```

The code has to retrieve the center of the generic control (the `HintInfo.HintControl`) and then convert its coordinates to screen coordinates, applying the `ClientToScreen` method to the control itself.

We can further update the CustHint example in a different way. The RadioGroup control in the form has three radio buttons. However, these are not stand-alone components, but simply radio button clones painted on the surface of the radio group. What if we want to add a hint for each of them?

The `CursorRect` field of the `THintInfo` record can be used for this purpose. It indicates the area of the component that the cursor can move over without disabling the hint. When the cursor moves outside this area, Delphi hides the hint window. If we specify a different text for the hint and a different area for each of the radio buttons, we can in practice provide three different hints. Since computing the actual position of each radio button isn't easy, I've simply divided the surface of the radio group into as many equal parts as there are radio buttons. The text of the radio button (not the selected item, but the item under the cursor) is then added to the text of the hint:

```
procedure TForm1.ShowHint (var HintStr: string;
  var CanShow: Boolean; var HintInfo: THintInfo);
var
  RadioItem, RadioHeight: Integer;
  RadioRect: TRect;
begin
  with HintInfo do
    if HintControl = Label1 ... // as before
    else
    if HintControl = RadioGroup1 then
    begin
      RadioHeight := (RadioGroup1.Height) div
        RadioGroup1.Items.Count;
      RadioItem := CursorPos.Y div RadioHeight;
      HintStr := 'Choose the ' +
        RadioGroup1.Items [RadioItem] + ' button';
      RadioRect := RadioGroup1.ClientRect;
      RadioRect.Top := RadioRect.Top +
        RadioHeight * RadioItem;
      RadioRect.Bottom := RadioRect.Top + RadioHeight;
      // assign the hints rect and pos
      CursorRect := RadioRect;
    end;
end;
```

The final part of the code builds the rectangle for the hint, starting with the rectangle corresponding to the client area of the component and moving its `Top` and `Bottom` values to the proper section of the `RadioGroup1` component. The resulting effect is that each radio button of the radio group appears to have a specific hint, as shown in Figure 7.5.

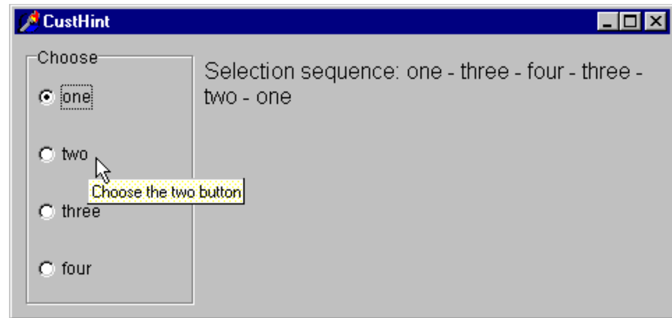


Figure 7.5: The RadioGroup control of the CustHint example shows a different hint, depending on the radio button the mouse is over. Image from the original book.

Toolbar Containers

Most modern applications have multiple toolbars, generally hosted by a specific container. Microsoft Internet Explorer, the various standard business applications, and the Delphi IDE all use this general approach. However, they each implement it differently. Delphi has two ready-to-use toolbar containers, the CoolBar and the ControlBar components. They have differences in their user interface, but the biggest one is that the CoolBar is a Win32 common control, part of the operating system, while the ControlBar is a VCL-based component.

Both components can host toolbar controls, as well as some extra elements, such as combo boxes and other controls. Actually, a toolbar can also replace the menu of an application, as we'll see later on.

We'll investigate the two components in the next two sections, but I want to emphasize here (without getting too far ahead of myself) that I generally favor the use of the ControlBar. It is based on the VCL (and not subject to upgrade along with each minor release of Microsoft Internet Explorer), and its user interface is nicer and more similar to that of common office applications.

A Really Cool Toolbar

The CoolBar component is basically a collection of `TCoolBand` objects. Unlike the toolbar buttons, these objects do not appear as stand-alone objects in the form, but are simply a collection of subitems. They appear in the Object Inspector only when

you select the editor of the CoolBar's `Bands` property, as you can see in Figure 7.6. You create one or more bands and then set their attributes.

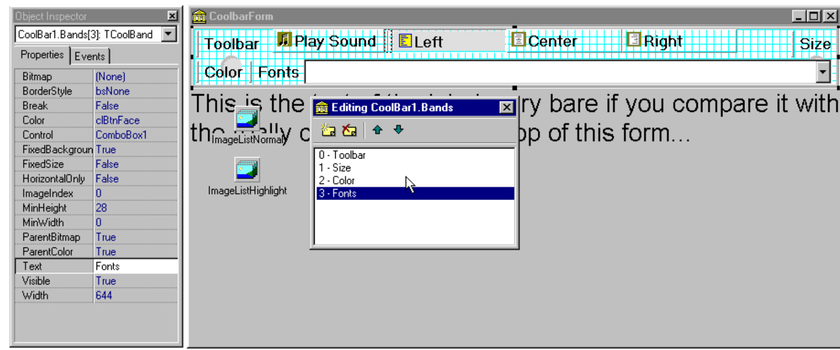


Figure 7.6: The property editor of the CoolBar component's Bands property works in conjunction with the Object Inspector. Image from the original book.

You can customize the CoolBar component in many ways: You can set a bitmap for its background, add some bands using the editor for the `Bands` property, and then assign to each band an existing component or component container. You can use any window-based control (not graphic controls), but only some of them will show up properly. If you want to have a bitmap on the background of the CoolBar, for example, you need to use partially transparent controls.

The typical component used in a CoolBar is the Toolbar (which can be made completely transparent), but combo boxes, edit boxes, and animation controls are also quite common. This is often inspired by the user interface of the Internet Explorer, the first Microsoft application featuring the CoolBar component.

You can place one band on each line or all of them on the same line. Each one would use a part of the available surface, and it would be automatically enlarged when the user clicks on its title. It is easier to use this new component than to explain it. Try it yourself or follow the description below, in which we build a new version of our continuing toolbar example based on a CoolBar control. You can see the form displayed by this application at run time in Figure 7.7.

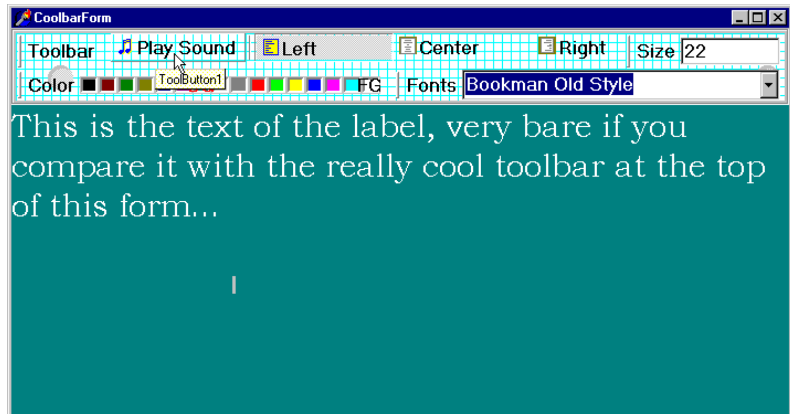


Figure 7.7: The form of the CoolBar example at run time. Image from the original book.

The CoolBar example has a `TCoolBar` component with four bands, two for each of the two lines. The first band includes a subset of the toolbar of the previous example, this time adding an ImageList for the highlighted images. The second has an edit box used to set the font of the text; the third has a ColorGrid component, used to choose the font color and that of the background. The last band has a ComboBox control with the available fonts.

The ControlBar

The user interface of the CoolBar component is really very nice, and Microsoft is increasingly using it in its applications[194]. However, the Windows CoolBar control has had many different and incompatible versions, as Microsoft has released different versions of the common control library with different versions of the Internet Explorer. Some of these versions “broke” existing programs built with Delphi.

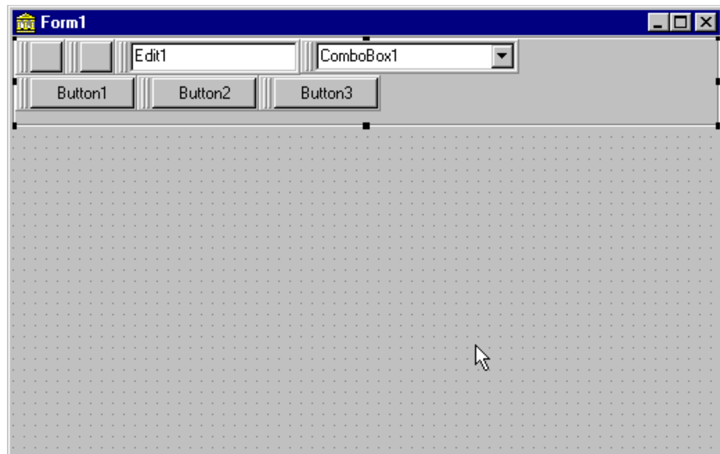
note It is interesting to note that Microsoft applications generally don’t use the common control libraries. Word and Excel use their own internal versions of the common controls, and VB uses an OCX, not the common controls directly. Part of the reason that Borland had so much trouble with the common controls is that it uses them more (and in more ways) than even Microsoft does.

194 This was true at the time. Microsoft later moved towards using the Ribbon control even outside of of Office, where it was originally introduced. The Ribbon control is a common replacement of toolbars.

For this reason, Borland introduced (in Delphi 4) a toolbar container called the ControlBar. A control bar hosts several controls, as a CoolBar does, and offers a similar user interface that lets a user drag items and reorganize the toolbar at run time. A good example of the use of the ControlBar control is Delphi's own toolbar, but Microsoft applications use a very similar user interface.

The ControlBar is a control container, and you build it just by placing other controls inside it, as you do with a panel. Every control placed in the bar gets its own dragging area (a small panel with two vertical lines, on the left of the control), as you can see in Figure 7.8. For this reason, you should generally avoid placing specific buttons inside the ControlBar, but rather add further containers with buttons inside them. Rather than using a panel, you should generally use one ToolBar control for every section of the toolbar.

Figure 7.8: The ControlBar is a container that allows a user to drag all the elements, using the special drag bar on the side. Notice that each button gets a separate drag bar, something you'll generally try to avoid. Image from the original book.



The MdEdit3 example is another version of the RichEdit demo we've developed throughout this chapter. I've basically grouped the buttons into three toolbars (instead of a single one) and left the combo box as a stand-alone control. All these components are inside a ControlBar, so that a user can arrange them at will, as you can see in Figure 7.9 and in the following DFM listing:

```
object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  object ToolBarFile: TToolBar
    AutoSize = True
    EdgeBorders = []
```

```
    EdgeInner = esNone
    EdgeOuter = esNone
    Flat = True
    Images = Images
    Wrapable = False
    object ToolButton1: TToolButton
      Action = acNew
    end
    // more buttons
  end
  object ToolBarEdit: TToolBar
    // similar properties
    object ToolButton6: TToolButton
      Action = acCut
    end
    // more buttons
  end
  object ToolBarFont: TToolBar
    // ...
  end
  object ComboFont: TComboBox
    Hint = 'Font Family'
    Style = csDropDownList
    Font.Height = -11
    Font.Name = 'Arial'
    ItemHeight = 14
    ParentFont = False
    Sorted = True
    OnClick = ComboFontClick
  end
end
```

Notice in the listing that to obtain the standard effect, you have to disable the edges of the toolbar controls and set their style to flat. Sizing all the controls alike, so that you obtain one or two rows of elements of the same height, is not as easy as it might seem at first. Some controls have automatic sizing or various constraints. In particular, to make the combo box the same height as the toolbars, you have to tweak the type and size of its font. Resizing the control itself has no effect.

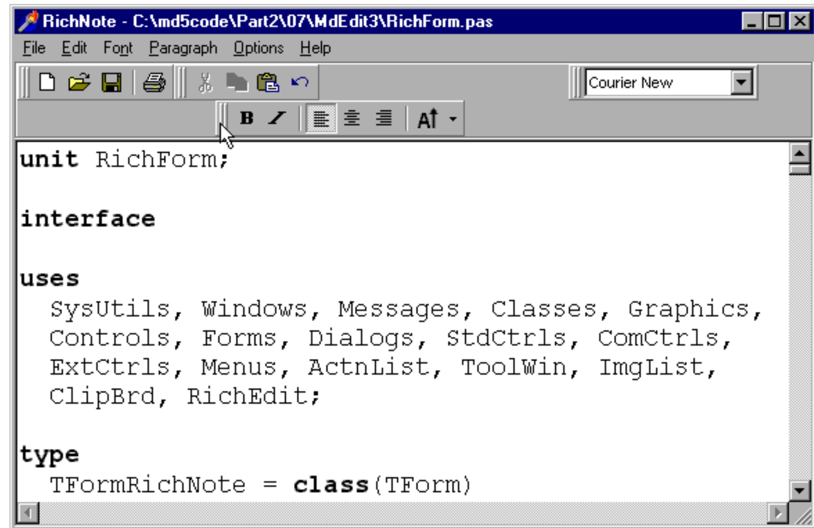


Figure 7.9: The MdEdit3 example at run time, while a user is rearranging the toolbars in the control bar. Image from the original book.

The ControlBar also has a shortcut menu that allows you to show or hide each of the controls currently inside it. Instead of writing code specific to this example, I've implemented a more generic (and reusable) solution. The shortcut menu, called `BarMenu`, is empty at design time and is populated when the program starts:

```
procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
begin
  ...
  // populate the control bar menu
  for I := 0 to ControlBar.ControlCount - 1 do
  begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
  end;
```

The `BarMenuClick` procedure is a single event handler that is used by all of the items of the menu and uses the `Tag` property of the `Sender` menu item to refer to the element of the ControlBar associated with the item in the `FormCreate` method:

```
procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
  aCtrl: TControl;
```

```
begin
  aCtrl := TControl ((Sender as TComponent).Tag);
  aCtrl.Visible := not aCtrl.Visible;
end;
```

Finally, the `OnPopup` event of the menu is used to refresh the check mark of the menu items:

```
procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
  I: Integer;
begin
  // update the menu checkmarks
  for I := 0 to BarMenu.Items.Count - 1 do
    BarMenu.Items [I].Checked :=
      TControl (BarMenu.Items [I].Tag).Visible;
end;
```

A Menu in a Control Bar

If you look at the user interface of the Delphi development environment, you can see that a ControlBar also hosts the application's menu, which can be dragged in the same way as the toolbars and the Component Palette[195]. How can we add a menu to the ControlBar of our application?

The menu of the form cannot be placed inside the ControlBar, but we can add another new toolbar control to host it. This control should have the `ShowCaptions` property and the `Flat` property set to `True`. Then you should add as many tool buttons as there are pull-down menus, set their `AutoSize` and `Grouped` properties to `True`, and connect each tool button with the proper pull-down menu using the `MenuItem` property.

note Borland has made available a free TMenuBar component on its Web site (in the Delphi Downloads area). This component connects directly with a MainMenu component, doing all the required settings automatically.[196]

Once more, instead of doing all of these operations at design time, we can automate the creation of as many buttons as requested by the menu, adding more code to the `FormCreate` method:

195 This is still the case today.

196 This extensions has been later integrated in the VCL library. You can now add a menu to the control bar.

```
// create the buttons of the menu toolbar
ToolSize := 0;
for I := MainMenu.Items.Count - 1 downto 0 do
begin
  tb := TToolButton.Create (ToolBarMenu);
  tb.Parent := ToolBarMenu;
  tb.AutoSize := True;
  tb.Grouped := True;
  tb.Caption := MainMenu.Items[I].Caption;
  tb.MenuItem := MainMenu.Items[I];
  Inc (ToolSize, tb.Width);
end;
// size the menu toolbar
ToolBarMenu.Width := ToolSize;
// hide the standard menu, using the form's Menu property
Menu := nil;
```

Notice that you have to disconnect the menu from the form, by removing the value of the form's `Menu` property, which is automatically set as you place the menu component in the form. The result is a menu inside the ControlBar, as you can see in Figure 7.10.

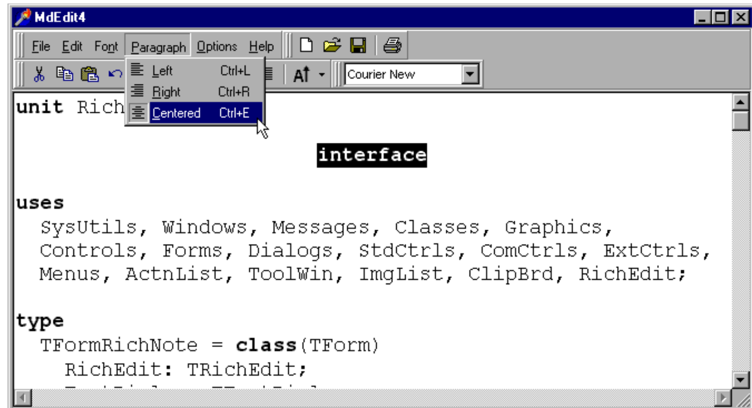


Figure 7.10: The MdEdit4 example shows how to place a menu inside a toolbar based on the ControlBar component. Image from the original book.

Creating a Status Bar

Building a status bar is even simpler than building a toolbar. Delphi includes a specific StatusBar component, based on the corresponding Windows common control. This component can be used almost as a panel when its `SimplePanel` property is set

to `True`. In this case you can use the `SimpleText` property to output some text. The real advantage of this component, however, is that it allows you to define a number of subpanels just by activating the editor of its `Panels` property. (You can also display this property editor by double-clicking on the status bar control.) Each subpanel has its own graphical attributes, which you can customize using the editor. Another feature of the status bar component is the “size grip” area added to the lower-right corner of the bar, which is useful for resizing the form itself. This is a typical element of the Windows user interface, and you can control it with the `SizeGrip` property.

There are a number of uses for a status bar. The most common is to display information about the menu item currently selected by the user. Besides this, a status bar often displays other information about the status of a program: the position of the cursor in a graphical application, the current line of text in a word processor, the status of the lock keys, the time and date, and so on.

Menu Hints in the Status Bar

A new version of the editor, MdEdit5, has a status bar capable of displaying the description of the current menu item, the status of the Caps Lock key, and the current editing position. The StatusBar component of this example has four panels. Although we’re going to display text on only three of them, we need to define the fourth in order to delimit the area of the third panel. The last panel, in fact, is always large enough to cover the remaining surface of the status bar.

To show information on a panel, you simply use its `Text` property, generally using an expression like this:

```
StatusBar1.Panels[1].Text := ‘message’;
```

The panels are not independent components, so you cannot access them by name. A good solution to improve the readability of the program is to define a constant for each panel you want to use, and then use these constants when referring to the panels. The MdEdit5 example defines the following constants:

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

Now we have to populate the panels of the status bar with the proper text. First, we want to display a hint message for the menu items and toolbar buttons. To obtain this effect, you need to take two steps. First, input a string as a `Hint` property of each

action of the ActionList component. This hint will be used both as a fly-by hint for toolbar buttons and as a status bar message when the cursor is over the button or the menu item is selected. Actually, we can use the `Hint` property to specify different strings for the two cases, by writing a string divided into two portions by a separator, the | character. For example, you might enter the following as the value of the `Hint` property:

```
'Help|Activate the help of the application'
```

The first portion of the string, *Help*, is used by fly-by hints, the second portion by the status bar. You can see an example of this effect in Figure 7.11.

note When the hint for a control is made up of two strings, you can use the `GetShortHint` and `GetLongHint` methods to extract the first (short) and second (long) substrings from the string you pass as a parameter, which is usually the value of the `Hint` property.

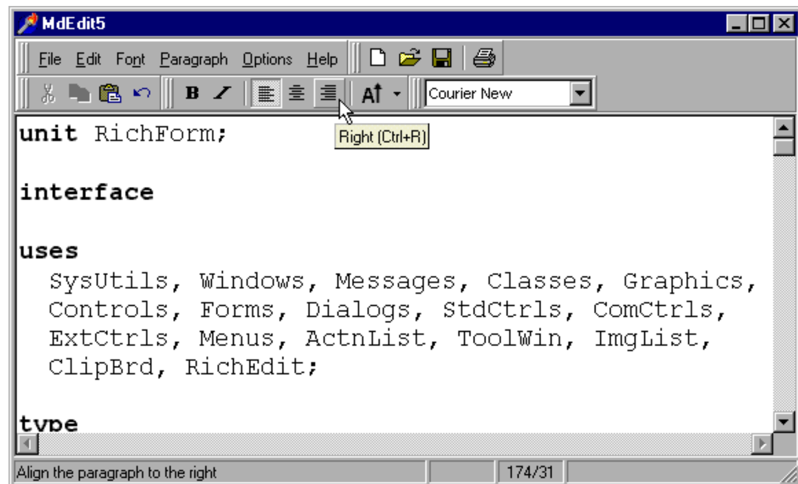


Figure 7.11: The status bar of the MdEdit5 example displays (among other information) a description of the current button or menu item. The two portions of the Hint property are displayed in the status bar and as a fly-by hint. Image from the original book.

To obtain the hint in the status bar, we have to write some code to handle the application's `OnHint` event. To avoid adding a new method to the form manually and then assign it to the `OnHint` event of the `Application` object, we can add to the form the ApplicationEvents component, and handle its event at design time.

The `ShowHint` procedure copies the current value of the application's `Hint` property, which temporarily contains a copy of the selected item's hint, to the status bar:

```
procedure TFormRichNote.ShowHint(Sender: TObject);
begin
```

```
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

This is all you need to do to display a hint indicating the effect of a menu in the status bar.

To display the status of the Caps Lock key, or of any other key, you have to call the `GetKeyState` API function, which returns a state number. If the low-order bit of this number is set (that is, if the number is odd), then the key is pressed. When do we check this state? We can do it every time the user presses a key on the form, when the application is idle, or we can add a timer and make the check every 5 seconds. This second approach has an advantage, because the user might press the Caps Lock key while working with a different application, and this should be indicated on the status bar of our program, too. However, using a timer makes the response to pressing the key quite slow, while speeding up the timer might slow down the program. So I've decided to write a simple procedure, called `CheckCapslock`, and then call it both in the `OnUpdate` event handler of the ActionList component (called when the application has some idle time) and in the `OnTimer` event handler of a timer component I've added to the form:

```
procedure TFormRichNote.CheckCapslock;
begin
  if Odd (GetKeyState (VK_CAPITAL)) then
    StatusBar1.Panels[sbpCaps].Text := 'CAPS'
  else
    StatusBar1.Panels[sbpCaps].Text := '';
end;
```

Finally, the program uses the third panel to display the current cursor position (measured in lines and characters per line) every time the selection changes. Because the `CaretPos` values are zero-based (that is, the upper-left corner is line 0, character 0), I've decided to add one to each value, to make them more reasonable for a casual user:

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ...
  // update the position in the status bar
  StatusBar.Panels[sbpPosition].Text := Format ('%d/%d',
    [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```

Scrolling a Form

When you build a simple application, a single form might hold all of the components you need. As the application grows, however, you may need to squeeze in the components, increase the size of the form, or add new forms.

If you reduce the space occupied by the components, you might add some capability to resize them at run time, possibly splitting the form into different areas. If you choose to increase the size of the form, you might use scroll bars to let the user move around in a form that is bigger than the screen.

note If you choose to add a new form, you can create secondary forms and dialog boxes, create forms with multiple pages, or use the MDI approach (as described in the next chapter).

Adding a scroll bar to a form is simple. In fact, you don't need to do anything. If you place a number of components in a big form and then reduce its size, a scroll bar will be added to the form automatically, as long as you haven't changed the value of the `AutoScroll` property from its default of `True`.

Along with `AutoScroll`, forms have two properties, `HorzScrollBar` and `VertScrollBar`, which can be used to set several properties of the two `TFormScrollBar` objects associated with the form. The `Visible` property indicates whether the scroll bar is present, the `Position` property determines the initial status of the scroll thumb, and the `Increment` property determines the effect of clicking one of the arrows at the ends of the scroll bar. The most important property, however, is `Range`.

The `Range` property of a scroll bar determines the virtual size of the form in one direction, not the actual range of values of the scroll bar. At first, this might be somewhat confusing. Here is an example to clarify how the `Range` property works. Suppose you need a form that will host a number of components and will therefore need to be 1000 pixels wide. We can use this value to set the "virtual range" of the form, changing the range of the horizontal scroll bar. See Figure 7.12 for an illustration of the virtual size of a form implied by the range of a scroll bar. If the width of the client area of the form is smaller than 1000 pixels, a scroll bar will appear. Now you can start using it at design time to add new components in the "hidden" portion of the form.

The `Position` property of the scroll bar ranges from 0 to 1000 minus the current size of the client area. For example, if the client area of the form is 300 pixels wide, you can scroll 700 pixels to see the far end of the form (the thousandth pixel).

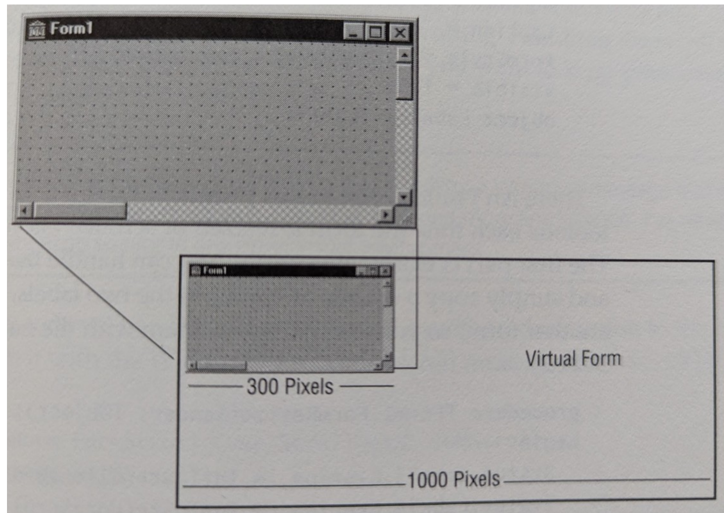


Figure 7.12: A representation of the virtual size of a form implied by the range of a scroll bar. Image based on a picture of the original printed book.

The Scroll Testing Example

I've built an example, Scroll1, which has a virtual form of 1000 pixels. To accomplish this, I simply set the range of the horizontal scroll bar to 1000:

```
object Form1: TForm1
  Width = 458
  Height = 368
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  Caption = 'Scrolling Form'
  OnResize = FormResize
  ...
```

The form of this example has been filled with a number of meaningless list boxes, and I could have obtained the same scroll bar range by placing the rightmost list box so that its position (`Left`) plus its size (`Width`) would equal 1000.

The interesting part of the example is the presence of a toolbox window displaying the status of the form and of its horizontal scroll bar. This second form has four labels; two with fixed text and two with the actual output. Besides this, the secondary form (called `Status`) has a `bsToolWindow` border style and is a topmost window. You should also set its `Visible` property to `True`, to have its window automatically displayed at startup:

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  Caption = 'Status'
  FormStyle = fsStayOnTop
  Visible = True
  object Label1: TLabel...
  ...
```

There isn't much code in this program. Its aim is to update the values in the toolbox each time the form is resized or scrolled (as you can see in Figure 7.13). The first part is extremely simple. You can handle the `OnResize` event of the form and simply copy a couple of values to the two labels. The labels are part of another form, so you need to prefix them with the name of the form instance, `Status`:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

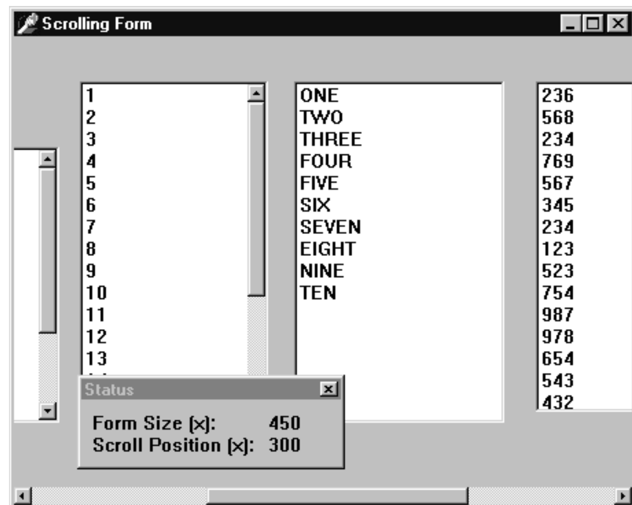


Figure 7.13: The output of the Scroll1 example. Image from the original book.

If we wanted to change the output each time the user scrolls the contents of the form, we could not use a Delphi event handler, because there isn't an `OnScroll` event for forms (although there is one for stand-alone ScrollBar components). Omitting this event makes sense, because Delphi forms handle scroll bars automatically in a powerful way. In Windows, by contrast, scroll bars are extremely low-level elements, requiring a lot of coding. Handling the scroll event makes sense only in

special cases, such as when you want to keep track precisely of the scrolling operations made by a user.

note Once again, what I really like in Delphi is that handling a Windows message that is not supported by the environment requires only one more line of code. I've never seen something so nice in any other visual environment.

Here is the code we need to write. First, add a method declaration to the class and associate it with the Windows horizontal scroll message (wm_HScroll):

```
  public
    procedure FormScroll (var ScrollData: TWMScroll);
      message wm_HScroll;
```

Then write the code of this procedure, which is almost the same as the code of the FormResize method we've seen before:

```
procedure TForm1.FormScroll (var ScrollData: TWMScroll);
begin
  inherited;
  Status.Label3.Caption := IntToStr(ClientWidth);
  Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

It's important to add the call to inherited, which activates the method related to the same message in the base class form. The inherited keyword in Windows message handlers calls the method of the base class we are overriding, which is the one associated with the corresponding Windows message (even if the procedure name is different). Without this call, the form won't have its default scrolling behavior; that is, it won't scroll at all.

Automatic Scrolling

The scroll bar's Range property can seem strange until you start to use it consistently. When you think about it a little, you'll start to understand the advantages of the "virtual range" approach. First of all, the scroll bar is automatically removed from the form when the client area of the form is big enough to accommodate the virtual size; and when you reduce the size of the form, the scroll bar is added again.

This feature becomes particularly interesting when the AutoScroll property of the form is set to True. In this case, the extreme positions of the rightmost and lower controls are automatically copied into the Range properties of the form's two scroll bars. Automatic scrolling works well in Delphi. In the last example, the virtual size

of the form would be set to the right border of the last list box. This was defined with the following attributes:

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

Therefore, the horizontal virtual size of the form would be `977` (which is the sum of the two above values). This number is automatically copied into the `Range` field of the `HorzScrollBar` property of the form, unless you change it manually to have a bigger form (as I've done for the Scroll1 example, setting it to 1000 to leave some space between the last list box and the border of the form). You can see this value in the Object Inspector, or make the following test: Run the program, size the form as you like, and move the scroll thumb to the rightmost position. When you add the size of the form and the position of the thumb, you'll always get 1000, the virtual coordinate of the rightmost pixel of the form, whatever the size.

Scrolling and Form Coordinates

We have just seen that forms can automatically scroll their components. But what happens if you paint directly on the surface of the form? Some problems arise, but their solution is at hand. Suppose that we want to draw some lines on the virtual surface of a form, as shown in Figure 7.14.

Since you probably do not own a monitor capable of displaying 2000 pixels on each axis, you can create a smaller form, add two scroll bars, and set their `Range` property, as I've done in the Scroll2 example. Here is the textual description of the form:

```
object Form1: TForm1
  HorzScrollBar.Range = 2000
  VertScrollBar.Range = 2000
  ClientHeight = 336
  ClientWidth = 472
  OnPaint = FormPaint
end
```

If we simply draw the lines using the virtual coordinates of the form, the image won't display properly. In fact, in the `OnPaint` response method, we need to compute the virtual coordinates ourselves. Fortunately, this is easy, since we know that the virtual `X1` and `Y1` coordinates of the upper-left corner of the client area correspond to the current positions of the two scroll bars:

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

```
var
  X1, Y1: Integer;
begin
  X1 := HorzScrollBar.Position;
  Y1 := VertScrollBar.Position;

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30-X1, 30-Y1);
  Canvas.LineTo (1970-X1, 1970-Y1);
// and so on ...
```

As a better alternative, instead of computing the proper coordinate for each output operation, we can call the `SetWindowOrgEx` API to move the origin of the coordinates of the `Canvas` itself. This way, our drawing code will directly refer to virtual coordinates but will be displayed properly:

```
procedure TForm2.FormPaint(Sender: TObject);
begin
  SetWindowOrgEx (Canvas.Handle,
    HorzScrollbar.Position,
    VertScrollbar.Position, nil);

  // draw a yellow line
  Canvas.Pen.Width := 30;
  Canvas.Pen.Color := clYellow;
  Canvas.MoveTo (30, 30);
  Canvas.LineTo (1970, 1970);

  // and so on ...
  ...
```

This is the version of the program you'll find in the source code you've downloaded. Try using the program and commenting out the `SetWindowOrgEx` call to see what happens if you don't use virtual coordinates: You'll find that the output of the program is not correct—it won't scroll, and the same image will always remain in the same position, regardless of scrolling operations.

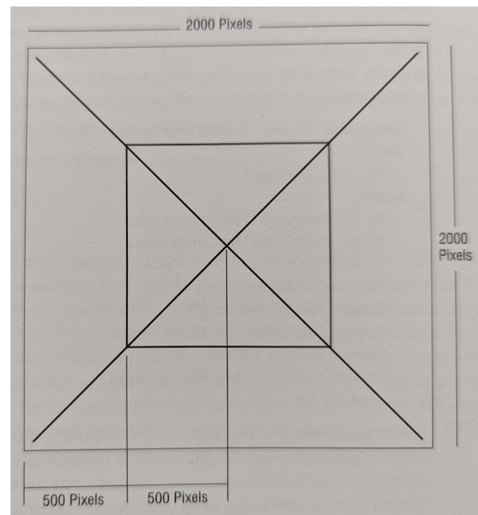


Figure 7.14: The lines to draw on the virtual surface of the form. Image based on a picture of the original printed book.

Form-Splitting Techniques

There are several ways to implement form-splitting techniques in Delphi, but the simplest approach is to use the Splitter component, found in the Additional page of the Component Palette. To make it more effective, the splitter can be used in combination with the `Constraints` property of the controls it relates to. As we'll see in the Split1 example, this allows us to define maximum and minimum positions of the splitter and of the form.

To build this example, simply place a ListBox component in a form; then add a Splitter component, a second ListBox, another Splitter, and finally a third ListBox component. The form also has a simple toolbar based on a panel.

By simply placing these two splitter components, you give your form the complete functionality of moving and sizing the controls it hosts at run time. The `Width`, `Beveled`, and `Color` properties of the splitter components determine their appearance, and in the Split1 example you can use the toolbar controls to change them. Another relevant property is `MinSize`, which determines the minimum size of the components of the form. During the splitting operation (see Figure 7.15), a line marks the final position of the splitter, but you cannot drag this line over a certain limit. The behavior of the Split1 program is not to let controls become too small. An

alternative technique is to set the new `AutoSnap` property of the splitter to `True`. This property will make the splitter hide the control when its size goes below the `MinSize` limit.

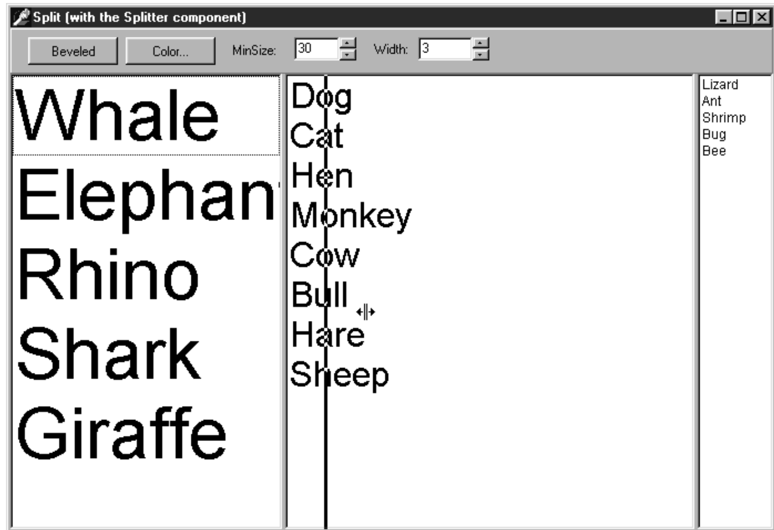


Figure 7.15: The splitter component of the Split1 example determines the minimum size for each control on the form, even those not adjacent to the splitter itself. Image from the original book.

I suggest you try using the Split1 program, so that you'll fully understand how the splitter affects its adjacent controls and the other controls of the form.

Even if I've set the `MinSize` property, a user of this program can reduce the size of its entire form to a minimum, hiding some of the list boxes. If you test the Split2 version of the example, instead, you'll get better behavior. In Split2 I've set some `Constraints` for the ListBox controls, as for example:

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

The size constraints are applied only as you actually resize the controls, so to make this program work in a satisfactory way, you have to set the `ResizeStyle` property of the two splitters to `rsUpdate`. This value indicates that the position of the controls is updated for every movement of the splitter, not only at the end of the operation. If you select the `rsLine` or the new `rsPattern` values, instead, the splitter simply draws a line in the required position, checking the `MinSize` property but not the constraints of the controls.

note The Splitter component in Delphi 5 has a new property, `AutoSnap`. When you set this to `True`, the splitter will completely hide the neighboring control when the size of that control is below the minimum set for it in the Splitter component.

Horizontal Splitting

The Splitter component can also be used for horizontal splitting, instead of the default vertical splitting. However, this approach is a little more complicated. Basically you can place a component on a form, align it to the top, and then place the splitter on the form. By default, it will be left-aligned. Choose the `alTop` value for the `Align` property, and then resize the component manually, by changing the `Height` property in the Object Inspector (or by resizing the component).

You can see a form with a horizontal splitter in the SplitH example. This program has two memo components you can open a file into, and it has a splitter dividing them, defined as:

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

When you double-click on a memo, the program loads a text file into it (notice the structure of the `with` statement):

```
procedure TForm1.MemoDblClick(Sender: TObject);
begin
  with Sender as TMemo, OpenDialog1 do
    if Execute then
      Lines.LoadFromFile (FileName);
end;
```

The program features a status bar, which keeps track of the current height of the two memo components. It handles the `OnMoved` event of the splitter (the only event of this component) to update the text of the status bar. The same code is executed whenever the form is resized:

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format (
    'Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

You can see the effect of this code by looking at Figure 7.16, or by running the SplitH example.

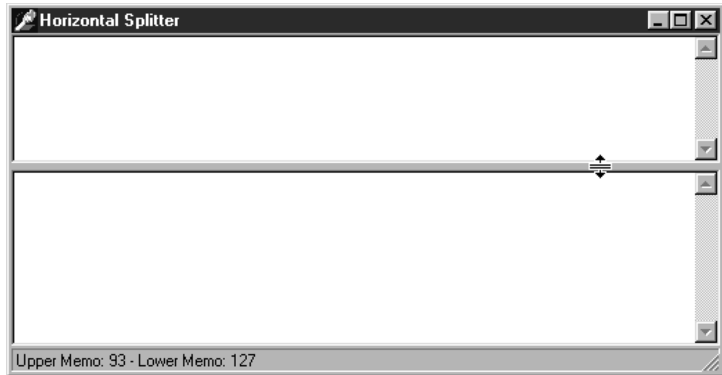


Figure 7.16: The status bar of the SplitH example indicates the position of the horizontal splitter component.

Splitting with a Header

An alternative to using splitters is to use the standard HeaderControl compo-nent. If you place this control on a form, it will be automatically aligned with the top of the form. Then you can add the three list boxes to the rest of the client area of the form. The first list box can be aligned on the left, but this time you cannot align the second and third list box as well. The problem is that the sections of the header can be dragged outside the visible surface of the form. If the list boxes use automatic alignment, they cannot move outside the visible surface of the form, as the program requires.

The solution is to define the sections of the header, using the specific editor of the `Sections` property. This property editor allows you to access the various subobjects of the collection, changing various settings. You can set the caption and alignment of the text; the current, minimum, and maximum size of the header; and so on. Set-ting the limit values is a powerful tool, and it replaces the `MinSize` property of the splitter or the constraints of the list boxes we've used in past examples. You can see the output of this program, named HdrSplit, in Figure 7.17.

We need to handle two events: `OnSectionResize` and `OnSectionClick`. The first handler simply resizes the list box connected with the modified section (determined by associating numbers with the `ImageIndex` property of each section and using it to determine the name of the list box control):

```
procedure TForm1.HeaderControl1SectionResize(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    Section.ImageIndex)) as TListBox;
  List.Width := Section.Width;
end;
```

Figure 7.17: The output of the HdrSplit example. Image from the original book.

Along with this event, we need to handle the resizing of the form, using it to synchronize the list boxes with the sections, which are all resized by default:

```
procedure TForm1.FormResize(Sender: TObject);
var
  I: Integer;
  List: TListBox;
begin
  for I := 0 to 2 do
  begin
    List := FindComponent ('ListBox' + IntToStr (
      HeaderControl1.Sections[I].ImageIndex)) as TListBox;
    List.Left := HeaderControl1.Sections[I].Left;
    List.Width := HeaderControl1.Sections[I].Width;
  end;
end;
```

After setting the height of the list boxes, this method simply calls the previous one, passing parameters that we won't use in this example. The second method of the

HeaderControl, called in response to a click on one of the sections, is used to sort the contents of the corresponding list box:

```
procedure TForm1.HeaderControl1SectionClick(
  HeaderControl: THeaderControl; Section: THeaderSection);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    Section.ImageIndex)) as TListBox;
  List.Sorted := not List.Sorted;
end;
```

Of course, this code doesn't provide the common behavior of sorting the elements when you click on the header and then sorting them in the reverse order if you click again. To implement this, you should write your own sorting algorithm. Finally, the HdrSplit example uses a new feature for the header control. It sets the `DragReorder` property to enable dragging operations to reorder the header sections. When this operation is performed, the control fires the `OnSectionDrag` event, where you can exchange the positions of the list boxes. This event fires before the sections are actually moved, so I have to use the coordinates of the other section:

```
procedure TForm1.HeaderControl1SectionDrag(Sender: TObject;
FromSection,
  ToSection: THeaderSection; var AllowDrag: Boolean);
var
  List: TListBox;
begin
  List := FindComponent ('ListBox' + IntToStr (
    FromSection.ImageIndex)) as TListBox;
  List.Left := ToSection.Left;
  List.Width := ToSection.Width;

  List := FindComponent ('ListBox' + IntToStr (
    ToSection.ImageIndex)) as TListBox;
  List.Left := FromSection.Left;
  List.Width :=fromSection.Width;
end;
```

Control Anchors

In this chapter I've described how you can use alignment and splitters to create nice and flexible user interfaces, which adapt to the current size of the form, giving users the maximum freedom. Delphi also supports right and bottom anchors. Before this

feature was introduced in Delphi 4, every control placed on a form had coordinates relative to the top and bottom sides, unless it was aligned to the bottom or right sides. Aligning is good for some controls but not all of them, particularly buttons.

By using anchors, you can make the position of a control relative to any side of the form. For example, to have a button anchored to the bottom-right corner of the form, place the button in the required position and set its `Anchors` property to `[akRight, akBottom]`. When the form size changes, the distance of the button from the anchored sides is kept fixed. In other words, if you set these two anchors and remove the two defaults, the button will remain in the bottom-right corner.

On the other hand, if you place a large component such as a Memo or a ListBox in the middle of a form, you can set its `Anchors` property to include all four sides. This way the control will behave as an aligned control, growing and shrinking with the size of the form, but there will be some margin between it and the form sides.

note Anchors, like constraints, work both at design time and at run time; so you should set them up as early as possible, to benefit from this feature while you're designing the form as well as at run time.

As an example of both approaches, you can try out the Anchors application, which has two buttons on the bottom-right corner and a list box in the middle. As shown in Figure 7.18, the controls automatically move and stretch as the form size changes. To make this form work properly, you must also set its `Constraints` property; otherwise, as the form becomes too small the controls can overlap or disappear.

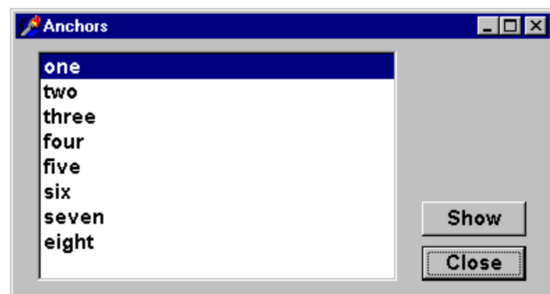


Figure 7.18: The controls of the Anchors example move and stretch automatically as the user changes the size of the form. No code is needed to move the controls, only a proper use of the Anchors property. Image from the original book.

note If you remove all of the anchors, or two opposite ones (for example, left and right), the resize operations will cause the control to float. The control keeps its current size, and the system adds or removes the same number of pixels on each side of it. This can be defined as a centered anchor, because if the component is initially in the middle of the form it will keep that position. In any case, if you want a centered control, you should generally use both opposite anchors, so that if the user makes the form larger the control size will grow as well. In the case just presented, in fact, making the form larger leaves a small control in its center.

Docking Toolbars and Controls

Another feature added in Delphi 4 was the support for *dockable* toolbars and controls[197]. In other words, you can create a toolbar and move it to any of the sides of a form, or even move it freely on the screen, undocking it. However, setting up a program properly to obtain this effect is not as easy as it sounds.

First of all, Delphi's docking support is connected with container controls, not with forms. A panel, a ControlBar, and other containers (technically, any control derived from `TWinControl`) can be set up as dock targets by enabling their `DockSite` property. You can also set the `AutoSize` property of these containers, so that they'll show up only if they actually hold a control.

To be able to drag a control (an object of any `TControl`-derived class) into the dock site, simply set its `DragKind` property to `dkDock` and its `DragMode` property to `dmAutomatic`. This way, the control can be dragged away from its current position into a new docking container. To undock a component and move it to a special form, you can set its `FloatingDockSiteClass` property to `TCustomDockForm` (to use a predefined stand-alone form with a small caption).

All the docking and undocking operations can be tracked by using special events of the component being dragged (`OnStartDock` and `OnEndDock`) and the component that will receive the docked control (`OnDragOver` and `OnDragDrop`). These docking events are very similar to the dragging events available in earlier versions of Delphi.

There are also commands you can use to accomplish docking operations in code and to explore the status of a docking container. Every control can be moved to a different location using the `Dock`, `ManualDock`, and `ManualFloat` methods. A container has a `DockClientCount` property, indicating the number of docked controls, and a `DockClients` property, with the array of these controls.

197 Docking remains a core feature of the VCL library and the Delphi IDE uses it heavily.

Moreover, if the dock container has the `UseDockManager` property set to `True`, you'll be able to use the `DockManager` property, which implements the `IDockManager` interface. This interface has many features you can use to customize the behavior of a dock container, even including support for streaming its status.

As you can see from this brief description, docking support in Delphi is based on a large number of properties, events, methods and objects (such as dock zones and dock trees)—more features than we have room to explore in detail. The next example introduces the main features you'll generally need.

Docking Toolbars in ControlBars

The MdEdit6 example is the final version of the RichEdit editor presented in this chapter. This new version has a second ControlBar at the bottom of the form, which accepts dragging one of the toolbars in the ControlBar at the top. Since both toolbar containers have the `AutoSize` property set to `True`, they are automatically removed when the host contains no controls. To let users drag the toolbars with the same anchor used for moving them inside the container, remember to set the `AutoDrag` property of the ControlBars, as well.

You can see an example of the program at run time in Figure 7.19. The components inside the control bar at the top have their `DragKind` property set to `dkDock`. However, the menu toolbar cannot be moved outside of its container, because we want to keep it close to the typical position of a menu bar. The combo box can be dragged, but we don't want to let a user dock it in the lower control bar. We implement the second constraint in the control bar's `OnDockOver` event handler, by accepting the docking operation only for toolbars:

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := Source.Control is TToolbar;
end;
```

Figure 7.19: The MdEdit6 example allows you to dock the toolbars (and the menu) at the top or bottom of the form or to leave them floating. Image from the original book.

Next, we want to have a border for the lower control bar, but only when it hosts some components, so that we don't see the empty border (as the control bar resizes itself to a very thin line when it is empty). To accomplish this, we can add the border whenever a control is dropped onto the bar (`OnDockDrop`) and remove it when the last control is being undocked (`OnUnDock`). To determine the number of controls, we can use the `DockClientCount` property, which is updated after the undocking is completed, so its value is still 1 when the last control is being undocked:

```
procedure TFormRichNote.ControlBarLowerDockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  ControlBarLower.BevelKind := bkTile;
end;

procedure TFormRichNote.ControlBarLowerUnDock(Sender: TObject;
  Client: TControl; NewTarget: TWinControl; var Allow: Boolean);
begin
  if ControlBarLower.DockClientCount = 1 then
    ControlBarLower.BevelKind := bkNone;
end;
```

This excerpt from the form's DFM file shows the properties related to docking support:

```
object FormRichNote: TFormRichNote
  object RichEdit: TRichEdit...
  object ControlBar: TControlBar
    AutoSize = True
    object ToolBarFile: TToolBar
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ToolBarEdit: TToolBar
      DragKind = dkDock
```

```
      DragMode = dmAutomatic
    end
    object ToolBarFont: TToolBar
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ComboFont: TComboBox
      DragKind = dkDock
      DragMode = dmAutomatic
    end
    object ToolBarMenu: TToolBar...
  end
  object StatusBar: TStatusBar...
  object ControlBarLower: TControlBar
    BevelKind = bkNone
    OnDockDrop = ControlBarLowerDockDrop
    OnDockOver = ControlBarLowerDockOver
    OnUnDock = ControlBarLowerUnDock
  end  ...
end
```

note When you move one of the toolbars to the automatically created floating form, you might be tempted to set it back by closing the floating form. This doesn't work, as the floating form is removed along with the toolbar it contains. However, you can use the shortcut menu of the top-most ControlBar to show this hidden toolbar.

Controlling Docking Operations

Delphi provides many events and methods that give you a lot of control over docking operations, including a dock manager. To explore some of these features, try out the DockTest example, a test bed for docking operations. The program assigns the `FloatingDockSiteClass` property of a Memo component to `TForm2`, so that you can design specific features and add them to the floating frame that will host the control when it is floating, instead of using an instance of the default `TCustomDockForm` class.

Another feature of the program is that it handles the `OnDockOver` and `OnDockDrop` events of a dock host panel to display messages to the user, such as the number of controls currently docked:

```
procedure TForm1.Panel1DockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  Caption := 'Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

In the same way, the program also handles the main form's docking events. Another control, a list box, has a shortcut menu you can invoke to perform docking and undocking operations in code, without the usual mouse dragging:

```
procedure TForm1.DocktoPanel1Click(Sender: TObject);
begin
  // dock to the panel
  ListBox1.ManualDock (Panel1, Panel1, alBottom);
end;

procedure TForm1.DocktoForm1Click(Sender: TObject);
begin
  // dock to the current form
  ListBox1.Dock (Self, Rect (200, 100, 100, 100));
end;

procedure TForm1.Floating1Click(Sender: TObject);
begin
  // toggle the floating status
  if ListBox1.Floating then
    ListBox1.ManualDock (Panel1, Panel1, alBottom)
  else
    ListBox1.ManualFloat (Rect (100, 100, 200, 300));
  Floating1.Checked := ListBox1.Floating;
end;
```

The final feature of the example is probably the most interesting one: Every time the program closes, it saves the current docking status of the panel, using the dock manager support. When the program is reopened, it reapplies the docking information, restoring the previous configuration of the windows. The program does this only with the panel, so the other floating windows will be displayed in their original positions. Here is the code for saving and loading:

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  FileStr: TFileStream;
begin
  if Panel1.DockClientCount > 0 then
  begin
    FileStr := TFileStream.Create (DockFileName,
      fmCreate or fmOpenWrite);
    try
      Panel1.DockManager.SaveToStream (FileStr);
    finally
      FileStr.Free;
    end;
  end
  else
    // remove the file
    DeleteFile (DockFileName);
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FileStr: TFileStream;
begin
  // reload the settings
  DockFileName := ExtractFilePath (Application.Exename) +
    'dock.dck';
  if FileExists (DockFileName) then
  begin
    FileStr := TFileStream.Create (DockFileName, fmOpenRead);
    try
      Panel1.DockManager.LoadFromStream (FileStr);
    finally
      FileStr.Free;
    end;
  end;
  Panel1.DockManager.ResetBounds (True);
end;
```

There are many more features you can test, but the DockTest program already tries to do too many things, some of which conflict. For example, automatic alignments don't work terribly well with the docking manager's code for restoring. I suggest you take this program and explore its behavior, extending it to support the type of user interface you prefer.

note Remember that although docking panels make an application look nice, some users get confused by the fact that their toolbars might disappear or be in a different position than they are used to. Don't overuse the docking features, or some of your inexperienced users may get lost.

What's Next?

In this chapter, we have examined a series of topics related to toolbars and forms: the definition of a toolbar and a status bar; and ways to scroll, split, and drag forms. Although these may seem very diverse topics, they all relate to the development of a modern user interface for a form.

You can consider this chapter the first step toward building professional applications. We will take other steps in the following chapters; but you already know enough to make your programs similar to some best-selling Windows applications, which may be very important for your clients. Now that the elements of the main form of our programs are properly set up, we can consider adding secondary forms and dialog boxes. This is the topic of the next chapter, although we have already

seen how simple it is to add a second form to a program to build a toolbox. In the next chapter we'll also explore multiple-page forms, another important addition to the toolkit of any developer who wants to create a modern user interface.

Chapter 8: Using Multiple Forms

Up to this point, most of the programs in this book have consisted of single forms. Usually, applications have a main window, some floating toolboxes or palettes, and a number of dialog boxes that can be invoked through menu commands or command buttons. More complex applications might have an MDI structure—a frame window with a number of child windows inside its client area. The development of MDI applications will be discussed briefly at the end of this chapter, after we focus on building dialog boxes and applications with multiple forms.

Dialog Boxes versus Forms

Before presenting examples of applications with multiple forms or user-defined dialog boxes, let me begin with a general description of these two alternatives. When you write a program, there is really no big difference between a dialog box and a

second form, aside from the border, the border icons, and other user-interface elements you can customize.

What users associate with a dialog box is the concept of a *modal window*—a window that takes the focus and must be closed before the user can move back to the main window. This is true for message boxes and usually for dialog boxes, as well. However, you can also have nonmodal—or *modeless*—dialog boxes. So if you think that dialog boxes are just modal forms, you are on the right track, but your description is not precise. In Delphi (as in Windows), you can have modeless dialog boxes and modal forms. We have to consider two different elements:

- The form's border and its user interface determine whether it looks like a dialog box.
- The use of two different methods (`Show` or `ShowModal`) to display the second form determines its behavior (modeless or modal).

Adding a Second Form to a Program

To add a second form to an application, you simply click on the New Form button on the Delphi toolbar or use the File ➢ New Form menu command. As an alternative you can select File ➢ New, move to the Forms or Dialogs page, and choose one of the available form templates or form wizards.

If you have two forms in a project, you can use the Select Form or the Select Unit button of the Delphi toolbar to navigate through them at design time. You can also choose which form is the main one and which forms should be automatically created at start-up using the Forms page of the Project Options dialog box. This information is reflected in the source code of the project file.

note Secondary forms are automatically created in the project source-code file depending on a new Delphi 5 setting, which is the Auto Create Forms check box of the Preferences page of the Environment Options dialog box. Although automatic creation is the simplest and most reliable approach for novice developers and quick-and-dirty projects, I suggest that you disable this check box for any serious development. When your application contains hundreds of forms, you really shouldn't have them all created at application start-up. Create instances of secondary forms when and where you need them, and free them when you're done.

Once you have prepared the secondary form, you can simply set its `Visible` property to `True`, and both forms will show up as the program starts. In general, the secondary forms of an application are left "invisible" and are then displayed by calling the `Show` method (or setting the `Visible` property at run time). If you use the

`Show` function, the second form will be displayed as modeless, so you can move back to the first one while the second is still visible. To close the second form, you might use its system menu or click a button or menu item that calls the `Close` method. As we saw in Chapter 6, the default close action (see the `OnClose` event) for a secondary form is simply to hide it, so the secondary form is not destroyed when it is closed. It is kept in memory (again, not always the best approach) and is available if you want to show it again[198].

Creating Secondary Forms at Run Time

Unless you create the forms when the program starts, you'll need to check whether a form exists and create it if necessary. The simplest case is when you want to create multiple copies of the same form at run time. In the MultiWin example, I've done this by writing the following code:

```
procedure TForm1.btnMultipleClick(Sender: TObject);
begin
  with TForm3.Create (Application) do
    Show;
end;
```

Every time you click the button, a new copy of the form is created. Notice that I don't use the `Form3` global variable, because it doesn't make much sense to assign this variable a new value every time you create a new form object. The important thing, however, is not to refer to the global `Form3` object in the code of the form itself or in other portions of the application. The `Form3` variable, in fact, will invariably be a pointer to `nil`, so you should actually remove it from the unit to avoid any confusion.

note In the code of a form, you should never explicitly refer to the form by using the global variable that Delphi sets up for it. For example, suppose that in the code of `TForm3` you refer to `Form3.Caption`. If you create a second object of the same type (the class `TForm3`), the expression `Form3.Caption` will invariably refer to the caption of the form object referenced by the `Form3` variable, which might not be the current object executing the code. To avoid this problem, refer to the `Caption` property in the form's method to indicate the caption of the current form object, and use the `Self` keyword when you need a specific reference to the object of the current form. To avoid any problem when creating multiple copies of a form, I suggest removing the global form object from the interface portion of the unit declaring the form. This global variable is required only for the automatic form creation.

198 All of this still applies 100% today.

When you create multiple copies of a form dynamically, remember to destroy each form object as is it closed, by handling the corresponding event:

```
procedure TForm3.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Failing to do so will result in a lot of memory consumption, because all the forms you create (both the windows and the Delphi objects) will be kept in memory and simply hidden from view.

Now let us focus on the dynamic creation of a form, in a program that accounts for only one copy of the form at a time. Creating a modal form is quite simple, because the dialog box can be destroyed when it is closed, with code like this:

```
procedure TForm1.btnModalClick(Sender: TObject);
var
  Modal: TForm4;
begin
  Modal := TForm4.Create (Application);
  try
    Modal.ShowModal;
  finally
    Modal.Free;
  end;
end;
```

Because the `ShowModal` call can raise an exception, you should write it in a `finally` block to make sure the object will be deallocated. Usually this block also includes code that initializes the dialog box before displaying it and code that extracts the values set by the user before destroying the form. The final values are read-only if the result of the `ShowModal` function is `mrOK`, as we'll see in the next example.

The situation is a little more complex when you want to display only one copy of a modeless form. In fact, you have to create the form, if it is not already available, and then show it:

```
procedure TForm1.btnSingleClick(Sender: TObject);
begin
  if not Assigned (Form2) then
    Form2 := TForm2.Create (Application);
  Form2.Show;
end;
```

With this code the form is created the first time it is required and then is kept in memory, visible on the screen or hidden from view. To avoid using up memory and

system resources unnecessarily, you'll want to destroy the secondary form when it is closed. You can do that by writing a handler for the `OnClose` event:

```
procedure TForm2.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
  // important: set pointer to nil!
  Form2 := nil;
end;
```

Notice that after we destroy the form, the global `Form2` variable is set to `nil`. Without this code, closing the form would destroy its object, but the `Form2` variable would still refer to the original memory location. At this point, if you try to show the form once more with the `btnSingleClick` method shown earlier, the `if not Assigned` test will succeed, as it simply checks whether the `Form2` variable is `nil`. The code fails to create a new object, and the `Show` method, invoked on a nonexistent object, will result in a system memory error.

As an experiment, you can generate this error by removing the last line of the listing above. As we have seen, the solution is to set the `Form2` object to `nil` when the object is destroyed, so that properly written code will "see" that a new form has to be created before using it. Again, experimenting with the MultiWin example can prove useful to test various conditions. I haven't illustrated any screens from this example because the forms it displays are quite bare (totally empty except for the main form, which has three buttons).

note Setting the form variable to `nil` makes sense—and works—if there is to be only one instance of the form present at any given instant. If you want to create multiple copies of a form, you'll have to use other techniques to keep track of them. Also keep in mind that in this case we cannot use the new Delphi 5 `FreeAndNil` procedure, because we cannot call Free on `Form2`. The reason is that we cannot destroy the form before its event handlers have finished executing.

Merging Form Menus

Another feature of modeless forms is worth mentioning. Although every form of an application can have its own menu bar, you can also use Delphi's menu merging technique to move the items of the secondary form's menu to the main form's menu bar. This technique is very useful in MDI applications but less interesting for modeless forms, as this behavior can confuse the user.

In this technique, the application's main window has a menu bar, as usual. The other forms have a menu bar with the `AutoMerge` property enabled, so their menu bar won't be displayed in the form but will instead be merged with the one from the main window. These are the rules for menu merging: Each pull-down menu has a `GroupIndex` property. When menu bars are merged, the pull-down menus are arranged as follows:

- If two elements of the different menu bars have the same `GroupIndex`, those of the original menu are removed.
- Elements are ordered by ascending `GroupIndex` values.

Creating a Dialog Box

I stated earlier in this chapter that a dialog box is not very different from other forms. There is a very simple trick to build a dialog box instead of a form. Just select the `bsDialog` value for the form's `BorderStyle` property. With this simple change, the interface of the form becomes like that of a dialog box, with no system icon, no Minimize or Maximize boxes, and a system menu you can activate by right-clicking over the caption. Of course, such a form has the typical thick dialog box border, which is nonresizable.

Once you have built a dialog box form, you can display it as a modal or modeless window using the two usual show methods (`Show` and `ShowModal`). Modal dialog boxes, however, are more common than modeless ones. This is exactly the reverse of forms; modal forms should generally be avoided since a user won't expect them. The following table lists the complete schema of the various combinations of styles:

Window Type	Modal	Modeless
Form	Never used	Usual, in SDI applications
Dialog box	Most common kind of secondary form	Used, but not very common

To avoid using too many secondary forms, you can build multipage forms, as discussed later in this chapter. Another alternative is to use MDI forms, also covered later in this chapter.

The Dialog Box of the RefList Example

In Chapter 5 we explored the RefList program, which used a ListView control to display references to books, magazines, Web sites, and more. In the RefList2 version I'll simply add to the basic version of that program a dialog box, used in two different circumstances: adding new items to the list and editing existing items. You can see the form of the dialog box in Figure 8.1 and its textual description in the following listing (detailed because it has many interesting features, so I suggest you read this code with care):

```
object FormItem: TFormItem
  Caption = 'Item'
  Color = clBtnFace
  Position = poScreenCenter
  object Label1: TLabel
    Caption = '&Reference:'
    FocusControl = EditReference
  end
  object EditReference: TEdit...
  object Label2: TLabel
    Caption = '&Type:'
    FocusControl = ComboType
  end
  object ComboType: TComboBox
    Style = csDropDownList
    Items.Strings = (
      'Book'
      'CD'
      'Magazine'
      'Mail Address'
      'Web Site')
  end
  object Label3: TLabel
    Caption = '&Author:'
    FocusControl = EditAuthor
  end
  object EditAuthor: TEdit...
  object Label4: TLabel
    Caption = '&Country:'
    FocusControl = EditCountry
  end
  object EditCountry: TEdit...
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

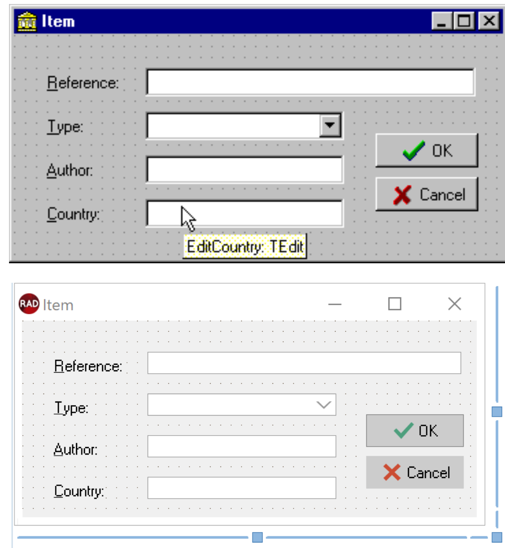


Figure 8.1: The form of the dialog box of the RefList2 example at design time. Images captured in Delphi 5 and Delphi 12.

note The items of the combo box in this dialog describe the available images of the image list, so that a user can select the type of the item and the system will show the corresponding glyph. An even better option would have been to show those glyphs in the combo box, along with their descriptions.

As I mentioned, this dialog box is used in two different cases. The first takes place as the user selects File ➢ Add Items from the menu:

```
procedure TForm1.AddItems1Click(Sender: TObject);
var
  NewItem: TListItem;
begin
  FormItem.Caption := ‘New Item’;
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
    NewItem := ListView1.Items.Add;
    NewItem.Caption := FormItem.EditReference.Text;
    NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
    NewItem.SubItems.Add (FormItem.EditAuthor.Text);
    NewItem.SubItems.Add (FormItem.EditCountry.Text);
  end;
end;
```

Besides setting the proper caption of the form, this procedure needs to initialize the dialog box, as we are entering a brand-new value. If the user clicks OK, however, the

program adds a new item to the list view and sets all its values. To empty the edit boxes of the dialog, the program calls the custom `Clear` method, which resets the text of each edit box control:

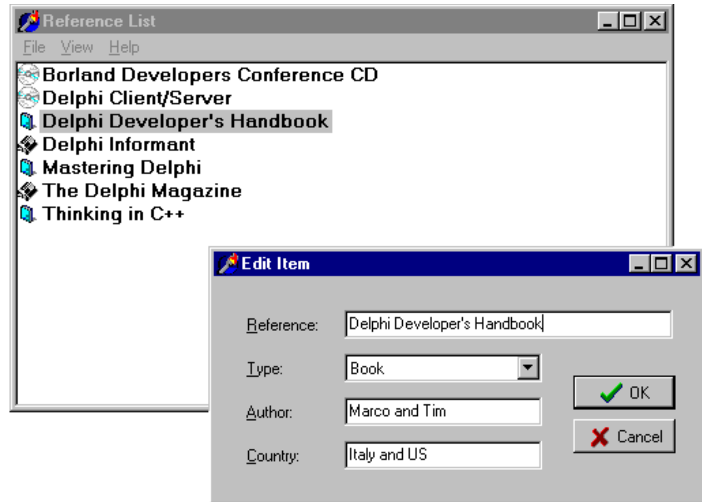
```
procedure TFormItem.Clear;
var
  I: Integer;
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
    if Controls [I] is TEdit then
      TEdit (Controls[I]).Text := '';
end;
```

Editing an existing item requires a slightly different approach. First, the current values are moved to the dialog box before it is displayed. Second, if the user clicks OK, the program modifies the current list item instead of creating a new one. Here is the code:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
begin
  if ListView1.Selected <> nil then
  begin
    // dialog initialization
    FormItem.Caption := 'Edit Item';
    FormItem.EditReference.Text := ListView1.Selected.Caption;
    FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
    FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
    FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

    // show it
    if FormItem.ShowModal = mrOK then
    begin
      // read the new values
      ListView1.Selected.Caption := FormItem.EditReference.Text;
      ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
      ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
      ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
    end;
  end;
end;
```

You can see the effect of this code in Figure 8.2. Notice that the code used to read the value of a new item or modified one is similar. In general, you should try to avoid this type of duplicated code and possibly place the shared code statements in a method added to the dialog box. In this case, the method could receive as parameter a `TListItem` object and copy the proper values into it.

Figure 8.2: The dialog box of the RefList2 example used in edit mode

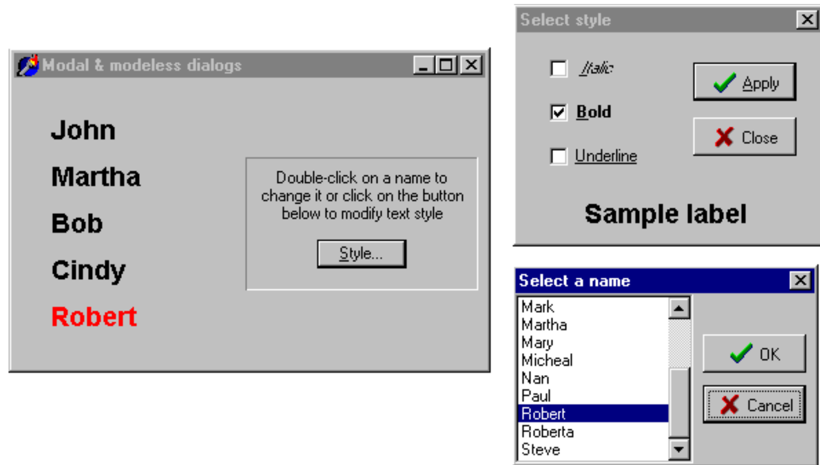


note What happens internally when the user clicks on the OK or Cancel buttons of the dialog box? A modal dialog box is closed by setting its `ModalResult` property, and it returns the value of this property. You can indicate the return value by setting the `ModalResult` property of the button. When the user clicks on the button, its `ModalResult` value is copied to the form, which closes the form and returns the value as the result of the `ShowModal` function.

A Modeless Dialog Box

The second example of dialog boxes shows a more complex modal dialog box that uses the standard approach as well as a modeless dialog box. The main form of the DlgApply example has five labels with names, as you can see in Figure 8.3 and by viewing the source code you've downloaded.

Figure 8.3: The three forms (a main form and two dialog boxes) of the DlgApply example at run time. Images from the original book.



If the user clicks on a name, its color turns to red; if the user double-clicks on it, the program displays a modal dialog box with a list of names to choose from. If the user clicks on the Style button, a modeless dialog box appears, allowing the user to change the font style of the main form's labels. The five labels of the main form are connected to two methods, one for the `OnClick` event and the second for the `OnDoubleClick` event. The first method turns the last label a user has clicked on to red, resetting to black all the others (which have the `Tag` property set to `1`, as a sort of group index). Notice that the same method is associated with all of the labels:

```
procedure TForm1.LabelClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ComponentCount - 1 do
   if (Components[I] is TLabel) and
       (Components[I].Tag = 1) then
     TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;
```

The second method common to all of the labels is the handler of the `OnDoubleClick` event. The `LabelDoubleClick` method selects the `Caption` of the current label (indicated by the `Sender` parameter) in the list box of the dialog and then shows the modal dialog box. If the user closes the dialog box by clicking on OK and an item of the list is selected, the selection is copied back to the label's caption:

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

```
begin
  with ListDial.Listbox1 do
  begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
      // copy the selected item to the label
      (Sender as TLabel).Caption := Items [ItemIndex];
end;
```

note Notice that all the code used to customize the modal dialog box is in the `LabelDoubleClick` method of the main form. The form of this dialog box has no added code.

The modeless dialog box, by contrast, has a lot of coding behind it. The main form simply displays the dialog box when the Style button is clicked (notice that the button caption ends with three dots to indicate that it leads to a dialog box), by callings its `Show` method. You can see the dialog box running in Figure 8.3 above.

Two buttons, Apply and Close, replace the OK and Cancel buttons in a modeless dialog box. (The fastest way to obtain these buttons is to select the `bkOK` or `bkCancel` value for the `Kind` property and then edit the `Caption`.) At times, you may see a Cancel button that works as a Close button, but the OK button in a modeless dialog box usually has no meaning. Instead, there might be one or more buttons that perform specific actions on the main window, such as Apply, Change Style, Replace, Delete, and so on.

If the user clicks on one of the check boxes of this modeless dialog box, the style of the sample label's text at the bottom changes accordingly. You accomplish this by adding or removing the specific flag that indicates the style, as in the following `OnClick` event handler:

```
procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style :=
      LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style :=
      LabelSample.Font.Style - [fsItalic];
end;
```

When the user selects the Apply button, the program copies the style of the sample label to each of the form's labels, rather than considering the values of the check boxes:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
```

```
begin
  Form1.Label1.Font.Style := LabelSample.Font.Style;
  Form1.Label2.Font.Style := LabelSample.Font.Style;
  ...
```

As an alternative, instead of referring to each label directly, you can look for it by calling the `FindComponent` method of the form, passing the label name as a parameter, and then casting the result to the `TLabel` type. The advantage of this approach is that we can create the names of the various labels with a `for` loop:

```
procedure TStyleDial.ApplyBitBtnClick(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 5 do
    (Form1.FindComponent ('Label' + IntToStr (I)) as TLabel).
       Font.Style := LabelSample.Font.Style;
end;
```

note The `ApplyBitBtnClick` method could also be written by scanning the `Controls` array in a loop, as I've already done in other examples. I decided to use the `FindComponent` method, instead, to show you a new technique.

This second version of the code is certainly slower, because it has more operations to do, but you won't notice the difference, because it is very fast anyway. Of course, this second approach is also more flexible; if you add a new label, you only need to fix the higher limit of the `for` loop, provided all the labels have consecutive numbers. Notice that when the user clicks on the Apply button, the dialog box does not close. Only the Close button has this effect. Consider also that this dialog box needs no initialization code because the form is not destroyed, and its components maintain their status each time the dialog box is displayed.

Windows Common Dialogs

Besides building your own dialog boxes, Delphi allows you to use some default dialog boxes of different kinds. Some are predefined by Windows, others are simple dialog boxes (such as message boxes) displayed by a Delphi routine. The Delphi Component Palette contains a page of dialog box components. Each of these dialog

boxes—known as *Windows common dialogs*—is defined in the system library `ComDlg32.DLL`[199].

I have already used some of these dialog boxes in several examples in the previous chapters, so you are probably familiar with them. Basically, you need to put the corresponding component on a form, set some of its properties, run the dialog box (with the `Execute` method, returning a `Boolean` value), and retrieve the properties that have been set while running it. To help you experiment with these dialog boxes, I've built the CommDlg test program. I won't discuss the program in detail nor show its simple but lengthy source code in the book. As always, you can find this code among the downloaded files.

What I want to do is simply highlight some key and nonobvious features of the common dialog boxes, and let you study the source code of the example for the details:

- The Open Dialog Component[200] can be customized by setting different file extensions filters, using the `Filter` property, which has a handy editor and can be assigned directly with a string like `Text File (*.txt)|*.txt`. Another handy feature is to let the dialog check whether the extension of the selected file matches the default extension, by checking the `ofExtensionDifferent` flag of the `Options` property after executing the dialog. Finally, this dialog allows multiple selections by setting its `ofAllowMultiSelect` option. In this case you can get the list of the selected files by looking at the `Files` string list property.
- The SaveDialog component is used in similar ways and has similar properties, although you cannot select multiple files, of course.
- The OpenPictureDialog and SavePictureDialog components provide similar features but have a customized form, which shows a preview of an image. Of course, it makes sense to use them only for opening or saving graphical files.
- The FontDialog component can be used to show and select from all types of fonts, fonts useable on both the screen and a selected printer (wysiwyg), or only TrueType fonts. You can show the portion related to the special effects or hide it,

---

199 Oddly, this is still the name of the library even in the 64-bit version of Windows. All of the ideas and most of the code in this ebook should equally apply to the Win64 target that Delphi offers today. For non-Windows targets, instead, you cannot use the VCL library, but have to switch to the similar FireMonkey library.

200 There are new versions of these dialog boxes available in Windows. They can be enable in Delphi by using Windows themes (the default for new applications), however some of the extended features are available only when using the newer FileOpenDialog and FileSaveDialog components. There is also a new TaskDialog now available in the VCL, mapped to another relatively new Windows API.

and obtain other different versions by setting its `Options` property. You can also activate an Apply button simply by providing an event handler for its `OnApply` event and using the `fdApplyButton` option. A Font dialog box with an Apply button (see Figure 8.4) behaves almost like a modeless dialog box (but isn't one).

- The ColorDialog component is used with different options, to show the dialog fully open at first or to prevent it from opening fully. These settings are the `cdFullOpen` or `cdPreventFullOpen` values of the `Options` property.

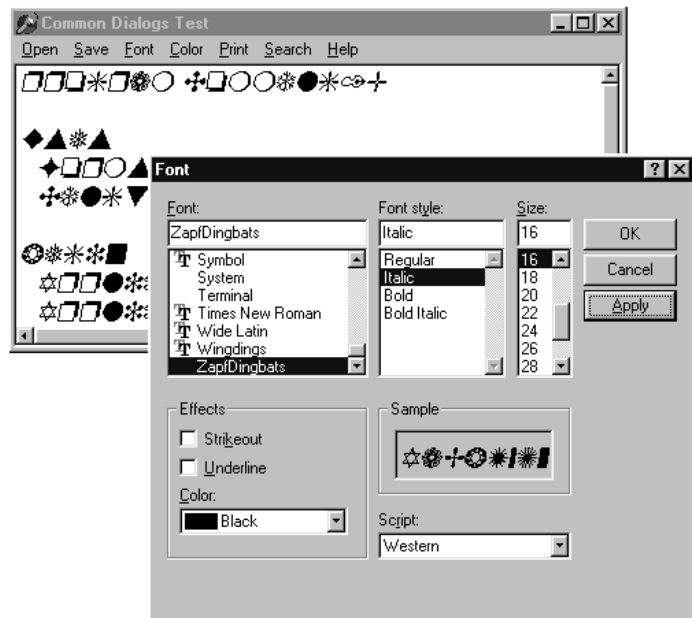


Figure 8.4: The Font selection dialog box with an Apply button. Image from the original book.

- The Find and Replace dialog boxes are truly modeless dialogs, but you have to implement the find and replace functionality yourself, as I've partially done in the CommDlg example. The custom code is connected to the buttons of the two dialog boxes by providing the `OnFind` and `OnReplace` events.

A Parade of Message Boxes

The Delphi message boxes and input boxes are another set of predefined dialog boxes. There are basically six Delphi procedures and functions you can use to display simple dialog boxes[201]:

- The `MessageDlg` function shows a customizable message box, with one or more buttons and usually a bitmap. We have used this function quite often in previous examples.
- The `MessageDlgPos` function is similar to the `MessageDlg` function. The difference is that the message box is displayed in a given position, not in the center of the screen.
- The `ShowMessage` procedure displays a simpler message box, with the application name as the caption, and just an OK button. The `ShowMessageFmt` procedure is a variation of `ShowMessage`, which has the same parameters as the `Format` function. It corresponds to calling `Format` inside a call to `ShowMessage`.
- The `ShowMessagePos` procedure does the same, but you also indicate the position of the message box.
- The `MessageBox` method of the `Application` object allows you to specify both the message and the caption; you can also provide various buttons and features. This is a simple and direct encapsulation of the `MessageBox` function of the Windows API, which passes as a main window parameter the handle of the `Application` object. This handle is required to make the message box behave like a modal window.
- The `InputBox` function asks the user to input a string. You provide the caption, the query, and a default string.

The `InputQuery` function asks the user to input a string, too. The only difference between this and the `InputBox` function is in the syntax. The `InputQuery` function has a `Boolean` return value that indicates whether the user has clicked on OK or Cancel.

To demonstrate some of the message boxes available in Delphi, I've written another sample program, with a similar approach to the preceding CommDlg example. In the MBParade example, you have a high number of choices (radio buttons, check boxes, edit boxes, and spin edit controls) to set before you press one of the buttons

---

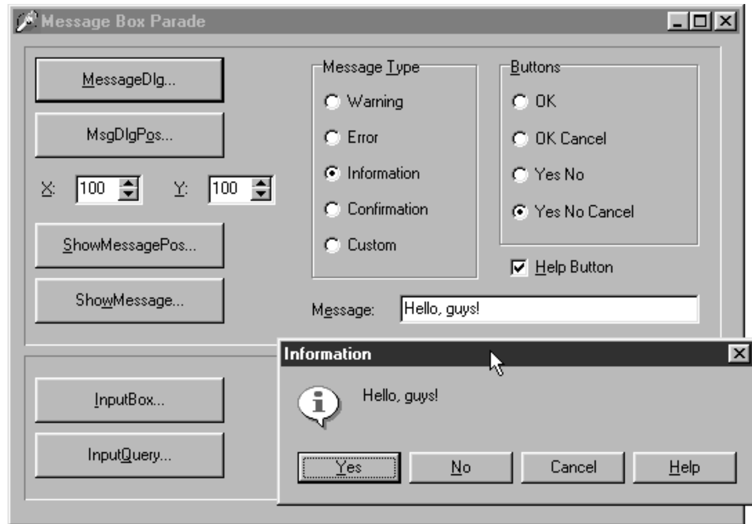
201 As already mentioned in a previous note, there is now also a TaskDialog component in the VCL, mapped to a specific, relatively new, Windows API.

that displays a message box. You can get a better idea of the program by looking at its form in Figure 8.5.

Expandable Dialog Boxes

Some dialog boxes display a number of components for the user to work with. At times, you can divide them into logical pages, which Delphi supports through the PageControl component (discussed later in this chapter). At other times, you can temporarily hide some dialog box controls to help first-time users of your application. Another alternative is to increase the size of the dialog box to host new controls when the user presses a More button[202].

Figure 8.5: The main form of the MBParade example, with a sample message box. Image from the original book.



I'll use this approach to create the simple dialog box in the More example. First of all, we need to create the dialog box and add some simple controls, a More button (see Figure 8.6), and two check boxes labeled *italic* and *bold,* which are in a panel placed outside the design-time surface of the form. In practice, once you have added the panel with some controls in it, you need to resize the dialog box so that the new panel is outside the visible surface of the form and set the `AutoScroll` property of the form to `False`. The panel is not visible, because I've removed its borders, and

202 I have to admin this UI style of expanding the size of a dialog box is way less frequent today.

makes the program more flexible, as you can add more controls to the hidden portion of the dialog without changing the source code: simply place them on the panel.

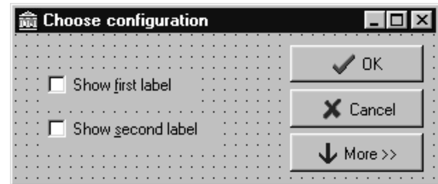


Figure 8.6: The dialog box of the More example at design time. Some of the components are invisible because they are beyond the border. Image from the original book.

The panel should be hidden; otherwise, the user might press the Tab key and move onto its controls even if they are not visible. As an alternative, you might disable its `TabStop` property. These properties (`Visible` or `TabStop`) are then set to `True` when the form is enlarged.

Now, in addition to the standard code required to move values from the main form to the dialog, we need to write some code to resize the form when a user clicks on the More button. To prepare the resizing effect, we need a couple of fields in the form (named `OldHeight` and `NewHeight`) to store the two different heights of the client area of the form. We can set up their values when the form is first created:

```
procedure TConfigureDialog.FormCreate(Sender: TObject);
begin
  OldHeight := ClientHeight;
  NewHeight := PanelMore.Top + PanelMore.Height;
end;
```

I determined the new height by adding to the height of the panel its position. The real dialog box resizing takes place when the More button is pressed. Here is a first version:

```
procedure TConfigureDialog.btnMoreClick(Sender: TObject);
begin
  PanelMore.Visible := True;
  btnMore.Enabled := False;
  ClientHeight := NewHeight;
end;
```

The result it produces is shown in Figure 8.7. If you want a more spectacular effect, you might increase the height a pixel at a time instead of setting the final value at once. If you write a `for` loop, increasing the client height and repainting the form each time, the new controls will appear with a nice effect, only a little slower. The last line of the `btnMoreClick` method above becomes

```
for I := ClientHeight to NewHeight do
begin
  ClientHeight := I;
  Update;
end;
```

Each time the dialog box is activated (`OnFormActivate` event), we reset its height, hide the panel (to avoid letting the user Tab to its controls), and enable the More button:

```
procedure TConfigureDialog.FormActivate(Sender: TObject);
begin
  ClientHeight := OldHeight;
  btnMore.Enabled := True;
  PanelMore.Visible := False;
end;
```

This code is required so that each time the dialog box is displayed it starts in the default small configuration.

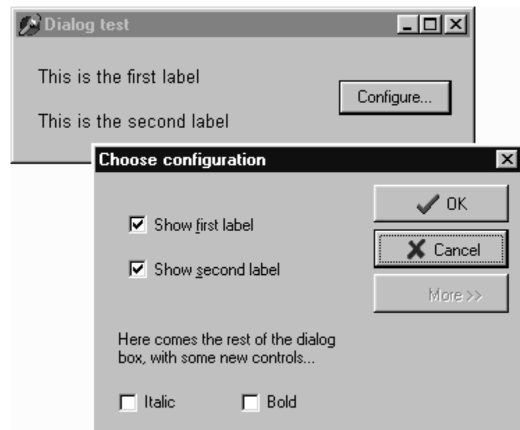


Figure 8.7: The dialog box of the More example after it has been resized. Image from the original book.

About Boxes and Splash Screens

Windows applications usually have an About box, where you can display information, such as the version of the product, a copyright notice, and so on. The simplest

way to build an About box is to use the `MessageDlg` function. With this method, you can show only a limited amount of text and no special graphics.

Therefore, the usual method for creating an About box is to use a simple dialog box, such as the one generated with one of the Delphi default templates. I say *simple* because when you have designed the form with a logo and so on, you seldom need much code. At most, some code might be required to display system information, such as the version of Windows or the amount of free memory, or some user information, such as the registered user name.

note In Chapter 19, we'll see how to create extract the version information from an executable file, which contains this type of Windows resources. This technique can be useful to build an About box that includes the version information.

Building a Custom Hidden Screen

While we build our own About box, we can add a hidden credit screen, which Delphi and many other applications have. You might want to add a hidden credit screen for a number of reasons. If you work in a big company, this might be your way to prove that you worked on that project, which might help you in finding a new job (if the project was successful). At times, a hidden About box can be fun to see, and they sometimes also provide a good occasion for making jokes about your competitors. A more serious reason is that a hidden credit screen can be used to demonstrate who wrote the program, as a sort of legal copyright.

I've written a simple example, showing how you might implement a hidden screen. The dialog box has a Panel component containing two Label components. The panel might contain any number of components to display graphics and text. Some of the strings might even be computed at run time. The only added feature required to show the hidden credits is a PaintBox component covering part of the form.

When the user makes a specific complex action (in this case, right-clicking on the upper label while holding down the Shift key), the panel is hidden and something appears on the screen. A simple solution is to have some text painted on the surface of the form—that is, on its canvas:

```
procedure TAboutBox.Label1MouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if (Button = mbRight) and (ssShift in Shift) then
  begin
    Panel1.Visible := False;
```

```
    PaintBox1.Canvas.Font.Name := 'Arial';
    PaintBox1.Canvas.Font.Size := 20;
    PaintBox1.Canvas.TextOut (40, 50, 'Author: Marco Cantù');
    PaintBox1.Canvas.TextOut (40, 100, 'Version 1.0');
  end;
end;
```

To build a more spectacular hidden screen, we might scroll some text in a `for` loop, as I've done in the final version of the Credits example. Notice that the position of the lines depend on the height of the text, retrieved by calling the `TextHeight` method of the `Canvas` of the PaintBox component:

```
Panel1.Visible := False;
LineH := PaintBox1.Canvas.TextHeight ('0');
for I := 0 to 100 + LineH * 10 do
  with PaintBox1.Canvas do
  begin
    // empty lines are used to delete descendants
    TextOut (40, 100 - I, 'CREDITS example from:');
    TextOut (40, 100 + LineH - I, '"Mastering Delphi"');
    TextOut (40, 100 + LineH * 2 - I, '                          ');
    ...
    // wait 5 milliseconds
    Delay (0, 5);
  end;
Panel1.Visible := True;
```

To avoid a scrolling rate that's too fast, particularly on faster computers, inside the `for` loop I've added a call to a `Delay` procedure, which requires as parameters the seconds and milliseconds you want to wait for. This `Delay` procedure simply checks the current time and then waits in a `while` loop until the required seconds and milliseconds have elapsed:

```
procedure Delay (Seconds, MilliSec: Word);
var
  TimeOut: TDateTime;
begin
  TimeOut := Now + EncodeTime (0,
    Seconds div 60, Seconds mod 60, MilliSec);
  // wait until the TimeOut time
  while Now < TimeOut do
    Application.ProcessMessages;
end;
```

Inside the loop I call the `ProcessMessages` method of the `Application` global object to let Windows generate and dispatch the needed paint messages. This `Delay` procedure is a fairly generic one, so you can use it in other applications quite easily.

note Consider another aspect of the preceding example. We have written some code to draw on the surface of a dialog box. Although it is not very common, dialog boxes can have graphical output and respond to mouse input just like any other form. In fact, a dialog box *is a form*.

Building a Splash Screen

Another typical technique used in applications is to display an initial screen before the main form is shown. This makes the application seem more responsive, because you show something to the user while the program is loading, but it also makes a nice visual effect. Sometimes, this same window is displayed as the application's About box.

For an example in which a splash screen is particularly useful, I've built a program displaying a list box filled with prime numbers. The prime numbers are computed on program start-up, so that they are displayed as soon as the form becomes visible:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to 20000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
end;
```

This method calls an `IsPrime` function I've added to the program. This function, which you can find in the source code, computes prime numbers in a terribly slow way; but I needed a slow form creation to demonstrate my point. The numbers are added to a list box that covers the full client area of the form and allows multiple columns to be displayed, as you can see in Figure 8.8.

As you can see by running the Splash0 example, the problem with this program is that the initial operation, which takes place in the `FormCreate` method, takes a lot of time. When you start the program, it takes several seconds (on a standard Pentium machine[203]) to display the main form. If your computer is very fast or very slow, you can change the upper limit of the `for` loop of the `FormCreate` method to make the program faster or slower.

203 I know this sounds old, but that's the type of CPU in use at the time this book was originally written.

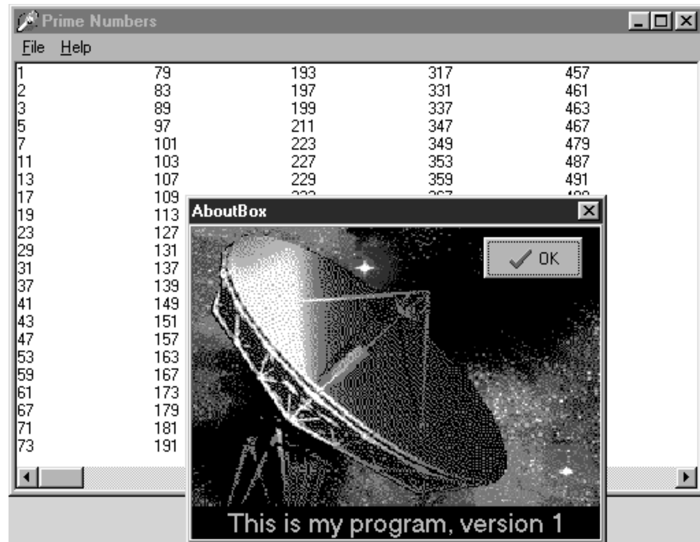


Figure 8.8: The main form of the Splash example, with the About box activated from the menu. Image from the original book.

This program has a simple dialog box with an image component, a simple caption, and a bitmap button, all placed inside a panel taking up the whole surface of the About box. This form is displayed when you select the Help ➢ About menu item. But what we really want is to display this About box while the program starts. You can see this effect by running the Splash1 and Splash2 examples, which show a splash screen using two different techniques.

First of all, I've added a method to the `TAboutBox` class. This method, called `MakeSplash`, changes some properties of the form to make it suitable for a splash form. Basically it removes the border and caption, hides the OK button, makes the border of the panel thick (to replace the border of the form), and then shows the form, repainting it immediately (see Figure 8.9 for the effect):

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
  Show;
  Update;
end;
```

Figure 8.9: The form of the splash screen of the Splash1 example is slightly different than the original About box (shown in Figure 8.8). Image from the original book.

This method is called after creating the form in the project file of the Splash1 example. This code is executed before creating the other forms (in this case only the main form), and the splash screen is then removed before running the application. These operations take place within a `try-finally` block. Here is the source code of the main block of the project file for the Splash2 example:

```
var
  SplashAbout: TAboutBox;

begin
  Application.Initialize;

  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  try
    SplashAbout.MakeSplash;
    // standard code...
    Application.CreateForm(TForm1, Form1);
    // get rid of the splash form
    SplashAbout.Close;
  finally
    SplashAbout.Free;
  end;

  Application.Run;
end.
```

This approach makes sense only if your application's main form takes a while to create, to execute its start-up code (as in this case), or to open database tables. Notice that the splash screen is the first form created, but because the program doesn't use the `CreateForm` method of the `Application` object, this doesn't become the main form of the application. In this case, in fact, closing the splash screen would terminate the program!

An alternative approach is to keep the splash form on the screen a little longer and use a timer to get rid of it after a while. I've implemented this second technique in the Splash2 example. This example also uses a different approach for creating the splash form: instead of creating it in the project source code, it creates the form at the very beginning of the `FormCreate` method of the main form.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
  SplashAbout: TAboutBox;
begin
  // create and show the splash form
  SplashAbout := TAboutBox.Create (Application);
  SplashAbout.MakeSplash;
  // standard code...
  for I := 1 to 20000 do
    if IsPrime (I) then
      ListBox1.Items.Add (IntToStr (I));
  // get rid of the splash form, after a while
  SplashAbout.Timer1.Enabled := True;
end;
```

note This code works properly regardless of the form's creation order, as indicated by the `OldCreateOrder` property (discussed in Chapter 6).

The timer is enabled just before terminating the method. After its interval has elapsed (in the example, 3 seconds) the `OnTimer` event is activated, and the splash form handles it by closing and destroying itself:

```
procedure TAboutBox.Timer1Timer(Sender: TObject);
begin
  Close;
  Release;
end;
```

note The `Release` method of a form is similar to the `Free` method of objects, only the destruction of the form is delayed until all event handlers have completed execution. Using `Free` inside a form might cause an access violation, as the internal code, which fired the event handler, might refer again to the form object.

There is one more thing to fix. The Main form will be displayed later and in front of the splash form, unless you make this a topmost form. For this reason I've added one line to the `MakeSplash` method of the About box in the Splash2 example:

```
FormStyle := fsStayOnTop;
```

Multiple-Page Forms

When you have a lot of information and controls to display in a dialog box or a form, you can use multiple pages. The metaphor is that of a notebook: Using tabs, a user can select one of the possible pages.

There are two controls you can use to built a multiple-page application in Delphi[204]:

- You can use the Windows 95 PageControl component, which has tabs on one of the sides and multiple pages (similar to panels) covering the rest of its surface. As there is one page per tab, you can simply place components on each page to obtain the proper effect both at design time and at run time.
- You can use the TabControl, which has only the tab portion but offers no pages to host the information. In this case you'll want to use one or more components to mimic the *page change* operation.

A third related component, the TabSheet, represents a single page of the PageControl. This is not a stand-alone component and is not available on the Component palette. You create a TabSheet at design time by using the local menu of the PageControl or at run time by using methods of the same control.

note Delphi still includes the Notebook, TabSet, and TabbedNotebook components introduced in earlier versions. Use these components only if you need to create a 16-bit version of an application. For any other purpose, the PageControl and TabControl components, which encapsulate Win32 common controls, provide a more modern user interface. Actually, in 32-bit versions of Delphi, the TabbedNotebook component was reimplemented using the Win32 PageControl internally, to reduce the code size and update the look.

PageControls and TabSheets

As usual, instead of duplicating the Help system's list of properties and methods of the PageControl component, I've built an example that stretches its capabilities and allows you to change its behavior at run time. The example, called Pages, has a PageControl with three pages. The structure of the PageControl and of the other key components is listed below:

204 This is still true today, although Delphi 12 added a new metaphor for hosting MDI or regular forms within a tab-based UI. The new component is called FormTabsBar and it offers a lot of power while simplifying the coding required in VCL to host forms in tabs.

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    HotTrack = True
    Images = ImageList1
    MultiLine = True
    object TabSheet1: TTabSheet
      Caption = 'Pages'
      object Label3: TLabel
      object ListBox1: TListBox
    end
    object TabSheet2: TTabSheet
      Caption = 'Tabs Size'
      ImageIndex = 1
      object Label1: TLabel
      // other controls
    end
    object TabSheet3: TTabSheet
      Caption = 'Tabs Text'
      ImageIndex = 2
      object Memo1: TMemo
        Anchors = [akLeft, akTop, akRight, akBottom]
        OnChange = Memo1Change
      end
      object BitBtnChange: TBitBtn
        Anchors = [akTop, akRight]
        Caption = '&Change'
      end
    end
  end
  object BitBtnPrevious: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Previous'
    OnClick = BitBtnPreviousClick
  end
  object BitBtnNext: TBitBtn
    Anchors = [akRight, akBottom]
    Caption = '&Next'
    OnClick = BitBtnNextClick
  end
  object ImageList1: TImageList
    Bitmap = {...}
  end
end
```

Notice that the tabs are connected to the bitmaps provided by an ImageList control and that some controls use the `Anchors` property to remain at a fixed distance from

the right or bottom borders of the form. Even if the form doesn't support resizing (this would have been far too complex to set up with so many controls), the positions can change when the tabs are displayed on multiple lines (simply increase the length of the captions) or on the left side of the form.

Each TabSheet object has its own `Caption`, which is displayed as the sheet's tab. At design time you can use the local menu to create new pages and to move between pages. You can see the local menu of the PageControl component in Figure 8.10, together with the first page. This page holds a list box and a small caption, and it shares two buttons with the other pages.

If you place a component on a page, it is available only in that page. How can you have the same component (in this case, two bitmap buttons) in each of the pages, without duplicating it? Simply place the component on the form, outside of the PageControl (or before aligning it to the client area) and then move it in front of the pages, calling the Bring to Front command of the form's local menu. The two buttons I've placed in each page can be used to move back and forth between the pages and are an alternative to using the tabs. Here is the code associated with one of them:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.SelectNextPage (True);
end;
```

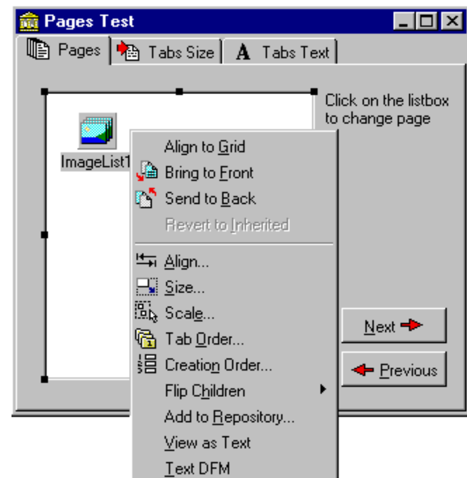


Figure 8.10: The first sheet of the PageControl of the Pages example, with its local menu. Image from the original book.

The other button calls the same procedure, passing `False` as its parameter to select the previous page. Notice that there is no need to check whether we are on the first or last page, because the `SelectNextPage` method considers the last page to be the one before the first and will move you directly between those two pages.

Now we can focus on the first page again. It has a list box, which at run time will hold the names of the tabs. If a user clicks on an item of this list box, the current page changes. This is the third method available to change pages (after the tabs and the Next and Previous buttons). The list box is filled in the `FormCreate` method, which is associated with the `OnCreate` event of the form and copies the caption of each page (the `Page` property stores a list of TabSheet objects):

```
for I := 0 to PageControl1.PageCount - 1 do
  ListBox1.Items.Add (PageControl1.Pages.Caption);
```

When you click on a list item, you can select the corresponding page:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.Pages [ListBox1.ItemIndex];
end;
```

The second page hosts two edit boxes (connected with two UpDown components), two check boxes, and two radio buttons, as you can see in Figure 8.11. The user can input a number (or choose it by clicking on the up and down buttons with the mouse or pressing ↑ or ↓ while the corresponding edit box has the focus), check the boxes and the radio buttons, and then press the Apply button to make the changes:

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
  // set tab width, height, and lines
  PageControl1.TabWidth := StrToInt (EditWidth.Text);
  PageControl1.TabHeight := StrToInt (EditHeight.Text);
  PageControl1.MultiLine := CheckBoxMultiLine.Checked;
  // show or hide the last tab
  TabSheet3.TabVisible := CheckBoxVisible.Checked;
  // set the tab position
  if RadioButton1.Checked then
    PageControl1.TabPosition := tpTop
  else
    PageControl1.TabPosition := tpLeft;
end;
```

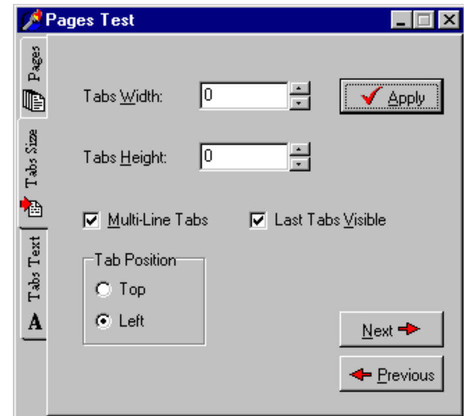


Figure 8.11: The second page of the example can be used to size and position the tabs. Here you can see the tabs on the left of the page control. Image from the original book.

With this code, we can change the width and height of each tab (remember that `0` means the size is computed automatically from the space taken by each string), choose to have either multiple lines of tabs or two small arrows to scroll the tab area, and move them to the left side. The control also allows tabs to be placed on the bottom or on the right; but our program doesn't allow that, because it would make the placement of the other controls quite complex.

You can also hide the last tab on the PageControl, which corresponds to the `TabSheet3` component. If you hide one of the tabs by setting its `TabVisible` property to `False`, you cannot reach that tab by clicking on the Next and Previous buttons, which are based on the `SelectNextPage` method. Instead, you should use the `FindNextPage` function, as shown below in this new version of the Next button's `OnClick` event handler:

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage :=
    PageControl1.FindNextPage (
      PageControl1.ActivePage, True, False);
end;
```

The last page has a memo component, again with the names of the pages (added in the `FormCreate` method). You can edit the names of the pages and press the Change button to change the text of the tabs, but only if the number of strings matches the number of tabs:

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
```

```
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount -1 do
      PageControl1.Pages [I].Caption := Memo1.Lines [I];
  BitBtnChange.Enabled := False;
end;
```

Finally the last button, Add Page, allows you to add a new tab sheet to the page control, although the program doesn't add any components to it. The (empty) tab sheet object is created using the page control as its owner, but it won't work unless you also set the `PageControl` property. Before doing this, however, you should make the new tab sheet visible. Here is the code:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
  begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
  end;
end;
```

note Whenever you write a form based on a PageControl, remember that the first page displayed at run time is the page you were in before the code was compiled. This means that if you are working on the third page and then compile and run the program, it will start with that page. A common way to solve this problem is to add a line of code in the `FormCreate` method to set the PageControl or notebook to the first page. This way, the current page at design time doesn't determine the initial page at run time.

Frames and Pages

When you have a dialog box with many pages full of controls, the code underlying the form becomes very complex because all the controls and methods are declared in a single form. Also, creating all these components (and initializing them) might result in a delay in the display of the dialog box.

The availability of frames in Delphi 5 (see Chapters 1 and 4) can solve both of these issues. First, you can easily divide the code of a single complex form into one frame per page. The form will simply host all of the frames in a PageControl. This certainly helps you to have simpler and more focused units and makes it simpler to reuse a specific page in a different dialog box or application. Reusing a single page of a PageControl without using a frame or an embedded form, in fact, is far from simple.

As an example of this approach I've built the FramePag example, which has some frames placed inside the three pages of a PageControl, as you can see in Figure 8.12. All of the frames are aligned to the client area, using the entire surface of the tab sheet (the page) hosting them[205].

Actually two of the pages have the same frame, but the two instances of the frame have some differences at design time. The frame, called `Frame3` in the example, has a list box that is populated with a text file at start up, has buttons to modify the items in the list and saves them to a file. The filename is placed inside a label, so that you can easily select a file for the frame at design time by changing the `Caption` of the label.

205 The new component I mentioned earlier, FormTabsBar, offers a similar architecture based on the use of regular forms, rather than frames. The result is similar in the two cases, although I personally tend to prefer using forms for tabbed applications when possible, because forms have a few extra methods and events that can be quite handy..

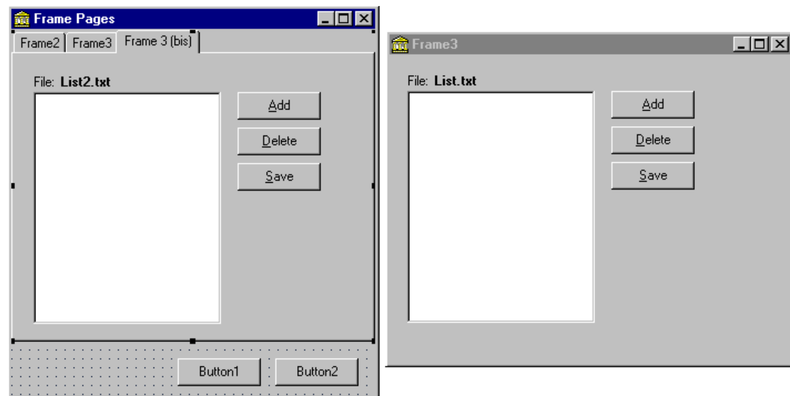


Figure 8.12: Each page of the FramePag example contains a frame, thus separating the code of this complex form into more manageable chunks. Images from the original book.

note Being able to use multiple instances of a frame is one of the reasons this technique was introduced, and customizing the frame at design time is even more important. Because adding properties to a frame and making them available at design time requires some customized and complex code, it is nice to use a component to host these custom values. You have the option of hiding these components (such as the label in our example) if they don't pertain to the user interface.

In the example, we need to load the file when the frame instance is created. Because frames have no `OnCreate` event, our best choice is probably to override the `CreateWnd` method. Writing a custom constructor, in fact, doesn't work as it is executed too early—before the specific label text is available. Here is the frame class code:

```
type
  TFrame3 = class(TFrame)
    ...
  public
    procedure CreateWnd; override;
```

Within the `CreateWnd` method, we simply load the list box content from a file.

Multiple Frames with No Pages

Another approach is to avoid creating all of the pages along with the form hosting them. This can be accomplished by leaving the PageControl empty and creating the frames only when a page is displayed. Actually, when you have frames on multiple pages of a PageControl, the windows for the frames are created only when they are

first displayed, as you can find out by placing a breakpoint in the creation code of the last example.

As an even more radical approach, you can get rid of the page controls and use a TabControl. Used this way, the tab has no connected tab sheets (or pages) but can display only one information at a time. For this reason, we'll need to create the current frame and destroy the previous one or simply hide it by setting its `Visible` property to `False` or by calling the `BringToFront` of the new frame. Although this sounds like a lot of work, in a large application this technique can be worth it for the reduced resource and memory usage you can obtain by applying it.

To demonstrate this approach, I've built an example similar to the previous one, this time based on a TabControl and dynamically created frames. The main form, visible at run time in Figure 8.13, has only a TabControl with one page for each frame:

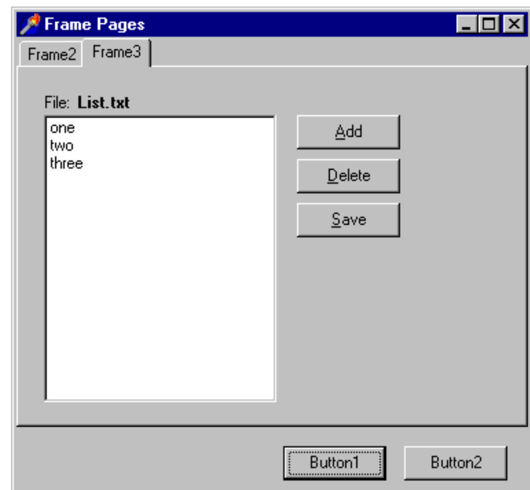


Figure 8.13: The first page of the FrameTab example at run time. The frame inside the tab is created at run time. Image from the original book.

```
object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = [akLeft, akTop, akRight, akBottom]
    Tabs.Strings = (
      'Frame2'
      'Frame3')
    OnChange = TabChange
  end
end
```

I've given each tab a caption corresponding to the name of the frame, because I'm going to use this information to create the new pages. When the form is created, and whenever the user changes the active tab, the program gets the current caption of the tab and passes it to the custom `ShowFrame` method. The code of this method, listed below, checks whether the requested frame already exists (frame names in this example follow the Delphi standard of having a number appended to the class name), and then brings it to the front. If the frame doesn't exist, it uses the frame name to find the related frame class, creates an object of that class, and assigns a few properties to it. The code makes extensive use of class references and dynamic creation techniques (discussed in Chapter 3):

```
type
  TFrameClass = class of TFrame;

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + '1') as TFrame;
  if not Assigned (Frame) then
  begin
    FrameClass := TFrameClass (FindClass ('T' + FrameName));
    Frame := FrameClass.Create (Self);
    Frame.Parent := Tab;
    Frame.Visible := True;
    Frame.Name := FrameName + '1';
  end;
  Frame.BringToFront;
end;
```

To make this code work, you have to remember to add a call to `RegisterClass` in the initialization section of each unit defining a frame.

An Image Viewer with Owner-Draw Tabs

The use of the TabControl and of a dynamic approach, as described in the last example, can also be applied in more general (and simpler) cases. Every time you need multiple pages that all have the same type of content, instead of replicating the controls in each page, you can use a TabControl and change its contents when a new tab is selected.

This is what I'll do in the multiple-page bitmap viewer I'll show in the next example, called TabOnly. The image that appears in the TabControl of this form, aligned to

the whole client area, depends on the selection in the tab above it (as you can see in Figure 8.14).

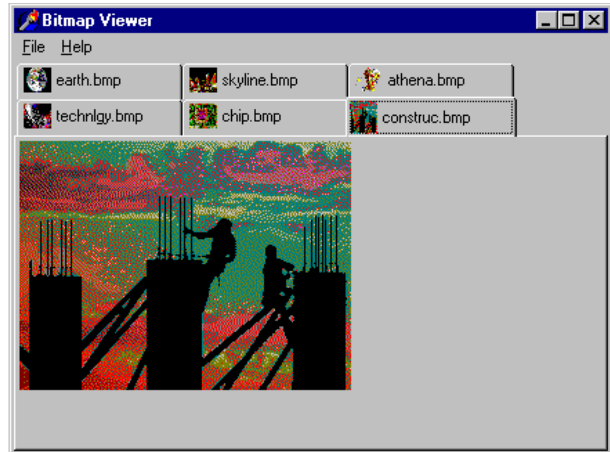


Figure 8.14: The interface of the bitmap viewer in the TabOnly example. Notice the owner-draw tabs. Image from the original book.

At the beginning, the TabControl has only a fake tab describing the situation (*No file selected*). After selecting File ➢ Open, the user can choose a number of files in the File Open dialog box, and the array of strings with the names of the files (the `Files` property of the `OpenDialog1` component) is used as the text for the tabs (the `Tabs` property of `TabControl1`):

```
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs := OpenDialog1.Files;
    TabControl1.TabIndex := 0;
    TabControl1Change (TabControl1);
  end;
end;
```

After we display the new tabs, we have to update the image so that it matches the first tab. To accomplish this, the program calls the method connected with the `OnChange` event of the TabControl, which loads the file corresponding to the current tab in the image component:

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (
    TabControl1.Tabs [TabControl1.TabIndex]);
end;
```

The only special feature of the example is that the TabControl has the `OwnerDraw` property set to `True`. This means that the control won't paint the tabs (which will be empty at design time) but will have the application do this, by calling the `OnDrawTab` event. In its code, the program displays the text vertically centered, using the `DrawText` API function. The text displayed is not the entire file path but only the filename. Then, if the text is not *None,* the program reads the bitmap the tab refers to and paints a small version of it in the tab itself. To accomplish this, the program uses the `TabBmp` object, which is of type `TBitmap` and is created and destroyed along with the form. The program also uses the `BmpSide` constant to position the bitmap and the text properly:

```
procedure TForm1.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
 TabText: string;
 OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle,
    PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)),
    OutRect, dt_Left or dt_SingleLine or dt_VCenter);
  if TabText <> 'None' then
  begin
    TabBmp.LoadFromFile (TabText);
    OutRect.Left := OutRect.Left - BmpSide - 3;
    OutRect.Right := OutRect.Left + BmpSide;
    Control.Canvas.StretchDraw (OutRect, TabBmp);
  end;
end;
```

This example works, unless you select a file that doesn't contain a bitmap. The program will warn the user with a standard exception, ignore the file, and continue its execution.

The User Interface of a Wizard

Just as you can use a TabControl without pages, you can also take the opposite approach and use a PageControl without tabs. What I want to focus on now is the development of the user interface of a wizard. In a wizard, you are directing the user through a sequence of steps, one screen at a time, and at each step you typically want to offer the choice of proceeding to the next step or going back to correct input

entered in a previous step. So instead of tabs that can be selected in any order, wizards typically offer Next and Back buttons to navigate. This won't be a complex example; its purpose is just to give you a few guidelines. The example is called WizardUI.

The starting point is to create a series of pages in a PageControl and set the `TabVisible` property of each TabSheet to `False` (while keeping the `Visible` property set to `True`). Contrary to what happened with past versions, in Delphi 5 you can now hide the tabs also at design time. In this case you'll need to use the shortcut menu of the page control or the combo box of the Object Inspector to move to a another page, instead of the tabs. But why don't you want to see the tabs at design time? So you can place controls on the pages and then place extra controls in front of the pages (as I've done in the example), without seeing their relative positions change at run time. You might also want to remove the useless captions of the tabs, which take up some space in memory and in the resources of the application.

In the first page, I've placed on one side an image and a bevel control and on the other side some text, a check box, and two buttons. Actually, the Next button is inside the page, while the Back button is over it (and is shared by all the pages). You can see this first page at design time in Figure 8.15. The following pages look similar, with a label, check boxes, and buttons on the right side and nothing on the left.

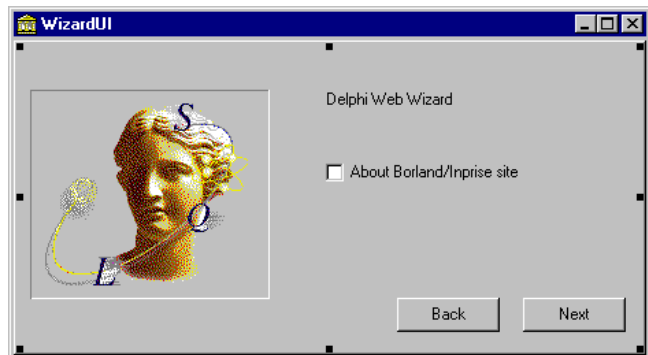


Figure 8.15: The first page of the WizardUI example at design time. Image from the original book.

When you click the Next button on the first page, the program looks at the status of the check box and decides which page is the following one. I could have written the code like this:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  BtnBack.Enabled := True;
  if CheckInprise.Checked then
```

```
    PageControl1.ActivePage := TabSheet2
  else
    PageControl1.ActivePage := TabSheet3;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

After enabling the common Back button, the program changes the active page and finally moves the graphical portion to the new page. Because this code has to be repeated for each button, I've placed it in a method after adding a couple of extra features. This is the actual code:

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  if CheckInprise.Checked then
    MoveTo (TabSheet2)
  else
    MoveTo (TabSheet3);
end;

procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
  // add the last page to the list
  BackPages.Add (PageControl1.ActivePage);
  BtnBack.Enabled := True;
  // change page
  PageControl1.ActivePage := TabSheet;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

Besides the code I've already explained, the `MoveTo` method adds the last page (the one before the page change) to a list of visited pages, which behaves like a stack. In fact, the `BackPages` object of the `TList` class is created as the program starts and the last page is always added to the end. As the user presses the Back button, which is not dependent on the page, the program extracts the last page from the list, deletes its entry, and moves to that page:

```
procedure TForm1.btnBackClick(Sender: TObject);
var
  LastPage: TTabSheet;
begin
  // get the last page and jump to it
  LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
  PageControl1.ActivePage := LastPage;
  // delete the last page from the list
  BackPages.Delete (BackPages.Count - 1);
  // eventually disable the back button
```

```
  BtnBack.Enabled := not (BackPages.Count = 0);
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

With this code, the user can move back several pages until the list is empty, at which point we disable the Back button. The complication we need to deal with is that while moving from a particular page, we know which pages are its "next" and "previous," but we don't know which page we came from, because there can be multiple paths to reach a page. Only by keeping track of the movements with a list can we reliably go back.

The rest of the code of the program, which simply shows some Web site addresses, is very simple. The good news is that you can reuse the navigational structure of this example in your own programs and modify only the graphical portion and the content of the pages.

Docking to a PageControl

Another interesting feature of page controls is the specific support for docking. As you dock a new control over a PageControl, a new page is automatically added to host it, as you can easily see in the Delphi environment. To accomplish this, you simply set the PageControl as a dock host and activate docking for the client controls. This works best when you have secondary forms you want to host. Moreover, if you want to be able to move the entire PageControl into a floating window and then dock it back, you'll need a docking panel in the main form.

This is exactly what I've done in the DockPage example, which has a main form with the following settings:

```
object Form1: TForm1
  Caption = 'Docking Pages'
  object Panel1: TPanel
    Align = alLeft
    AutoSize = True
    DockSite = True
    OnMouseDown = Panel1MouseDown
    object PageControl1: TPageControl
      ActivePage = TabSheet1
      Align = alClient
      DockSite = True
      DragKind = dkDock
      object TabSheet1: TTabSheet
        Caption = 'List'
```

```
        object ListBox1: TListBox
          Align = alClient
        end
      end
    end
  end
  object Splitter1: TSplitter
    Cursor = crHSplit
  end
  object Memo1: TMemo
    Align = alClient
  end
end
```

Notice that the Panel has the `UseDockManager` property set to `True` and that the PageControl invariably hosts a page with a list box, as removing all pages apparently causes problems. Now the program has two other forms, with similar settings (although they host different controls):

```
object Form2: TForm2
  Caption = 'Small Editor'
  DragKind = dkDock
  DragMode = dmAutomatic
  object Memo1: TMemo
    Align = alClient
  end
end
```

You can drag these forms onto the page control to add new pages to it, with captions corresponding with the form titles. You can also undock each of these controls and even the entire PageControl. To do this, the program doesn't enable automatic dragging, which would make it impossible to switch pages anymore. Instead, the feature is activated when the user clicks on the area of the PageControl that has no tabs—that is, on the underlying panel:

```
procedure TForm1.Panel1MouseDown(Sender: TObject;
  Button: TmouseButton; Shift: TShiftState; X, Y: Integer);
begin
  PageControl1.BeginDrag (False, 10);
end;
```

You can test this behavior by running the DockPage example, although Figure 8.16 tries to depict it. Notice that when you remove the PageControl from the main form,

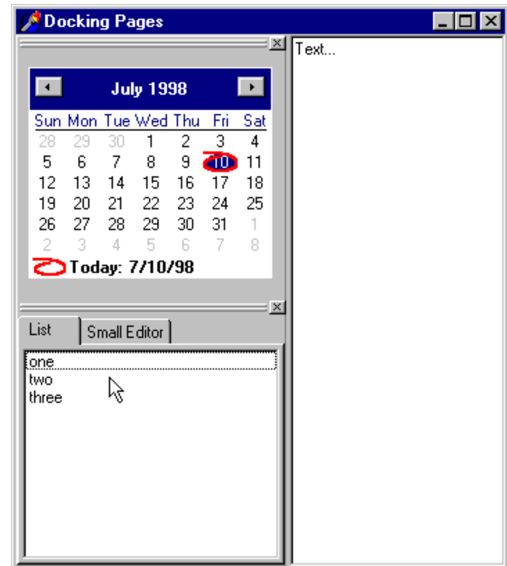


Figure 8.16: The main form of the DockPage example after a form has been docked to the page control on the left. Notice that another form uses part of the area of a hosting panel. Image from the original book.

you can directly dock the other forms to the panel and then split the area with other controls. This is the situation captured by the figure.

Creating MDI Applications

Besides using dialog boxes, or secondary forms, and squeezing components into a form, there is a third approach that used to be common in Windows applications: MDI (*Multiple Document Interface*). An MDI application is made up of a number of forms that appear inside a single main form[206].

If you use Windows Notepad, you can open only one text document, because Notepad isn't an MDI application[207]. But with your favorite word processor, you can

---

206 The MDI has long been phased out of the modern UI toolkit and Microsoft has in practical terms deprecated this model, as they've neglected fixing bugs in Windows when using MDI on HighDPI screens. Given that MDI has remained fairly popular among Delphi developes, Delphi 12 added renewed support for it VCL styled applications and also HighDPI applications (overcoming some of the platform issues). There was also the addition of the FormTabsBar component to help modernize the UI of MDI applications with the addition of tabs.

207 Microsoft introduced a multi tab UI for Notepad, just recently.

probably open a number of different documents, each in its own child window, because they are MDI applications. All these document windows are usually held by a *frame,* or *application,* window.

In Windows 3 and 3.1, Microsoft stressed the use of MDI. With the advent Windows 95, Microsoft had to acknowledge that many users were not comfortable with this interface. Office 2000 is the first large applications suite that drops the MDI model for the SDI (*Single Document Interface*) model used by Windows Resource Explorer and the entire operating system. MDI isn't dead and can be a useful model at times, but multipage and dockable forms seems to be more popular now.

MDI in Windows: A Technical Overview

This section provides a short overview of MDI, in technical Windows terms. Just forget Delphi for a moment, and I'll try to give you an idea of what MDI really is (not what an MDI application looks like). If you've never built an MDI application and you want a quick start, you might consider skipping this section for now.

The MDI structure gives programmers a number of benefits automatically. For example, Windows handles a list of the child windows in one of the pull-down menus of an MDI application, and there are specific Delphi methods that activate the corresponding MDI functionality, to tile or cascade the child windows. The following is the technical structure of an MDI application in Windows[208]:

- The main window of the application acts as a frame or a container. This window requires a proper menu structure and some specific coding (at least when programming with the API).
- A special window, known as the *MDI client,* covers the whole client area of the frame window, providing some special capabilities. For example, the MDI client handles the list of child windows. Although this might seem strange at first, the MDI client is one of the Windows predefined controls, just like an edit box or a list box. The MDI client window does not have the typical elements of the interface of a window, such as a caption or border, but it is visible. In fact, you can change the standard system color of the MDI work area (called the "Application Background") in the Appearance page of the Display Properties dialog box in Windows.

208 This is still the case today. Most of the platform and Delphi MDI features have seen very limited changes over the years, until Delphi 12, as mentioned in an earlier footnote.

- There are a number of child windows, of the same kind or of different kinds. These child windows are not placed in the frame window directly, but each is defined as a child of the MDI client window, which in turn is a child of the frame window. (We might say that the child windows are the "grandchildren" of the frame.)

When you program using the Windows API, some work is usually required to build and maintain this structure, and other coding is needed to handle the menu properly. As you'll see in the next section, these tasks become much easier with Delphi.

Frame and Child Windows in Delphi

Delphi makes the development of MDI applications easy, even without using the MDI Application template available in Delphi (see the Applications page of the File ➢ New dialog box). You only need to build at least two forms, one with the `FormStyle` property set to `fsMDIForm` and the other with the same property set to `fsMDIChild`. That's all, almost. Simply set these two properties in a simple program and run it, and you'll see the two forms nested in the typical MDI style.

Generally, however, the child form is not created at start-up, and you need to provide a way to create one or more child windows. This can be done by adding a menu with a New menu item and writing the following code:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  ChildForm := TChildForm.Create (Application);
  ChildForm.Show;
end;
```

In the code fragment above, as well as in the program example I'll discuss shortly, I've named the two forms `MainForm` and `ChildForm`. Another important feature is to add a "Window" pull-down menu and use it as the value of the `WindowMenu` property of the form. This pull-down menu will automatically list all the available child windows. Of course, you can choose any other name for the menu item, but "Window" is the standard.

With these simple operations, you can build a simple MDI application. To make this program work properly, we can add a number to the title of any child window when it is created:

```
procedure TMainForm.New1Click(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  WindowMenu := Window1;
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption := ChildForm.Caption + ' ' +
    IntToStr (Counter);
  ChildForm.Show;
end;
```

You can also open a number of child windows, minimize or maximize each of them, close them, and use the Window pull-down menu to navigate among them. If you create more than nine child windows, a More Windows menu item is added to the pull-down menu; when you select this menu item, you'll see a dialog box (provided by Windows, not part of your program) with a complete list of the child windows.

Now suppose that we want to close some of these child windows, to unclutter the client area of our program. Click on the Close box of some of the child windows and they are minimized! What is happening here? Remember that when you close a window, you generally hide it from view. The closed forms in Delphi still exist, although they are not visible. In the case of child windows, simply hiding them won't work, because the MDI Window menu and the list of windows will still list existing child windows, even if they are hidden. For this reason, Delphi simply minimizes the MDI child windows when you try to close them. To solve this problem, we need to delete the child windows when they are closed, setting the `Action` reference parameter of the `OnClose` event to `caFree`.

Building a Complete Window Menu

Our first task is to define a better menu structure for the example. Typically the Window pull-down menu has at least three items, titled Cascade, Tile, and Arrange Icons. To handle the menu commands, we can use some of the predefined methods that are available in forms that have the `fsMDIForm` value for the `FormStyle` property:

- The `Cascade` method cascades the open MDI child windows. The child forms are arranged starting from the upper-left corner of the client area of the frame windows and moving toward the lower-left corner. The windows overlap each other. Iconized child windows are also arranged (see `ArrangeIcons` below).

- The `Tile` method tiles the open MDI child windows. The child forms are arranged so that they do not overlap. The client area of the frame windows is divided into equal portions for the different windows, so that they can all be shown on the screen, no matter how many windows there are. The `Tile` method will also arrange iconized child windows. The default behavior is horizontal tiling, although if you have several child windows, they will be arranged in several columns. This default can be changed by using the `TileMode` property.
- The `TileMode` property determines how the `Tile` procedure should work. The only two choices are `tbHorizontal`, for horizontal tiling, and `tbVertical`, for vertical tiling. Some applications use two different menu commands for the two tiling modes; other applications offer only one Tile menu command but check whether the Shift key is pressed when the user selects it. This actually confuses most users, so you'll probably want to keep your application simple, with one tiling option.
- The `ArrangeIcons` procedure arranges all the iconized child windows, starting from the lower-left corner of the client area of the frame window, and moving to the upper-right corner. Open forms are not moved.

These procedures and properties are useful for handling the Window menu of an MDI application. For example, you can write the following code:

```
procedure TMainForm.Cascade1Click(Sender: TObject);
begin
  Cascade;
end;
```

As a better alternative, you can place an ActionList in the form and add to it a series of predefined MDI actions. The related classes are `TWindowArrange`, `TWindowCascade`, `TWindowClose`, `TWindowTileHorizontal`, `TWindowTileVertical`, and `TWindowMinimizeAll`. The connected menu items will perform the corresponding actions and will be disabled if no child window is available. The MdiDemo example, which we'll look at next, demonstrates the use of the MDI actions, among other things.

There are also some other interesting methods and properties related strictly to MDI in Delphi:

- `ActiveMDIChild` is a run-time and read-only property of the MDI frame form, and it holds the active child window. The user can change this value by selecting a new child window, or the program can change it using the `Next` and `Previous` procedures.

- The `Next` procedure activates the child window that follows the active one in the internal order.
- The `Previous` procedure activates the child window preceding the active one in the internal order.
- The `ClientHandle` property holds the Windows handle of the MDI client window, which covers the client area of the main form.
- The `MDIChildCount` property stores the current number of child windows.
- The `MDIChildren` property is an array of child windows. You can use this and the `MDIChildCount` property to cycle among all of the child windows, for example using a `for` loop. This can be useful for finding a particular child window or to operate on each of them.

Note that the internal order of the child windows is the reverse order of activation. This means that the last child window selected is the active window (the first in the internal list), the second-to-last child window selected is the second, and the first child window selected is the last. This order determines how the windows are arranged on the screen. The first window in the list is the one above all others, while the last window is below all others, and probably hidden away. You can imagine an axis (the *z*-axis) coming out of the screen toward you. The active window has a higher value for the *z*-coordinate and, thus, covers other windows. For this reason, the Windows ordering schema is known as the *z-order*.

The MdiDemo Example

I've built a first example to demonstrate most of the features of a simple MDI application. MdiDemo is actually a full-blown MDI text editor, because each child window hosts a Memo component and can open and save text files. The child form has a `Modified` property used to indicate whether the text of the memo has changed. It is used by the file operations and when the form is closed. The file operations are performed by a couple of extra methods, as you can see in the class declaration of the form:

```
type
 TChildForm = class(TForm)
   Memo1: TMemo;
   procedure FormClose(Sender: TObject; var Action: TCloseAction);
   procedure Memo1Change(Sender: TObject);
   procedure FormCreate(Sender: TObject);
   procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
 private
```

```
    fModified: Boolean;
    procedure SetModified(const Value: Boolean);
  public
    procedure Load (FileName: string);
    procedure Save;
    property Modified: Boolean
      read FModified write SetModified;
  end;
```

note As discussed in Chapter 3, if you want to follow the rules of OOP and provide a good encapsulation, always add properties to a form to export a field instead of making the field public. The Class Completion feature of Delphi 4 makes it so fast to add a property to a form that there are no more excuses not to do it.

The `fModified` flag is set to `True` in the handler of the memo's `OnChange` event, and it is set to `False` in the form's `OnCreate` event handler. It is also set to `False` every time a new file is loaded or saved, as you can see in the code of the two file methods:

```
procedure TChildForm.Load (FileName: string);
begin
  Memo1.Lines.LoadFromFile (FileName);
  Caption := FileName;
  fModified := False;
end;

procedure TChildForm.Save;
begin
  Memo1.Lines.SaveToFile (Caption);
  fModified := False;
end;
```

Notice that the child form uses the caption to store the filename, a shortcut I've adopted instead of adding a second property to the form.

The `fModified` flag is checked when a child form is closed, as you can see in the following listing. Keep in mind that the `OnCloseQuery` method of the child forms is also automatically activated when you close the main form:

```
procedure TChildForm.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  CanClose := not fModified or (MessageDlg ('Close without saving?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

As I've already mentioned, the main form of this example is based on an ActionList component. The actions are available through some menu items and a toolbar, as

you can see in Figure 8.17. You can see the details of the ActionList in the source code of the example.

Next, I want to focus on the code of the custom actions. Once more, this example demonstrates that using actions makes it very simple to modify the user interface of the program, without writing any extra code. In fact, there is no code directly tied to the user interface.

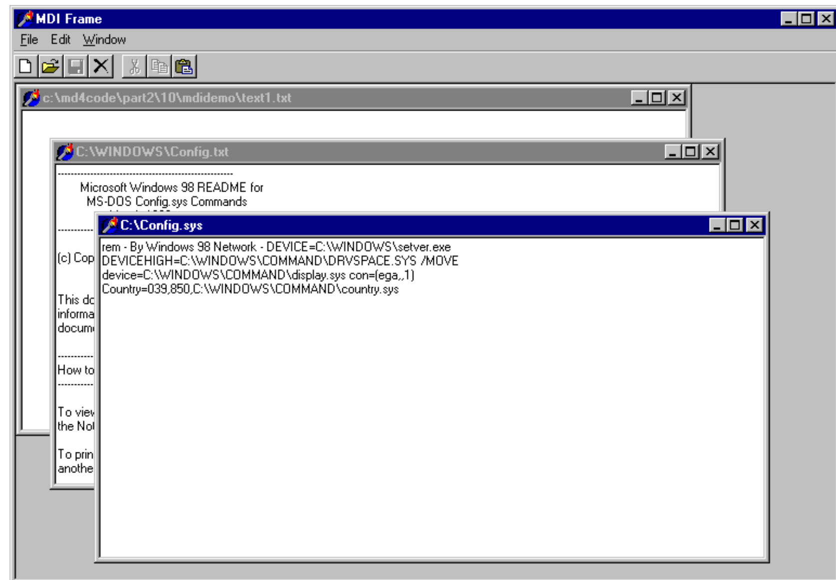


Figure 8.17: The MdiDemo program uses a series of predefined Delphi actions connected to a menu and a toolbar. Image from the original book.

One of the simplest actions is the `ActionFont` object, which has both an `OnExecute` handler, which uses a FontDialog component, and an `OnUpdate` handler, which disables the action (and hence the associated menu item and toolbar button) when there are no child forms:

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
  if FontDialog1.Execute then
    (ActiveMDIChild as TChildForm).Memo1.Font :=
      FontDialog1.Font;
end;

procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
  ActionFont.Enabled := MDIChildCount > 0;
end;
```

The action named New creates the child form and sets a default filename. The Open action calls the `ActionNewExcecute` method prior to loading the file:

```
procedure TMainForm.ActionNewExecute(Sender: TObject);
var
  ChildForm: TChildForm;
begin
  Inc (Counter);
  ChildForm := TChildForm.Create (Self);
  ChildForm.Caption :=
    LowerCase (ExtractFilePath (Application.Exename)) +
    'text' + IntToStr (Counter) + '.txt';
  ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    ActionNewExecute (Self);
    (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
  end;
end;
```

The actual file loading is performed by the `Load` method of the form. Likewise, the `Save` method of the child form is used by the Save and Save As actions. Notice the `OnUpdate` handler of the Save action, which enables the action only if the user has changed the text of the memo:

```
procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
  // suggest the current file name
  SaveDialog1.FileName := ActiveMDIChild.Caption;
  if SaveDialog1.Execute then
  begin
    // modify the file name and save
    ActiveMDIChild.Caption := SaveDialog1.FileName;
    (ActiveMDIChild as TChildForm).Save;
  end;
end;

procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
  ActionSave.Enabled := (MDIChildCount > 0) and
    (ActiveMDIChild as TChildForm).Modified;
end;

procedure TMainForm.ActionSaveExecute(Sender: TObject);
begin
  (ActiveMDIChild as TChildForm).Save;
end;
```

MDI Applications with Different Child Windows

A common approach in complex MDI applications is to include child windows of different kinds (that is, based on different child forms). We will build a new example, called MdiMulti, to highlight some problems you may encounter with this approach. For this example, we need to build two different types of child forms. The first type will host a circle drawn in the position of the last mouse click, while the second will contain a bouncing square. Another feature I'll add to the main form is a custom background obtained by painting a tiled image in it.

Child Forms and Menus

The first type of child form can display a circle in the position where the user clicked one of the mouse buttons. Figure 8.18 shows an example of the output of the MdiMulti program. The program includes a Circle menu, which allows the user to change the color of the surface of the circle as well as the color and size of its border. What is interesting here is that to program the child form, we do not need to consider the existence of other forms or of the frame window. We simply write the code of the form, and that's all. The only special care required is for the menus of the two forms.

If we prepare a main menu for the child form, it will replace the main menu of the frame window when the child form is activated. An MDI child window, in fact, cannot have a menu of its own. But the fact that a child window can't have any menus should not bother you, because this is the standard behavior of MDI applications. You can use the menu bar of the frame window to display the menus of the child window. Even better, you can merge the menu bar of the frame window and that of the child form. For example, in this program, the menu of the child form can be placed between the frame window's File and Window pull-down menus. You can accomplish this using the following `GroupIndex` values:

- File pull-down menu, main form: 1
- Window pull-down menu, main form: 3

Circle pull-down menu, child form: 2

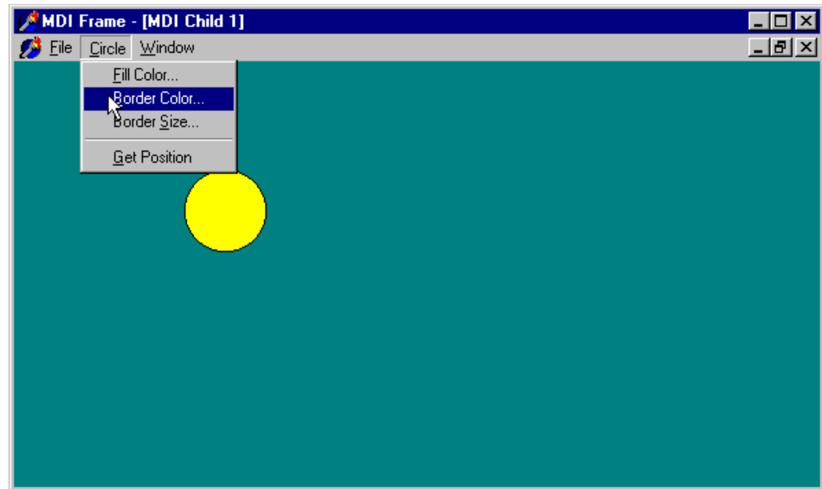


Figure 8.18: The output of the MdiMulti example, with a child window that displays circles. Image from the original book.

Using these settings for the menu group indexes, the menu bar of the frame window will have either two or three pull-down menus. At start-up, the menu bar has two menus. As soon as you create a child window, there are three menus, and when the last child window is closed (destroyed), the Circle pull-down menu disappears. You can see this in Figure 8.18, but you should also spend some time testing this behavior by running the program.

The code of the child window simply draws a shape on its sources. (For a complete discussion of this program, check out the Chapter 22, "Graphics in Delphi"). If you look at the source code, it is interesting to notice how the menu commands of the running program pertains to the two forms, and that in the source code, each form handles its own commands, regardless of the existence of other elements.

The data of the child form, particularly the coordinates of the center of the circle, must be declared using some fields of the form and not other variables declared inside the unit. In fact, we need a specific memory location to store the center of the circle for each child window.

The second type of child form shows a moving image. The square, a Shape component, moves around the client area of the form at fixed time intervals, using a Timer component, and bounces on the edges of the form, changing its direction. This turning process is determined by a fairly complex algorithm, which we don't have space to examine. The main point of the example, instead, is to show you how menu merging behaves when you have an MDI frame with child forms of different types. (You can study the downloaded source code to see how it works.)

Changing the Main Form

Now we need to integrate the two child forms into an MDI application. The main form must provide a menu command to create a child form of the selected kind and to check the group indexes of the pull-down menus. The File pull-down menu here has two separate New menu items, which are used to create a child window of either kind. The code uses a single child window counter. As an alternative, you could use two different counters for the two kinds of child windows. Again, the Window menu uses the predefined MDI actions.

As soon as a form of this kind is displayed on the screen, its menu bar is automatically merged with the main menu bar. When you select a child form of one of the two kinds, the menu bar changes accordingly. Once all the child windows are closed, the original menu bar of the main form is reset. By using the proper menu group indexes, we let Delphi accomplish everything automatically, as you can see in Figure 8.19.

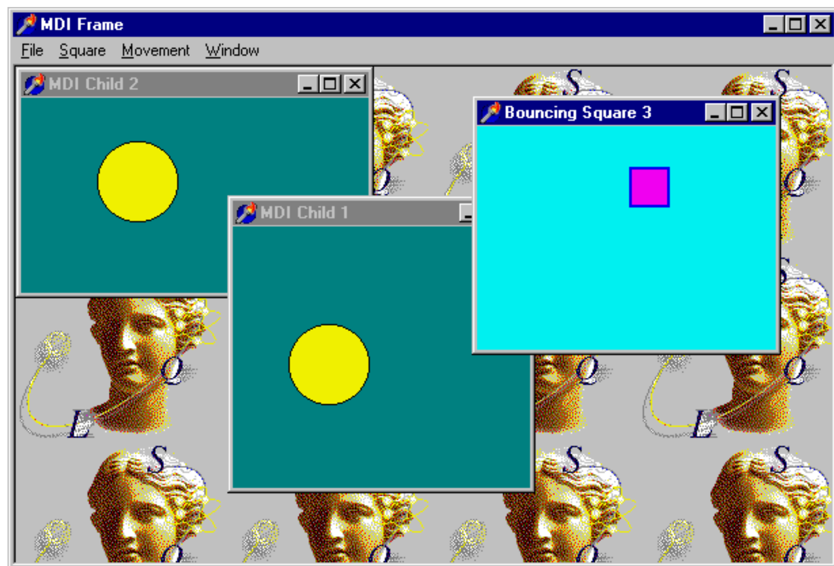


Figure 8.19: The menu bar of the MdiMulti Demo4 application changes automatically to reflect the currently selected child window, as you can see by comparing the menu bar with that of Figure 8.18. Image from the original book.

I've added a few other menu items in the main form. One menu choice is used to close every child window and another shows some statistics about them. The method related to the `Count` command scans the `MDIChildren` array property to count the number of child windows of each kind (using the RTTI operator `is`). Once

these values are computed, they are shown on the screen with the `MessageDlg` function:

```
procedure TMainForm.Count1Click(Sender: TObject);
var
  NBounce, NCircle, I: Integer;
begin
  NBounce := 0;
  NCircle := 0;
  for I := 0 to MDIChildCount - 1 do
    if MDIChildren is TBounceChildForm then
      Inc (NBounce)
    else
      Inc (NCircle);
  MessageDlg (
    Format ('There are %d child forms.'#13 +
      '%d are Circle child windows and ' +
      '%d are Bouncing child windows',
      [MDIChildCount, NCircle, NBounce]),
    mtINformation, [mbOk], 0);
end;
```

Subclassing the MdiClient Window

Finally, the program includes support for a background-tiled image. The bitmap is taken from an Image component and should be painted on the form in the `wm_EraseBkgnd` Windows message's handler. The problem is that we cannot simply connect the code to the main form, as its surface is covered by a separate window, the MdiClient window described earlier in this chapter.

We have no corresponding Delphi form for this window, so how can we handle its messages? We have to resort to a low-level Windows programming technique known as *subclassing*. (In spite of the name, this has little to do with OOP inheritance). The basic idea is that we can replace the window procedure, which receives all the messages of the window, with a new one we provide. This can be done by calling the `SetWindowLong` API function and providing the memory address of the procedure, the function pointer.

note A window procedure is a function receiving all the messages for a window. Every window must have a window procedure and can have only one. Even Delphi forms have a window procedure; although this is hidden in the system, it calls the `WndProc` virtual function, which you can use. But the VCL has a predefined handling of the messages, which are then forwarded to the message-handling methods of a form after some preprocessing. With all this support, you need to handle window procedures explicitly only when working with non-Delphi windows, as in this case. For a thorough description of this topic you can refer to *Delphi Developer's Handbook* (Sybex, 1998), among other books.[209]

Unless we have some reason to change the default behavior of this system window, we can simply store the original procedure and call it to obtain a default processing. The two function pointers referring to the two procedures (the old and the new one) are stored in two local fields of the form:

```
private
  OldWinProc, NewWinProc: Pointer;
  procedure NewWinProcedure (var Msg: TMessage);
```

The form also has a method we'll use as a new window procedure, with the actual code used to paint on the background of the window. Because this is a method and not a plain window procedure, the program has to call the `MakeObjectInstance` method to add a prefix to the method and let the system use it as if it were a function. All this description is summarized by only two complex statements:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  NewWinProc := MakeObjectInstance (NewWinProcedure);
  OldWinProc := Pointer (SetWindowLong (
    ClientHandle, gwl_WndProc, Cardinal (NewWinProc)));
  OutCanvas := TCanvas.Create;
end;
```

The window procedure we install calls the default one. Then, if the message is `wm_EraseBkgnd` and the image is not empty, we draw it on the screen many times using the `Draw` method of a temporary canvas. This canvas object is created when the program starts (see the code above) and connected to the handle passed as `wParam` parameter by the message. With this approach, we don't have to create a new `TCanvas` object for every background painting operation requested, thus saving a little time in the frequent operation. Here is the code, which produces the output already seen in Figure 8.19:

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
```

209 That book is now hard to find. Restoring it would be another interesting project.

```
  BmpWidth, BmpHeight: Integer;
  I, J: Integer;
begin
  // default processing first
  Msg.Result := CallWindowProc (OldWinProc,
    ClientHandle, Msg.Msg, Msg.wParam, Msg.lParam);

  // handle background repaint
  if Msg.Msg = wm_EraseBkgnd then
  begin
    BmpWidth := MainForm.Image1.Width;
    BmpHeight := MainForm.Image1.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
      OutCanvas.Handle := Msg.wParam;
      for I := 0 to MainForm.ClientWidth div BmpWidth do
        for J := 0 to MainForm.ClientHeight div BmpHeight do
          OutCanvas.Draw (I * BmpWidth,
            J * BmpHeight, MainForm.Image1.Picture.Graphic);
    end;
  end;
end;
```

What’s Next?

We have explored different ways to build applications that have several forms or forms with multiple pages. We have seen how you can create a secondary modeless form or a modal dialog box. Besides the basic examples, we have delved into some advanced topics, such as dynamically building a number of forms; creating extensible dialog boxes, using the common dialogs and the Delphi message boxes; building special About boxes, with hidden credits or used as a splash screen; as well as the MDI technique.

There are many things we could do to further explore how to build applications with multiple forms and extend their user interface. I’ve given equal coverage to various techniques, although I have my preferences: fewer secondary forms, more dialog boxes, MDI only for specific programs, notebooks, and docking whenever possible.

Now we can move forward to a very hot Delphi programming topic: building database applications. This will take the next four chapters, which cover most of the fundamental topics of Delphi database programming. It is possible to write a specific book about this, so the description won’t be exhaustive, but you should be able to get a comprehensive overview of this key element of Delphi development.

After these three database chapters, we'll be able to start looking into Delphi behind the scenes and focus on topics such as the construction of Delphi components and ActiveX controls.

Chapter 9: Writing Database Applications

Delphi's support for database applications is one of the key features of the programming environment. Many programmers spend most of their time writing data-access code, which needs to be the most robust portion of a database application. This chapter provides an overview of Delphi's extensive support for database programming. You can create very complex database applications, starting from a blank form or one generated by Delphi's Database Form Wizard[210].

210 The Database Form Wizard is no longer available, but also a lot of the specific techniques described in this chapter are not applicable any more. For one, the BDE data access library no longer ships with Delphi, replaced by newer alternatives like FireDAC. Also using paradox tables, while technically possible, has long been deprecated and it no longer recommended (or even suggested). In other words, the content of this chapter and the demos have severe limitations, but some of the key concepts of the core DB.pas unit are still valid today.

What you won't find here is a discussion of the theory of database design. I'm assuming that you already know the fundamentals of database design and have already designed the structure of a database. I won't delve into database-specific problems; my goal is to help you understand how Delphi supports this kind of programming.

We'll begin with an explanation of how data access works in Delphi, and then we'll review the database components that are available in Delphi. I won't discuss the simplest examples and techniques step by step, such as the use of the Database Form Wizard, but instead I will focus on the foundations. Some of the topics covered in this chapter include an in-depth example of the `TField` components, creating new tables with Delphi code, and using graphics. The following chapters will provide information on many other more advanced database programming topics.

Accessing Data with and without the BDE

On a computer, permanent data—including database data—is always stored in files. The two most common approaches are to store a whole database in what appears to the file system as a single file or to store each table, index, and any other elements of the database in separate files, usually on the same directory. Delphi supports both approaches, depending on the database format you are using:

- Paradox and dBASE tables[211] define databases as directories and each table as a separate file (or actually multiple files if you include indexes and other support files).
- Access, InterBase, and most SQL servers use a single file containing the entire database, with all the tables and indexes.[212].

211 This is a very old approach, not recommended today. For simple local data storage, a good replacement is to use memory tables (FDMemTable) and store their content to file. The different, though, is that persistent tables can be loaded by one application at a time, while Paradox and dBase accounted for access by multiple apps at the same time, with techniques explained later in this chapter.

212 This is still generally true today.

note The Borland Database Engine (BDE)[213] uses an alias to refer to a database file or directory. You can define new aliases for databases by using the Database Explorer or the Database Engine Configuration utility. It is also possible to define them by writing code in Delphi that calls the `AddStandardAlias` and `AddAlias` methods of the `Session` global object, followed by a call to `SaveConfigFile` to make the alias persistent. The alternative is the low-level `DbiAddAlias` BDE function.

Delphi database applications do not have direct access to the data sources they reference and cannot manipulate database files directly. Instead, they use an available database engine, such as the Borland Database Engine (BDE) or Microsoft's ActiveX Data Objects (ADO)[214].

The BDE has direct access to a number of data sources, including dBASE, Paradox, ASCII, FoxPro, and even Access tables. The BDE can also interface with Borland's SQL Links, a series of drivers allowing access to a number of local and remote SQL servers (available only in Delphi Enterprise). Database servers include Oracle, Sybase, Informix, InterBase, and DB2[215]. If you need access to a different database or data format, the BDE can interface with ODBC drivers, although in this case you might want to use ADO instead. Notice, anyway that the BDE provides advanced features (such as sophisticated caching and heterogeneous joins) that ADO doesn't offer.

ADO is Microsoft's high-level interface[216]. ADO is implemented on Microsoft's data access OLE DB technology, which provides access to relational and non-relational databases as well as e-mail and file systems and custom business objects. Applications built with Delphi 5's ADO components don't require the BDE libraries. Of course, users need to have the ADO/OLE DB run time, which is distributed by Microsoft and is part of the Windows 2000 operating system[217]. ADO will also need to be configured on the user's machine, even if it is already installed. Chapter 12 will more fully cover ADO and related technologies.

---

213 As mentioned, the BDE no longer ships with Delphi, although it has remained available as a separate download for some time. Needless to day the associated Database Explorer and Database Engine Configuration tools are also no longer part of the Delphi installation.

214 Or FireDAC, or IBX, or the now deprecated dbExpress (DBX), or other third party database access libraries.

215 FireDAC offers many more alternatives in terms of direct database support and it also includes an ODBC gateeway.

216 Support for ADO ~(via the dbGo library) is still available in today's Delphi, even if it's not considered a top option as the underlying Microsoft library, while still available, has been neglected in favor of ADO.NET.

217 The library is still part of Windows 11 and also of recent Windows Server editions.

Delphi Enterprise includes native components that access Borland's own InterBase server[218] (available on the Delphi installation CD; see Chapter 11 for more details) and the ClientDataSet component (see Chapter 21), which can be used for local or remote data access. These technologies provide alternatives to the traditional use of the BDE to access a database from Delphi applications. Figure 9.1 shows the alternative data-access techniques available in Delphi 5[219], indicating that all the data-access components inherit from a common base class, `TDataSet`.

If you choose the traditional BDE approach (as most of the examples in this chapter do), you will need to install the BDE along with your applications on your clients' computers. This is not difficult, because Delphi includes the "lite" version of a widely used installation program (InstallShield) that can be used to prepare installation disks for the BDE, along with your own application. The BDE files are required —your Delphi database applications won't work without them—but you can distribute them freely.

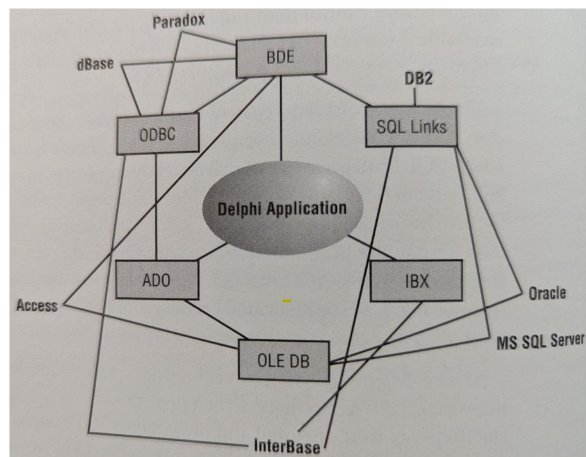


Figure 9.1: The alternative data access technologies available in Delphi 5. Image based on a picture of the original printed book.

218 InterBase Developer edition is an optional installation of todays's Delphi, while the embedded version of the database (IBLite/IBToGo) is installed automatically and is available for desktop and mobile applications. See the InterBase section of Embarcadero web site for more details.

219 This image depicts what was available at the time, very different of what you can do today with FireDAC and other alternatives.

Delphi Database Components

Delphi includes a number of components related to databases. The Data Access page[220] of the Component Palette contains components used to interact with databases in BDE-oriented applications. Most of them are non-visual components, since they encapsulate database connections, tables, queries, and similar elements. Fortunately, Delphi also provides a number of predefined components you can use to view and edit database data. In the Data Controls page, there are visual components used to view and edit the data in a form. These controls are called *data-aware* controls[221].

To access a database in Delphi, you generally need a data source, identified by the DataSource component[222]. The DataSource component, however, does not indicate the data directly; it refers to a DataSet component. This can be a table, the result of a query, the result of a stored procedure, the data fetched from a remote server (using the ClientDataSet component), ADO, InterBase, or some other custom dataset.

As soon as you have placed a dataset component on the form, you can use the DataSet property of the DataSource component to refer to it. For this property, the Object Inspector lists the available datasets of the current form or of other forms and data modules connected with the current one (using the File ➢ Use Unit command).

Tables and Queries

The simplest way to specify data access in Delphi is to use the Table component[223]. A Table object simply refers to a database table. When you use a Table component, you need to indicate the name of the database you want to use in its `DatabaseName`

---

220 The Data Access page still exists, but it hosts general purpose components only, not the BDE ones, which are not in the product any more.

221 The VCL library still offers data-aware controls, but it also include Visual Live Bindings, the primary technique used in FireMonkey to associated UI controls and data.

222 The role of the DataSource component hasn't changed. It still refers to a TDataSet descendant, exactly like in the early days of Delphi. The difference is that newer TDataSet descendants are not available, like FireDAC datasets.

223 With this component missing, a good starting point is now the FDMemTable component, which can be mapped to local data files. The alternative is to use the FDTable component, but it requires the connection to a RDBMS to fetch any data.

property. You can enter an alias or the path of the directory with the table files. The Object Inspector lists the available names, which depend on the aliases installed in the BDE.

You also need to indicate a proper value in the `TableName` property[224]. The Object Inspector lists the available tables of the current database (or directory), so you should generally select the `DatabaseName` property.

A second data set available in Delphi is the Query component[225]. A query is usually more complex than a table, because it requires a SQL language command. However, you can customize a query using SQL more easily than you can customize a table (as long as you know at least the basic elements of SQL, of course). The Query component has a `DatabaseName` property like the Table component, but it does not have a `TableName` property. The table is indicated in the SQL statement, stored in the `SQL` property.

note SQL is a standard language for writing database queries and generally interacting with a database. If you are not fluent in SQL, you can find a description of its basic commands in Chapter 11. Delphi Enterprise includes a tool to create SQL queries, called SQL Builder[226], and is discussed in Chapter 11, as well.

For example, you can write a simple SQL statement like this

```
select * from Country
```

where `Country` is the name of a table, and the star symbol (`*`) indicates that you want to use all of the fields in the table. If you are fluent in SQL, you might use the Query component more often, but the efficiency of a table or a query varies depending on the database you are using. In general, we can say that the Table component tends to be faster on local tables, while the Query component tends to be faster on SQL servers, although this is just a very general rule, and in many cases you might have the opposite effect. I'll cover some efficiency issues in Chapters 10 and 11.

---

224 By contrast, FDMemTable only needs to refer to a local file.

225 The matching FireDAC component is called FDQuery.

226 This tool is also long gone, but FireDAC designers offer some help in creating queries and specific tools replacing the old query builder..

note The Country table mentioned above refers to the file COUNTRY.DB, which is part of the Delphi's demo database, installed by default in the C:\Program Files\Common Files\Borland Shared\Data directory[227]. This directory is referenced by the DBDEMOS alias, set up by Delphi during the installation. Many of my examples in the following chapters will use tables from this Delphi database. In others, I'll show you how to build new tables, but I'll generally use that demo database, as well.

The third component for data sets is StoredProc, which refers to stored procedures of a SQL server database. You can run these procedures and get the results in the form of a database table. Stored procedures can only be used with SQL servers.

The Status of a Data Set[228]

When you operate on a data set in Delphi, you can work in different states, indicated by a specific `State` property, which can assume several different values:[229]

- `dsBrowse` indicates that the data set is in normal browse mode, used to look at the data and scan the records.
- `dsEdit` indicates that the data set is in edit mode. A data set enters this state when the program calls the `Edit` method or the DataSource has the `AutoEdit` property set to `True`, and the user starts editing a data-aware control, such as a DBGrid or DBEdit. When the changed record is posted, the data set exits the `dsEdit` state.
- `dsInsert` indicates that a new record is being added to the data set. Again, this might happen when calling the Insert method, moving to the last line of a DBGrid, or using the corresponding command of the DBNavigator component.
- `dsInactive` is the state of a closed data set.

---

227 A similar directory under the demos folder (installed by default under the Windows users public directory, *C:\Users\Public\Embarcadero\xx.0*) includes same of the same old Paradox tables converted to the format required by FireDAC's FDMemTable.

228 The concept of Dataset status is still applicable and very important today, regardless of the data access components used. The core idea remains the same and it's very important to understand for Delphi database access.

229 Additional values compared to this list are dsBlockRead (used for reading data without chancing the current record), dsInternalCalc (similar to dsCalcFields, but for a modified version of the calculated fields logic), and dsOpening (used to indicate the dataset is being opened, but it' isn't ready yet)

- `dsSetKey` indicates that we are preparing a search on the data set. This is the state between a call to the `SetKey` method and a call to the `GotoKey` or `GotoNearest` methods (see the Search example later in this chapter).
- `dsCalcFields` is the state of a data set while a field calculation is taking place; that is, during a call to an `OnCalcFields` event handler. Again, I'll show this in an example.
- `dsNewValue`, `dsOldValue`, and `dsCurValue` are the states of a data set when an update of the cache is in progress.
- `dsFilter` is the state of a data set while setting a filter; that is, during a call to an `OnFilterRecord` event handler.

In simple examples, the transitions between these states are handled automatically, but it is important to understand them because there are many events referring to the state transitions.

note We will use a simple state-transition event, the `OnStateChange` event of the DataSource component, in the GridDemo example, the first example of this chapter.

Other Database Related Components

Along with the Table, Query, StoredProc, and DataSource, there are some other components in the Data Access page of the Component palette, the BDE page. I'll cover these components in the next two chapters, but here is a short summary:

- The Database component is used for transaction control, security, and connection control[230]. It is generally used only to connect to remote databases in client/server applications or to avoid the overhead of connecting to the same database in several forms. The Database component is also used to set a local alias used only inside a program. Once this local alias is set to a given path, the Table and Query components of the application can refer to the local database alias. This is much better than replicating the hard-coded path in each DataSet component of the program.

230 In FireDAC, this is replaced by a combination of the FDConnection and FDTransaction components.

- The Session component[231] provides global control over database connections for an application, including a list of existing databases and aliases and an event to customize database login.
- The BatchMove[232] component is used to perform batch operations, such as copying, appending, updating, or deleting values, on one or more databases.
- The UpdateSQL[233] component allows you to write SQL statements to perform various update operations on the data set, when using a read-only query (that is, when working with a complex query). This component is used as the value of the UpdateObject property of tables or queries.

Delphi Data-Aware Controls

We have seen how it is possible to connect a data source to a database, using either a table or query, but we still do not know how to view the data. For this purpose, Delphi provides many components that resemble the usual Windows controls but are data-aware. For example, the DBEdit component is similar to the Edit component, and the DBCheckBox component corresponds to the CheckBox component. You can find all of these components in the Data Controls page[234] of the Delphi Component Palette:

- DBGrid is a grid capable of displaying a whole table at once. It allows scrolling and navigation, and you can edit the grid's contents. It is an extension of the other Delphi grid controls.
- DBNavigator is a collection of buttons used to navigate and perform actions on the database. The buttons perform basic actions, so you can easily replace them with your own toolbar.
- DBText displays the contents of a field that cannot be modified. It is a data-aware Label graphical control.

---

231 The concept of Session was specific to the BDE. It has no match in other data access libraries.

232 FireDAC offers an extremely sophisticated "*batch move*" subsystem, based on FDBatchMove but also many other specific components for reading and writing different data formats, from database tables to XML, from JSON to CVS, from text files to other formats.

233 The same concept exists in FireDAC and other libraries, although FDUpdateSQL is only used in very complex scenarios, as the core components like FDQuery offer automatic updates in many cases.

234 These data-aware controls still exists aand are still frequently used. The only exception is the DBCltrGrid, which is no longer very commonly used even if it's still available.

- DBEdit lets the user edit a field (change the current value) using an Edit control.
- DBMemo lets the user see and modify a large text field, eventually stored in a memo or BLOB (Binary Large OBject) field. It resembles the Memo component.
- DBImage is an extension of an Image component that shows a picture stored in a BLOB field.
- DBListBox and DBComboBox let the user select a single value from a specified set. If this set is extracted from another database table or is the result of another query, you should use the DBLookupListBox or DbLookupComboBox components instead.
- DBCheckBox can be used to show and toggle an option, corresponding to a Boolean field, and extends the CheckBox component.
- DBRadioGroup provides a series of choices, with a number of exclusive selection radio buttons, such as the RadioGroup control.
- DBRichEdit is a component that lets the user edit a formatted text file; it is based on a Windows 95 RichEdit control.
- DBCtrlGrid is a multi-record grid, which can host a number of other data-aware controls. These controls are duplicated for each record of the data set.
- DBChart[235] is a data-aware business graphic component or the data-aware version of the Chart component.

All of these components are connected to a data source using the corresponding property, `DataSource`. Some of them relate to the entire data set, such as the DBGrid and DBNavigator components, while the others refer to a specific field of the data source, as indicated by the `DataField` property. Once you select the `DataSource` property, the `DataField` property will have a list of values available in the drop-down combo box of the Object Inspector.

235 This is tied to the Steema TeeChart add-on available in Delphi.

Customizing a Database Grid

Our first database example, called GridDemo, uses the COUNTRY.DB[236] table from the DBDEMOS database, which lists New World countries, along with each one's capital and population. Simply place a Table, a DataSource, and a DBGrid component on a form and connect them. If you set the `Active` property of the table to `True`, the data will appear in the form at design time. (This technique is usually called *live-data* design[237].) When a grid displays live data, you can even use its scroll bars to navigate through the records.

At this point, you can already run the program and even edit the data of the database table, making permanent changes. This is possible because the DBGrid component's `Options` property includes the flag `dgEditing` and the `ReadOnly` property is set to `False`. This program also allows you to insert a new record in a given position by pressing the Insert key, to append a new record at the end by going to the last record and pressing ↓, and to delete the current record by pressing Ctrl+Del.

note Try using this program for a while, testing how it works when you toggle the various flags of the `Options` property of the grid on and off. These flags determine the behavior of the grid, which can vary a lot. You can also see the description of the various options in Delphi's help file.

Besides the `Options` property, you can customize the DBGrid component with the easy-to-use yet very powerful `Columns` property. This property is a collection, so you can choose one of the items in the list and then set its property in the Object Inspector, as you can see in Figure 9.2.

You can easily choose the fields of the table you want to see in the grid as columns and then set a number of column properties (color, font, width, alignment, and so on) for each field and title properties, such as the caption, font, and colors. This allows you to customize a grid easily, in a number of ways. Some of the more advanced properties, such as `ButtonStyle` and `DropDownRows`, can be used to provide custom editors for the cells of a grid or a drop-down list.

---

236 This table remains available in multiple format, including FireDAC's FDMemTable native format (with .fds extension).

237 Nowadays, the feature is generally indicated as "*live-data at design time*" and, after all of these years, Delphi remains one of the few dev tools offering this very handy feature.

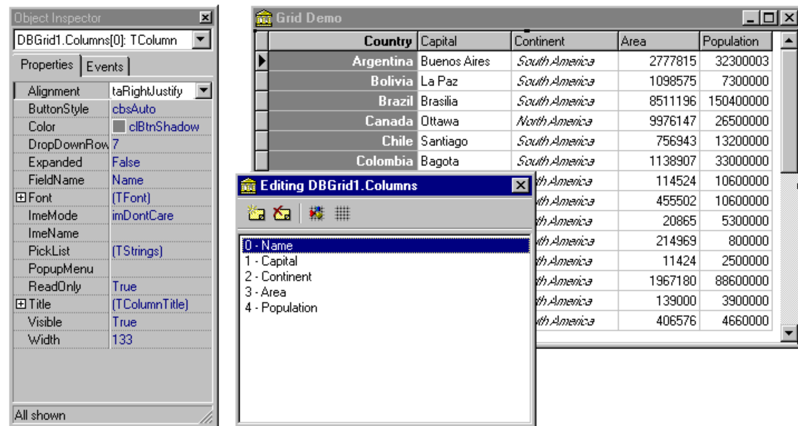


Figure 9.2: You can edit the properties of the Columns of a DBGrid by selecting one of the columns in the collection editor and using the Object Inspector. Image from the original book.

In the GridDemo example, I've changed the caption of the first column and the font of the first and third. I've also chosen a dark gray background and a white font color for the first column. I've also entered the names of a few continents in the `PickList` string list of the Continent field. You can see the result in Figure 9.3.

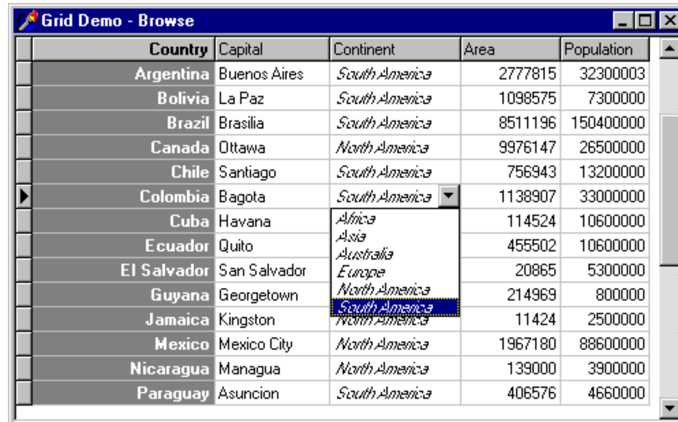


Figure 9.3: The DBGrid of the GridDemo example has a few customized Columns, including a PickList for the continents. Image from the original book.

note Notice that once you have defined the `Columns` property of the DBGrid, you can size the columns at design time simply by dragging the lines separating them. The same capability is optionally available at run time, and it can be set along with many others using the `Options` property of the grid.

To summarize the features of this simple example, here is an extract of the form description file:

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'Grid Demo'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
    Columns = <
      item
        Alignment = taRightJustify
        Color = clBtnShadow
        FieldName = 'Name'
        Font.Style = [fsBold]
        ReadOnly = True
        Title.Alignment = taRightJustify
        Title.Caption = 'Country'
        Title.Font.Style = [fsBold]
      end
      item
        FieldName = 'Capital'
      end
      item
        Expanded = False
        FieldName = 'Continent'
        Font.Style = [fsItalic]
        PickList.Strings = (
          'Africa'
          'Asia'
          'Australia'
          'Europe'
          'North America'
          'South America')
      end
      item
        FieldName = 'Area'
      end
      item
        FieldName = 'Population'
      end>
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
    OnStateChange = DataSource1StateChange
  end
end
```

The Table State

There are many more things you can do to customize grids, and we'll explore some of them in the next chapter, where we will also discuss how to add graphics to a grid. For the moment, I want to add an extra feature (and some code) to the example. If you look at the caption of the form in Figure 9.3, you'll notice something new: the title of the form indicates the status of the Table component. How do we get this information? Simply by handling the `OnStateChange` event of the DataSource component. In this event handler, the DemoGrid example merely outputs the current status, determined by using a simple `case` statement:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
  Title: string;
begin
  case Table1.State of
    dsBrowse: Title := 'Browse';
    dsEdit: Title := 'Edit';
    dsInsert: Title := 'Insert';
  else
    Title := 'Other state';
  end;
  Caption := 'Grid Demo - ' + Title;
end;
```

The code considers only the three states the Table component[238] of this program can have as the user interacts with the corresponding DBGrid.

Field-Oriented Data-Aware Controls

The GridDemo example works well, but we want to try using other controls, such as edit boxes, and we want to see specific information rather than all the data in our database. Before we really look at the core of the VCL database structure, by examining the `TField` components, I want to cover the usage of some of the data-aware controls you can use to see and edit the value of a database field. The starting point is the use of edit boxes.

238 Given the State property is part of the TDataSet base class, the same logic and code would work for any TDataSet descendant, including the FDMemTable component.

Using DBEdit Controls

The next example, called EditDemo, uses some DBEdit components and some labels, along with the table and the data source. We also need to add a brand-new component, the DBNavigator. Figure 9.4 shows the form of the EditDemo example at design time (with live data).

Again, we need to connect the three data-aware controls to the data source by setting their `DataSource` property, and we must also indicate a specific field for each of the three edit boxes in their `DataField` property (Name, Capital, and Continent are the fields for this example). If you have already connected the data source to the table and the edit boxes to the data source, you can simply select a field in the list displayed by the Object Inspector for the `DataField` property. When this connection is made, if the `Active` property of the Table is set to `True` the values of the first record's fields appear automatically in the edit boxes (see Figure 9.4).

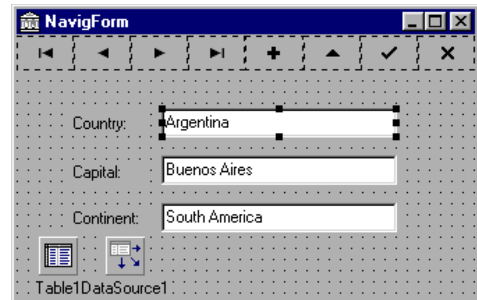


Figure 9.4: The three DBEdit and the DBNavigator components of the EditDemo example, with live data. Image from the original book.

Another step we can take is to disable some of the buttons of the DBNavigator control, by removing some of the elements of the `VisibleButtons` set.

note If you turn on its `ShowHint` property, the navigator will show a different fly-by hint for every button. You can provide a customized description of each of them, using the `Hints` string list. The strings you insert are used for the buttons in order: the first string is used for the first button, the second for the second, and so on. If some buttons are not visible, you can provide an empty string as a placeholder.

In the EditDemo program, I've used only some of the buttons, disabling the delete and refresh operations. I've also aligned the navigator to the top of the form and set its `Flat` property to `True` to activate the flat button style. You can run it to test whether it works properly, and look at the caption again: I've copied to this program

the `OnStateChange` event handler of the GridDemo program's DataSource component.

Notice that when the program is running at the beginning or when you jump to the first or last record of the table, two of the navigator's buttons will be disabled automatically. However, if you move step-by-step to the first or last record, the buttons are disabled only when you try to move beyond those records. The navigator (or the dataset, to be more precise) only realizes at this point that there are no more records in that direction. Other buttons are automatically enabled and disabled when you enter or exit the edit state.

Creating a Database Table

Before we can move to some other data-aware controls, we need to do an extra step. In the first two examples of this chapter, I've used existing tables of the DBDEMOS database, but for the following ones I need to create a table with specific types of fields. For this reason, I'll introduce here a topic we'll return to later on: the creation of new database tables.

Starting with version 4, Delphi allows you to set the definition of the fields of a table —its internal structure—at design time, using the collection editor of the `FieldDefs` property. You can see the settings I've used for the DbAware example in Figure 9.5.

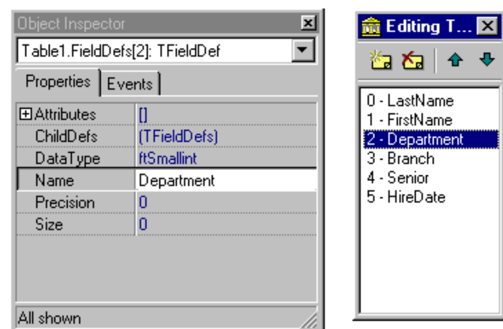


Figure 9.5: The editor of the field definitions collection. Images from the original book.

Having defined the fields, you can now right-click the table component and select the Create Table command[239]. This creates the new table at design time. In this specific example, there is no need to do this, since the program creates the table when it starts, unless the table already exists:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not Table1.Exists then
    Table1.CreateTable;
  Table1.Open;
end;
```

To make this code work, the Table component must save the definition of the fields in the DFM file along with the other properties. This is done only if you set the `StoreDefs` property of the table to `True`. In Table 9.1, you can see the table field definitions, and the following listing shows the initial portion of the corresponding definitions in the DFM file.

Table 9.1: The Fields of the Workers Database Table

Name	DataType	Size
LastName	`ftString`	20
FirstName	`ftString`	20
Department	`ftSmallint`	
Branch	`ftString`	20
Senior	`ftBoolean`	
HireDate	`ftDate`	

```
object Table1: TTable
  FieldDefs = <
    item
      Name = 'LastName'
      DataType = ftString
```

239 A very similar mechanism is available for the FDMemTable component, which offers design time commands for creating a table or a `CreateDataset` method. The same approach of using `FieldDefs` described in the following paragraphs applies to FDMemTable and other datasets.

```
    Size = 20
  end
  item
    Name = 'Department'
    DataType = ftSmallint
  end
```

This new database table, called Workers, is intended to store some data about the employees of a company. (Note that calling it "Employee" might have caused a name conflict or confusion with one of the predefined tables.)

---

note The effect of the `StoreDefs` property is more complex than it seems at first. If you right-click the form, you'll notice that its local menu offers an Update Table Definition option, along with the expected Delete Table and Rename Table. That is, you can store the field definitions locally, but if the structure of the physical table changes, you should then update this definition, as well. In previous versions of Delphi, the field definitions were invariably loaded from the database table at run time; now you can preload them, speeding up the table opening. However, if the local and the actual table definitions do not match, you can get in trouble.

As we've seen, the DbAware example creates the table at start-up, unless it was already created. The program then opens up the table. To avoid having you type in data to start using the program, I've added to the program a simple `AddRandomData` method:

```
const
  FirstNames : array [1..10] of string =
    ('John', 'Paul', 'Mark', 'Joseph', 'Bill',
    'Peter', 'Tim', 'Ralph', 'Bob', 'Gary');
  LastNames : array [1..10] of string =
    ('Ford', 'Osborse', 'White', 'MacDonald', 'Lee',
    'Young', 'Parker', 'Reed', 'Gates', 'Green');
  NoDept = 3;
  NoBranch = 30;
  NewRecords = 10;

procedure TDbaForm.AddRandomData;
var
  I: Integer;
begin
  Randomize;
  for I := 1 to NewRecords do
    Table1.InsertRecord ([
      LastNames [Random (High (LastNames)) + 1],
      FirstNames [Random (High (FirstNames)) + 1],
      Random (NoDept) + 1,
      DbComboBox1.Items [Random (NoBranch)],
      Boolean (Random (2)),
      Date - Random (1000)]);
  ShowMessage (IntToStr (NewRecords) + ' added');
```

```
end;
```

`AddRandomData` calls the `InsertRecord` method of the table, which adds new data in a direct way—without setting the table in insert mode, setting the value of the fields, and then posting the data. In other examples, we'll see alternative approaches for adding data to a database table. Notice also that for the "branches" field I've used the list of values available in the associated data-aware combo box.

Listing Alternative Values

Now that I've created the table I can use it for creating a simple demo application of some of the other data-aware controls available in Delphi. For example, we can connect the Boolean field, Senior, with a DBCheckBox control. This allows a user to change the status of the field by clicking the control and setting or removing the check mark.

While this is quite trivial, using the components that list alternative values requires a little extra effort. There are basically three components with this capability: the DBListBox, the DBComboBox, and the DBRadioGroup. In general, all the three components provide a selection, which saves the user some typing and reduces the chance of input errors. If the three components seem similar, providing a list of strings in the `Items` property, they do have some differences:

- The DBListBox component allows selection of predefined items ("closed selection"), but not text input, and can be used to list many elements. Generally it's best to show only about six or seven items, to avoid using up too much space on the screen.
- The DBComboBox component can be used both for closed selection and for user input. It also uses a smaller area of the form because the drop-down list is usually displayed only on request.
- The DBRadioGroup component allows only a closed selection, should be used only for a limited number of alternatives, and allows a mapping of the display `values` to different internal values, through the Values string list.

In the DbAware example I've used the combo box for the selection of a country and a radio group for the selection for the department. This is actually saved in the database with a code, so I've mapped it as follows:

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
```

```
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
  Values.Strings = (
    '1'
    '2'
    '3'
    '4')
end
```

You can see an example of this program in Figure 9.6. Notice that the program is based on a page control: moving to the second page, you can see the database data inside a DBGrid. This should help you understand the mapping done by the RadioGroup control. The other element is that the main page doesn't allow you to edit the hire date, which is displayed in a read-only DBText control. We'll see how to handle dates in later examples.

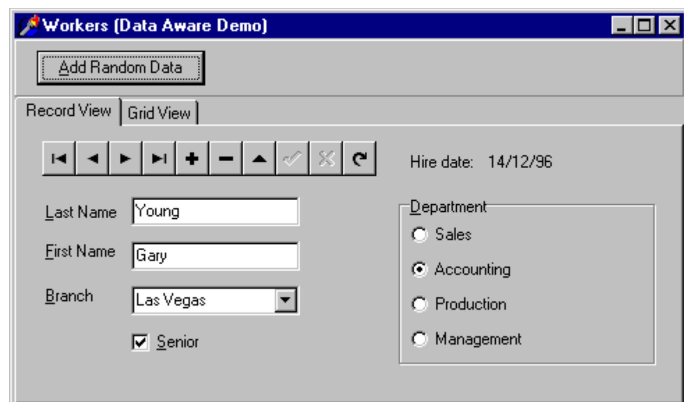


Figure 9.6: The output of the DbAware demo, which uses check box, combo box, and radio group data-aware controls. Image from the original book.

Accessing the Data Fields

Before we try to build more attractive and complex application examples, there are a few more technical elements we should explore. Up to now, we have included all of the fields in the source database tables. Suppose that we want to remove a field or add a new one, such as calculated fields? In trying to solve these problems, we face a

more general question: How do we access the values—the fields—of the current record from a program? How can we change them without direct editing by the user?

The answer to all of these questions lies in the concept of *field*. Field components (instances of class `TField` or of one of its subclasses) are non-visual components that are fundamental for every Delphi database application. Data-aware controls are directly connected to these Field objects, which correspond to database fields.

In the examples we have built up to now, Delphi automatically created the `TField` classes at run time[240]. This happens each time the program opens a data set component. These fields are stored in the `Fields` array property of tables and queries, which is an array of fields. We can access these values in our program by number (accessing the array directly) or by name (using the `FieldByName` method or the array notation):

```
Table1.Fields[0].AsString
Table1.FieldByName('LastName').AsString
Table1 ['LastName'].AsString
```

As an alternative, the field components can be created at design time, using the Fields editor. In this case, you can also set a number of properties for these fields at design time. These properties affect the behavior of the data-aware controls using them, both for visualization and for editing. When you define new fields at design time, they are listed in the Object Inspector, just like any other component.

note Although the Fields editor is similar to the editors of the collections used by Delphi, fields are not part of a collection. They are components created at design time, listed in its published section of the form class, and available in the drop-down combo box at the top of the Object Inspector.

To open the Fields editor for a table, select a Table object, activate its local menu with a right-click, and choose the Fields Editor command. Double-clicking the table component produces the same effect. An empty Fields editor appears. Now you have to activate the local menu of this editor, to access its capabilities. The simplest operation you can do is to select the Add command, which allows you to add any other fields in the database table to the list of fields. Figure 9.7 shows the Add Fields dialog box, which lists all the fields that are available in a table. These are the database table fields that are not already present in the list of fields in the editor.

The Define command of the Fields editor, instead, lets you define a new calculated field, a lookup field, or a field with a modified type. In this dialog box, you can enter

240 The concept of `TField` and the role of these objects in Delphi database application hasn't changed at all, even if some new features have been added over the years.

a descriptive field name, which might include blank spaces. Delphi generates an internal name—the name of the field component—that you can further customize. Next, select a data type for the field. If this is a calculated field or a lookup field, and not just a copy of a field redefined to use a new data type, simply check the proper radio button. We'll see how to define a calculated field in the section "Adding a Calculated Field" and a lookup field in the next chapter.

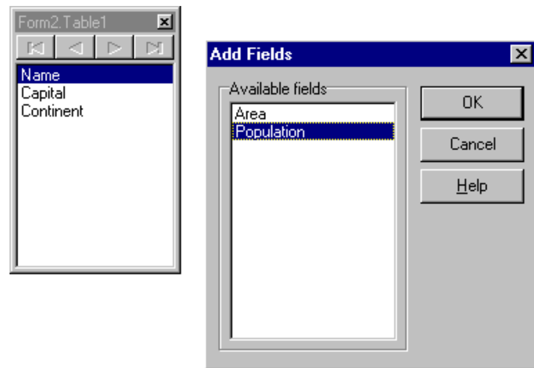


Figure 9.7: The Fields editor with the Add Fields dialog box. Images from the original book.

note A `TField` component has both a `Name` property and a `FieldName` property. The `Name` property is the usual component name. The `FieldName` property is either the name of the column in the database table or the name you define for the calculated field. It can be more descriptive than the `Name`, and it allows blank spaces. The `FieldName` property of the `TField` component is copied to the `DisplayLabel` property by default, but this field name can be changed to any suitable text. It is used, among other things, to search a field in the `FieldByName` method of the `TDataSet` class and when using the array notation.

All of the fields that you add or define are included in the Fields editor and can be used by data-aware controls or displayed in a database grid. If a field of the original database table is not in this list, it won't be accessible. When you use the Fields editor, Delphi adds the declaration of the available fields to the class of the form, as new components (much as the Menu Designer adds `TMenuItem` components to the form). The components of the `TField` class, or more specifically its subclasses, are fields of the form, and you can refer to these components directly in the code of your program to change their properties at run time or to get or set their value, as in the expression:

```
Table1LastName.AsString
```

In the Fields editor, you can also drag the fields to a different position to change their order. Proper field ordering is particularly important when you define a grid, which arranges its columns using this order.

note An even better feature of the Fields editor is that you can drag fields from this editor to the surface of a form and have Delphi automatically create a corresponding data-aware control (such as a DBEdit, a DBMemo, or a DBImage). The type of control created depends on the data type of the field and on eventual definitions in the Data Dictionary (as discussed in the next chapter). This is a very fast way to generate custom forms, and I suggest you try it out if you've never used it before. This is my preferred way to build database-related forms, much better than using the Database Form Wizard.

The Hierarchy of Field Classes

Before we look at an example, let's go over the use of the `TField` class. The importance of this component should not be underestimated. Although it is often used behind the scenes, its role in database applications is fundamental. As I already mentioned, even if you do not define specific objects of this kind, you can always access the `fields` of a table or a query using their `Fields` array property, the `FieldValues` indexed property, or the `FieldByName` method. Both the `Fields` property and the `FieldByName` function return an object of type `TField`, so you sometimes have to use the `as` operator to downcast their result to its actual type (like `TFloatField` or `TDateField`) before accessing specific properties of these subclasses.

The FieldAcc example is a simple extension of a form generated by the Database Form Wizard. I've added to it three speed buttons in the toolbar panel, accessing various Field properties at run time. The first button changes the formatting of the population column of the grid. To do this, we have to access the `DisplayFormat` property, a specific property of the `TFloatField` class. For this reason we have to write:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (Table1.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

When you set field properties related to data input or output, the change applies to every record in the table. When you set properties related to the `value` of the field, instead, you always refer to the current record only. For example, we can output the population of the current country in a message box by writing:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (string (Table1 ['Name']) +
    ': ' + string (Table1 ['Population']));
end;
```

When you access the value of a field, you can use a series of *As* properties to handle the current field value using a specific data type (if this is available, otherwise an exception is raised):

```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt[241];
AsString: string;
AsVariant: Variant;
```

These properties can be used to read or change the value of the field. Changing the value of a field is possible only if the DataSet is in edit mode. As an alternative to the *As* properties indicated above, you can access the value of a field by using its `Value` property, which is defined as a `Variant`.

Most of the other properties of the `TField` component, such as `Alignment`, `DisplayLabel`, `DisplayWidth`, and `Visible`, reflect elements of the field's user interface and are used by the various data-aware controls, particularly DBGrid. In the FieldAcc example, clicking the third speed button changes the `Alignment` of every field:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to Table1.FieldCount - 1 do
    Table1.Fields[I].Alignment := taCenter;
end;
```

This affects the output of the DBGrid, and of the DBEdit control I've added to the toolbar, which shows the name of the country. You can see this effect, along with the new display format, in Figure 9.8.

241 The type of `AsInteger` is now Integer.

Figure 9.8: The output of the FieldAcc example after the Center and Format buttons have been pressed. Image from the original book.

There are several field class types in the VCL. Delphi automatically uses one of them depending on the data definition in the database, when you open a table at run time or when you use the Fields editor at design time. Table 9.2 shows the complete list of subclasses of the `TField` class[242].

Table 9.2: The Subclasses of TField (the field types in bold are new to Delphi 5 and relate with ADO support)

Subclass	Base Class	Definition
`TADTField`	`TObjectField`	An ADT (Abstract Data Type) field, corresponding to an object field in an object relational database.
`TAggregateField`	`TField`	An aggregate field represents a maintained aggregate. It is used in the ClientDataSet component and discussed in Chapter 21.
`TArrayField`	`TObjectField`	An array of objects in an object relational database.
`TAutoIncField`	`TIntegerField`	Whole positive number connected with a Paradox auto-increment field of a table, a special field automatically assigned a different value for each record. Note that Paradox AutoInc fields do not always work perfectly, as discussed in the next chapter.

242 There have been only a few additions to the list of `TField` descendant data types, including `TSQLTimeStampField` and `TFMTBCDField`.

TBCDField	TNumericField	Real numbers, with a fixed number of digits after the decimal point.
TBinaryField	TField	Generally not used directly. This is the base class of the next two classes.
TBlobField	TField	Binary data and no size limit (BLOB stands for Binary Large OBject). The theoretical maximum limit is 2GB.
TBooleanField	TField	Boolean value.
TBytesField	TBinaryField	Arbitrary data with a large (up to 64K characters) but fixed size.
TCurrencyField	TFloatField	Currency values, with the same range as the new `Real` data type.
TDataSetField	TObjectField	An object corresponding to a separate table in an object relational database.
TDateField	TDateTimeField	Date value.
TDateTimeField	TField	Date and time value.
TFloatField	TNumericField	Floating-point numbers (8 byte).
TGraphicField	TBlobField	Graphic of arbitrary length.
TGuidField	TStringField	A field representing a COM Globally Unique Identifier, part of the ADO support.
TIDispatchField	TInterfaceField	A field representing pointers to `IDispatch` COM interfaces, part of the ADO support.
TIntegerField	TNumericField	Whole numbers in the range of long integers (32 bits).
TInterfaceField	TField	Generally not used directly. This is the base class of fields that contain pointers to interfaces (`IUnknown`) as data.
TLargeIntField	TIntegerField	Very large integers (64 bit).

TMemoField	TBlobField	Text of arbitrary length.
TNumericField	TField	Generally not used directly. This is the base class of all the numeric field classes.
TObjectField	TField	Generally not used directly. The base class for the fields providing support for object relational databases.
TReferenceField	TObjectField	A pointer to an object in an object relational database.

TSmallIntField	TIntegerField	Whole numbers in the range of integers (16 bits).
TStringField	TField	Text data of a fixed length (up to 8192 bytes).
TTimeField	TDateTimeField	Time value.
TVarBytesField	TBytesField	Arbitrary data, up to 64K characters. Very similar to the TBytes-Field base class.
TVariantField	TField	A field representing a variant data type, part of the ADO support.
TWideStringField	TStringField	A field representing a Unicode (16-bit per character) string.
TWordField	TIntegerField	Whole positive numbers in the range of words or unsigned integers (16 bits).

The availability of any particular field type, and the correspondence with the data definition, depends on the database in use. For example, InterBase doesn't support BCD, so you'll never get a BCDField for a table on the InterBase server. This is particularly true for the new field types that provide support for object relational databases.

Adding a Calculated Field

Now that you've been introduced to TField objects and seen an example of their run-time use, it is time to build a simple example based on the declaration of field objects at design time using the Fields editor. We can start again from the first example we've built, GridDemo, and add a calculated field. The COUNTRY.DB database table we are accessing has both the population and the area of each country, so we can use this data to compute the population density.

To build the new example, named Calc, select the Table component in the form, and open the Fields editor (using the form's SpeedMenu). In this editor, choose the Add command, and select some of the fields. (I've decided to include them all.) Now

select the Define command, and enter a proper name and data type (`TFloatField`) for the new calculated field, as you can see in Figure 9.9[243].

note It is obvious that as you create some field components at design time using the Fields editor, the fields you skip won't get a corresponding object. What might not be obvious is that the fields you skip will not be available even at run time, with `Fields` or `FieldByName`. When a program opens a table at run time, if there are no design-time field components, Delphi creates field objects corresponding to the table definition. If there are some design-time fields, however, Delphi uses those fields without adding any extra ones.

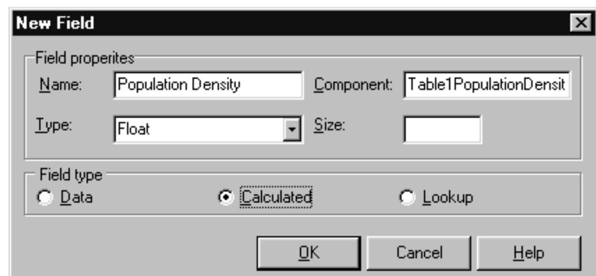


Figure 9.9: The definition of a calculated field in the Calc example. Image from the original book.

Of course, we also need to provide a way to calculate the new field. This is accomplished in the `OnCalcFields` event of the Table component, which has the following code (at least in a first version):

```
procedure TForm2.Table1CalcFields(DataSet: TDataSet);
begin
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value;
end;
```

Everything fine? Not at all! If you enter a new record and do not set the value of the population and area, or if you accidentally set the area to zero, the division will raise an exception, making it quite problematic to continue using the program. As an

243 There is now a second flavor of calculated fields, called internally calculated fields and available for component with memory storage (like FDMemTable and ClientDataSet). The difference is that in this second case the calculated value is kept in memory and used for display. Rather than calculating the value each time the record becomes active, the calculation is performed only the first time or when one of the other fields of the same record changes.

alternative, we could have handled every exception of the division expression and simply set the resulting value to zero:

```
try
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value;
except
  on Exception do
    Table1PopulationDensity.Value := 0;
end;
```

However, we can do even better. We can check if the value of the area is defined—if it is not null—and if it is not zero. It is better to avoid using exceptions when you can anticipate the possible error conditions:

```
if not Table1Area.IsNull and
    (Table1Area.Value <> 0) then
  Table1PopulationDensity.Value :=
    Table1Population.Value / Table1Area.Value
else
    Table1PopulationDensity.Value := 0;
```

The code of the `Table1CalcFields` method above (in each of the three versions) accesses some fields directly. This is possible because I used the Fields editor, and it automatically created the corresponding field declarations, as you can see in this excerpt of the interface declaration of the form:

```
type
  TCalcForm = class(TForm)
    Table1: TTable;
    Table1PopulationDensity: TFloatField;
    Table1Area: TFloatField;
    Table1Population: TFloatField;
    Table1Name: TStringField;
    Table1Capital: TStringField;
    Table1Continent: TStringField;
    procedure Table1CalcFields(DataSet: TDataset);
    ...
```

Each time you add or remove fields in the Fields editor, you can see the effect of your action immediately in the grid present in the form. Of course, you won't see the values of a calculated field at design time; they are available only at run time, because they result from the execution of compiled Pascal code.

Since we have defined some components for the fields, we can use them to customize some of the visual elements of the grid. For example, to set a display format that adds a comma to separate thousands, we can use the Object Inspector to change the `DisplayFormat` property of some field components to *"###,###,###"*. This change has an immediate effect on the grid at design time.

note The display format I've just mentioned (and used in the previous example) uses the Windows International Settings to format the output. When Delphi translates the numeric value of this field to text, the comma in the format string is replaced by the proper `ThousandSeparator` character. For this reason, the output of the program will automatically adapt itself to different International Settings. On computers that have the Italian configuration, for example, the comma is replaced by a period.

After working on the table components and the fields, I've customized the DBGrid using its `Columns` property editor. I've set the Population Density column to read-only and set its `ButtonStyle` property to `cbsEllipsis`, to provide a custom editor. When you set this value, a small button with an ellipsis is displayed when the user tries to edit the grid cell. Pressing the button invokes the `OnEditButtonClick` event of the DBGrid:

```
procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  MessageDlg (Format (
    'The population density (%.2n)'#13 +
    'is the Population (%.0n)'#13 +
    'divided by the Area (%.0n).'#13#13 +
    'Edit these two fields to change it.',
    [Table1PopulationDensity.AsFloat,
    Table1Population.AsFloat,
    Table1Area.AsFloat]),
    mtInformation, [mbOK], 0);
end;
```

Actually, I haven't provided a real editor, but rather a message describing the situation, as you can see in Figure 9.10, which shows the values of the calculated fields. To create an editor, you might build a secondary form to handle special data entries.

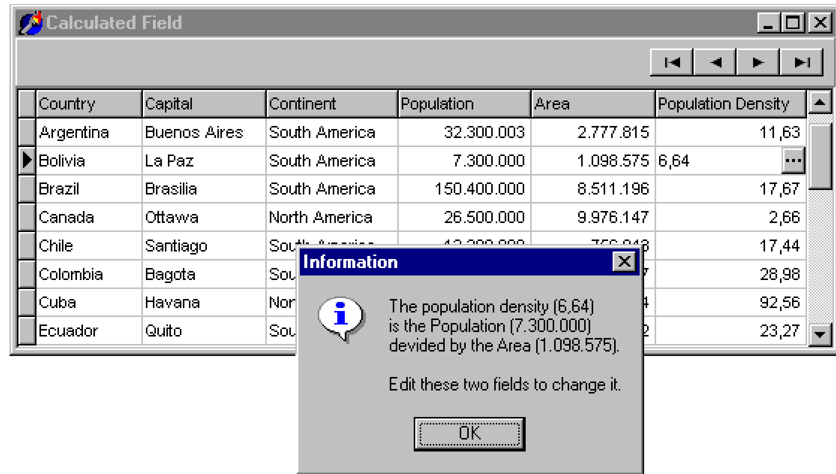


Figure 9.10: The output of the Calc example. Notice the Population Density calculated column, the ellipsis button, and the message displayed when you select it. Image from the original book.

Searching and Adding the Fields of a Table

`TField` components can be used to access data and manipulate a table at run time. We have seen only a limited example of direct data access; in the previous example, we used the value of two fields to calculate a third one. Now we will build some simple examples that will allow us to use the fields to search elements in a table, operate on the values, and access information about the tables of a database. There are many more possible uses of field components, but this should give you an idea of what can be done.

Looking for Records in a Table

For this example we need a new form, this time connected to EMPLOYEE.DB, another of the sample Delphi tables. To prepare the form, you can use the Database

Form Wizard or drag the fields from the Fields editor, an operation that will automatically add the corresponding labels[244].

note If you place the data-aware edit boxes inside a scroll box aligned to the client area, you can freely resize the form without any problems. When the form becomes too small, scroll bars will appear automatically in the area holding the edit boxes.

Instead of the default Delphi navigator component, we can add a standard Toolbar control and connect the buttons to some of the predefined dataset actions available in the ActionList component. I've simply added an ImageList to the form and connected it to the ActionList, to let the image list receive the images for the standard actions. Then I've added to the ActionList the predefined standard actions `TDataSetFirst`, `TDataSetLast`, `TDataSetNext`, and `TDataSetPrior`, plus two normal actions to host the custom search code.

Now you can simply connect the buttons of the toolbar with the corresponding actions and add to the toolbar an edit box where the user can enter the name to search for, as you can see in Figure 9.11. The buttons will carry out the proper action when pressed, and they will be disabled when the data set is at its beginning or end.

The searching capabilities are activated by the two buttons connected with custom actions. The first button is connected with the `ActionGoto`, used for an exact match, and the second with `ActionGoNear` for a nearest search. In both cases, we want to compare the text in the edit box with the LastName fields of the Employee table.

Figure 9.11: An example of a best-match search using the Search application. Image from the original book.

244 In terms of the database form wizards, this has never been ported to recent data access libraries and is not available today. Notice, instead, that the ability to drag fields from the field editor to the form to display a matching UI control is still available today. This is a very handy and fast way to build a UI, but it's little known and rarely used by Delphi developers. I'm really not sure why, considering it can be configured (in code) including the UI controls mapping.

The Table component has methods to accomplish this look up, such as `GotoKey`, `FindKey`, `GotoNearest`, `FindNearest`, and `Locate`. The `Locate` method uses the optimal access: if an index is available it uses the index for a faster search; otherwise, it does a plain sequential search. To use the first group of search methods, you need to set the `IndexFieldNames` property of the Table component to the proper value. (In this case, you can directly select the string *LastName;FirstName* in the drop-down list.)

The Find Methods

When the index is properly set, we can make the actual search. The simplest approach is to use the `FindNearest` method for the approximate search and the `FindKey` method to look for an exact match:

```
// goto
Table1.FindNearest ([EditName.Text]);

// go near
if not Table1.FindKey ([EditName.Text]) then
  MessageDlg ('Name not found', mtError, [mbOk], 0);
```

Both Find methods use as parameters an array of constants. Each array element corresponds to an indexed field. In our case, we pass only the value for the first field of the index, so the other fields will not be considered.

The Goto Methods

The `FindNearest` and `FindKey` methods are easy to use. To better understand how they work, though, we can look at the usage of the `GotoNearest` and `GotoKey` methods. These last two methods, in fact, map very closely to the actual low-level BDE calls. The simpler of the two is the best-guess search of the `GotoNearest` speed button:

```
// go near
Table1.SetKey;
Table1 ['LastName'] := EditName.Text;
Table1.GotoNearest;
```

As you can see in this code, each search on a table is done in three steps: start up the search state of the table, set a target value for each lookup field, and start the lookup process, by moving the current record to the requested position.

The code used to call the other search method, using an exact-match algorithm, is similar. The differences are in two statements:

```
// go to
Table1.SetKey;
Table1 ['LastName'] := EditName.Text;
Table1.KeyFieldCount := 1;
if not Table1.GotoKey then
  MessageDlg ('Name not found', mtError, [mbOK], 0);
```

As I've mentioned before, this code requires a proper index for the table. Notice the value set for the `KeyFieldCount` property, which indicates that I want to use just the first of the two fields that contribute to the index. The second difference is that the `GotoNearest` procedure always succeeds, moving the cursor to the closest match (a closest match always exists, even if it is not very close). On the other hand, the `GotoKey` method fails if no exact match is available, and you can check the return value of this function, and eventually warn the user of the error.

`FindKey` performs exactly the same steps as the `GotoKey` version of the above code. `FindKey` and `GotoKey` provide equivalent functionality, except that the former is easier to use and the latter provides for better error handling.

The Locate Method

If the table doesn't have an index on the field you are searching for (at least for local tables), you cannot use the two techniques above. A third, more general, technique is to use the `Locate` method. This approach is very handy in any case, because if there is an index on the field you are searching, `Locate` automatically uses it; otherwise it does a plain (and slower) search.

Using `Locate` is quite simple: Just provide a first string with the fields you want to search and a variant with the value or values you are searching for. To search for multiple fields, you need an array of values. (You can create one with the `VarArrayCreate` call.) Here is an example of its use, extracted again from the Search program:

```
// goto
if not Table1.Locate ('LastName', EditName.Text, []) then
  MessageDlg ('Name not found', mtError, [mbOk], 0);
```

The Total of a Table Column

So far in our examples, the user can view the current contents of a database table and manually edit the data or insert new records. Now we will see how we can change some data in the table through the program code. The idea behind this example is quite simple. The Employee table we have been using has a Salary field.

A manager of the company could indeed browse through the table and change the salary of a single employee. But what will be the total salary expense for the company? And what if the manager wants to give a 10 percent salary increase (or decrease) to everyone?

These are the two aims of the Total example, which is an extension of the previous program. The toolbar of this new example has two more buttons (and two related actions) and a SpinEdit component. There are few other minor changes from the previous example. I opened the Fields Editor of the table and removed the `Table1Salary` field, which was defined as a `TFloatField`. Then I selected the New Field command and added the same field, with the same name, but using the `TCurrencyField` data type. This is not a calculated field; it's simply a field converted into a new (but equivalent) data type. Using this new field type the program will default to a new output format, suitable for currency values.

Now we can turn our attention to the code of this new program. First, let's look at the code of the total action. This action lets you calculate the sum of the salaries of all the employees, then edit some of the values, and compute a new total. Basically, we need to scan the table, reading the value of the `Table1Salary` field for each record:

```
begin
  Table1.First;
  while not Table1.EOF do
  begin
    Total := Total + Table1Salary.Value;
    Table1.Next;
  end;
end
```

This code works, as you can see from the output in Figure 9.12, but it has a number of problems. One problem is that the record pointer is moved to the last record, so the previous position in the table is lost. To avoid this problem, we need to store the current position of the record pointer in the table and restore it at the end. This can be accomplished using a *table bookmark,* a special variable storing the position of a record in a database table. The traditional approach is to declare a variable of the `TBookmark` data type, and initialize it while getting the current position from the table:

```
var
  Bookmark: TBookmark;
begin
  Bookmark := Table1.GetBookmark;
```

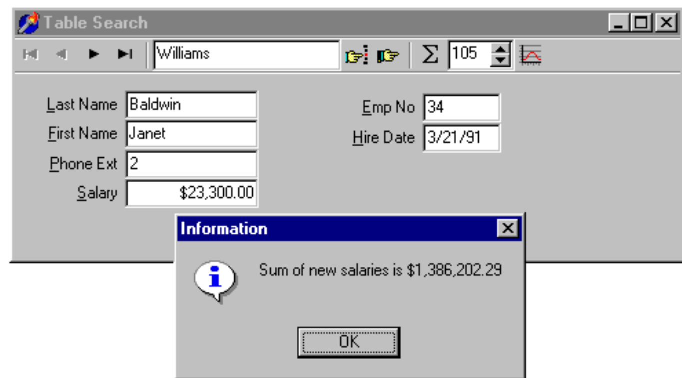


Figure 9.12: The output of the Total program, showing the total salaries of the employees. Image from the original book.

At the end of the `ActionTotalExecute` method, we can restore the position and delete the bookmark with the following two statements:

```
Table1.GotoBookmark (Bookmark);
Table1.FreeBookmark (Bookmark);
```

As a better alternative, we can use the `Bookmark` property of the `TDataset` class, which refers to a bookmark that is disposed of automatically. (This is technically implemented as an *opaque string*, a structure subject to string lifetime management, but it is not a string, so you're not supposed to look at what's inside it.) This is how you can modify the code above:

```
var
  Bookmark: TBookmarkStr;
begin
  Bookmark := Table1.Bookmark;
  ...
  Table1.Bookmark := Bookmark;
```

Another side effect of the program is that, although we will restore the record pointer to the initial position, we might see the records scrolling while the routine browses through the data. This can be avoided by disabling the controls connected with the table during browsing. The table has a `DisableControls` method we can call before the `while` loop starts and an `EnableControls` method we can call at the end, after the record pointer is restored.

note Disabling the data-aware controls connected with a table during long operations not only improves the user interface (since the output is not changing constantly), it also speeds up the program considerably. In fact, the time spent to update the user interface is much greater than the time spent performing the calculations. To test this, try commenting out the `DisableControls` and `EnableControls` methods of the Total example, and see the speed difference.

Finally, we face some dangers from errors in reading the table data, particularly if the program were reading the data from a server using a network. If any problem occurs while retrieving the data, an exception takes place, the controls remain disabled, and the program cannot resume its normal behavior. So we should use a `try-finally` block. Actually, if you want to make the program 100 percent error-proof you should use two nested `try-finally` blocks. Including this change and the two discussed above, here is the resulting code:

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
  Bookmark: TBookmarkStr;
  Total: Real;
begin
  Bookmark := Table1.Bookmark;
  try
    Table1.DisableControls;
    Total := 0;
    try
      Table1.First;
      while not Table1.EOF do
      begin
        Total := Total + Table1Salary.Value;
        Table1.Next;
      end;
    finally
      Table1.EnableControls;
  end
  finally
    Table1.Bookmark := Bookmark;
  end;
  MessageDlg ('Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;
```

I've written this code to show you an example of a loop to browse the contents of a table, but keep in mind that there is an alternative approach based on the use of a SQL query returning the sum of the values of a field.

note When you use a SQL server, the speed advantage of a SQL call to compute the total can be very large, since you don't need to move all the data of each field from the server to the client computer. The server sends the client only the final result.

Editing a Table Column

The code of the increase action is similar to the one we have just seen. The `ActionIncreaseExecute` method also scans the table, computing the total of the salaries, as the previous method did. Although it has just two more statements, there is a key difference. When you increase the salary, you actually change the data in the table. The two key statements are within the `while` loop:

```
while not Table1.EOF do
begin
  Table1.Edit;
  Table1Salary.Value := Round (Table1Salary.Value *
    SpinEdit1.Value) / 100;
  Total := Total + Table1Salary.Value;
  Table1.Next;
end;
```

The first statement brings the table into edit mode, so that changes to the fields will have an immediate effect. The second statement computes the new salary by multiplying the old one by the value of the SpinEdit component (by default, 105) and dividing it by 100. That's a 5 percent increase, although the values are rounded to the nearest dollar. With this program, you can change salaries by any amount—even double the salary of each employee—with the click of a button.

note Notice that the table enters the edit mode every time the `while` loop is executed. This is because in a dataset, edit operations can take place only one record at a time. You must finish the edit operation, calling `Post` or moving to a different record as in the code above. At that time, if you want to change another record, you have to enter edit mode once more.

Database Application with Standard Controls

Although it is generally faster to write Delphi applications based on data-aware controls, this is certainly not required. When you need to have very precise control over

the user interface of a database application, you might want to customize the transfer of the data from the field objects to the visual controls. My personal view is that this is necessary only in very specific cases, as you can customize the data-aware controls extensively by setting the properties and handling the events of the field objects. However, trying to work without the data-aware controls should help you understand the default behavior of Delphi, and it will help me introduce some of the database-related events (discussed in the sections “Database Events” and “Field Events”).

The development of an application not based on data-aware controls can follow two different approaches. You can mimic the standard Delphi behavior in code, possibly departing from it in specific cases, or you can go for a much more customized approach. I’ll demonstrate the first technique in the NonAware example and the latter in the SendToDb example.

Mimicking Delphi Data-Aware Controls

If you want to build an application that doesn’t use data-aware controls but behaves like a standard Delphi application, you can simply write event handlers for the operations that would be performed automatically by data-aware controls[245]. Basically you need to place the data set in edit mode as the user changes the content of the visual controls, and update the field objects of the data set as the user exits from the controls, moving the focus to another element.

note This approach can be handy for integrating a control that’s not data-aware, such as a DateTimePicker component, into a standard application.

The other element of the NonAware example is another list of buttons corresponding to some of those in the DBNavigator control. The five buttons are connected to five methods of the table component: `Next`, `Previous`, `Insert`, `Cancel`, and `Delete`. This is a summary of the Delphi form file:

```
object Form1: TForm1
  Caption = 'Non Aware'
  // 5 labels omitted
  object EditName: TEdit
    Text = 'EditName'
    OnExit = EditNameExit
```

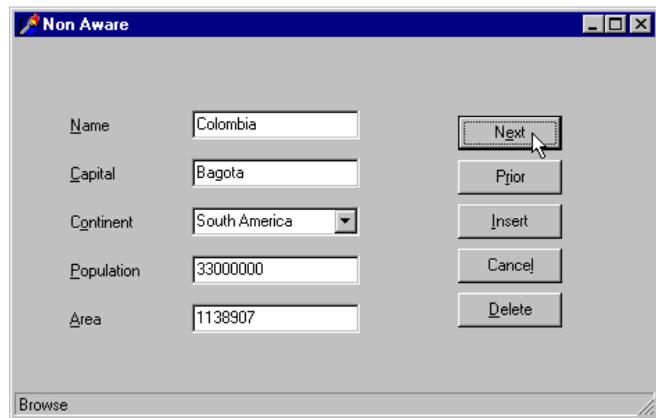
245 The additional alternative available today is the use of Live Bindings to associated database fields and regular (non data-aware) Ui controls.

```
    OnKeyPress = EditKeyPress
  end
  object EditCapital: TEdit...
  object EditPopulation: TEdit...
  object EditArea: TEdit...
  object ComboContinent: TComboBox
    Items.Strings = (
      'South America'
      'North America'
      'Europe'
      'Asia'
      'Africa')
    Text = 'ComboContinent'
    OnDropDown = ComboContinentDropDown
    OnExit = ComboContinentExit
    OnKeyPress = EditKeyPress
  end
  // 5 buttons omitted
  object StatusBar1: TStatusBar
    SimplePanel = True
  end
  object DataSource1: TDataSource
    DataSet = Table1
    OnStateChange = DataSource1StateChange
    OnDataChange = DataSource1DataChange
  end
  object Table1: TTable
    Active = True
    AfterInsert = Table1AfterInsert
    BeforePost = Table1BeforePost
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
    // 5 field objects omitted
  end
end
```

As you can see in the listing above, the program has several event handlers we've not used for past applications using data-aware controls. First of all, we have to show the data of the current record in the visual controls (as in Figure 9.13), by handling the `OnDataChange` event of the `DataSource1` component:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:
TField);
begin
  EditName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
  EditPopulation.Text := Table1Population.AsString;
end;
```

Figure 9.13: The output of the NonAware example in Browse mode. The program manually fetches the data every time the current record changes. Image from the original book.



The handler of the `OnStateChange` event of the control, instead, uses some code we've already seen in the GridDemo example. This time, the status of the table is displayed in a status bar control. As the user starts typing in one of the edit boxes or drops down the combo box list, the program sets the table in edit mode:

```
procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
  if not (Table1.State in [dsEdit, dsInsert]) then
    Table1.Edit;
end;
```

This method is connected with the `OnKeyPress` event of the five components and is similar to the `OnDropDown` event handler of the combo box. As the user leaves one of

the visual controls, the handler of the OnExit event copies the data to the corresponding field, as in this case:

```
procedure TForm1.EditCapitalExit(Sender: TObject);
begin
  if (Table1.State = dsEdit) or (Table1.State = dsInsert) then
    Table1Capital.AsString := EditCapital.Text;
end;
```

The operation takes place only if the table is in Edit mode; that is, only if the user has typed in this or another control. This is not really ideal, because extra operations are done even if the text of the edit box didn't change, but the extra steps happen fast enough not to be a concern. For the first edit box, we check the text before copying it, raising an exception if the edit box is empty:

```
procedure TForm1.EditNameExit(Sender: TObject);
begin
  if (Table1.State = dsEdit) or (Table1.State = dsInsert) then
    if EditName.Text <> '' then
      Table1Name.AsString := EditName.Text
    else
    begin
      EditName.SetFocus;
      raise Exception.Create ('Undefined Country');
    end;
end;
```

An alternative approach for testing the value of a field is to handle the BeforePost event of the data set (with the effect shown in Figure 9.14). Keep in mind that in this example the posting operation is not handled by a specific button but takes place as soon as a user moves to a new record or inserts a new one:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  if Table1Area.Value < 100 then
    raise Exception.Create ('Area too small');
end;
```

In each of these cases, an alternative to raising an exception is to set a default value. However, if a field has a default value it is better to set it up front, so that a user can see which value will be sent to the database. To accomplish this, you can handle the AfterInsert event of a data set, which is fired immediately after a new record has been created (we could have used the OnNewRecord event, as well):

```
procedure TForm1.Table1AfterInsert(DataSet: TDataSet);
begin
  Table1Continent.Value := 'Asia';
end;
```

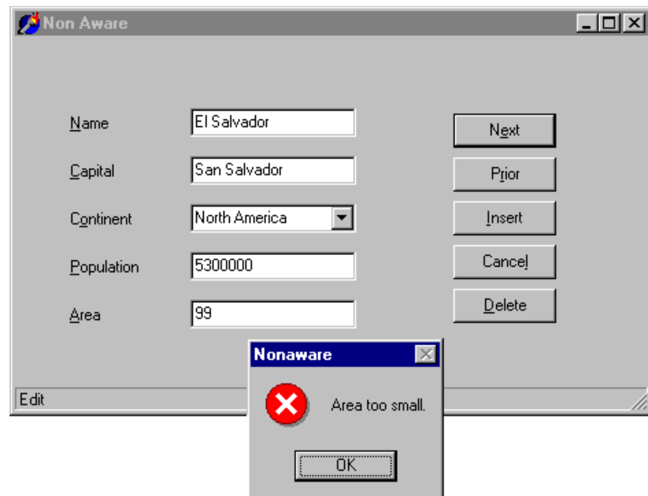


Figure 9.14: The error message displayed when the value of the area is too small. Image from the original book.

Sending Requests to the Database

You can further customize the user interface of your application if you decide not to handle the same sequence of editing operations as in standard Delphi data-aware controls. This allows you complete freedom, although there might be some side effects (such as limited ability to handle concurrency, which is something I'll discuss in the next chapter).

For this new example, I've replaced the first edit box with another combo box, and replaced all the buttons related to table operations (which corresponded to DBNavigator buttons) with two custom ones, used to get the data from the database and send an update to it. To underline the difference of this example, I've even removed the DataSource component.

The `GetData` method, connected with the corresponding button, simply gets the fields corresponding to the record indicated in the first combo box:

```
procedure TForm1.GetData;
begin
  Table1.FindNearest ([ComboName.Text]);
  ComboName.Text := Table1Name.AsString;
  EditCapital.Text := Table1Capital.AsString;
  ComboContinent.Text := Table1Continent.AsString;
  EditArea.Text := Table1Area.AsString;
```

```
  EditPopulation.Text := Table1Population.AsString;
end;
```

This method is called whenever the user presses the button, selects an item of the combo box, or presses the Enter key while in the combo box:

```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
  GetData;
end;

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = #13 then
    GetData;
end;
```

To make this example work smoothly, at start-up the combo box is filled with all the names of the countries of the table:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // fill the list of names
  Table1.Open;
  while not Table1.Eof do
  begin
    ComboName.Items.Add (Table1Name.AsString);
    Table1.Next;
  end;
end;
```

With this approach, the combo box becomes a sort of selector of the record, as you can see in Figure 9.15. Notice that thanks to this selection, the program doesn't need navigational buttons.

Finally, the user can change the values of the controls and press the Send button. The code to be executed depends on whether the operation is an update or an insert. We can determine this by looking at the name (although with this code, a wrong name cannot be modified any more):

Figure 9.15: In the SendToDb example, you can select the record you want to see in a combo box. Image from the original book.

```
procedure TForm1.SendData;
begin
  // raise an exception if there is no name
  if ComboName.Text = '' then
    raise Exception.Create ('Insert the name');

  // check if the record is already in the table
  if Table1.FindKey ([ComboName.Text]) then
  begin
    // modify found record
    Table1.Edit;
    Table1Capital.AsString := EditCapital.Text;
    Table1Continent.AsString := ComboContinent.Text;
    Table1Area.AsString := EditArea.Text;
    Table1Population.AsString := EditPopulation.Text;
    Table1.Post;
  end
  else
  begin
    // insert new record
    Table1.InsertRecord ([ComboName.Text,
      EditCapital.Text, ComboContinent.Text,
      EditArea.Text, EditPopulation.Text]);
    // add to list
    ComboName.Items.Add (ComboName.Text)
  end;
```

Before sending the data to the table, you can do any sort of validation test on the values. In this case, it doesn't make much sense to handle the events of the database components, because we have full control on when the update or insert operation is done.

Database Events

To further illustrate how you can use the events of a database application, I've written a simple program that logs all the events being fired. This program handles all of the events of a table and a data source component (although some of these events won't actually be executed, unless you add some extra code, as described later). For each event, I simply send its description to a list box, with the effect you can see in Figure 9.16.

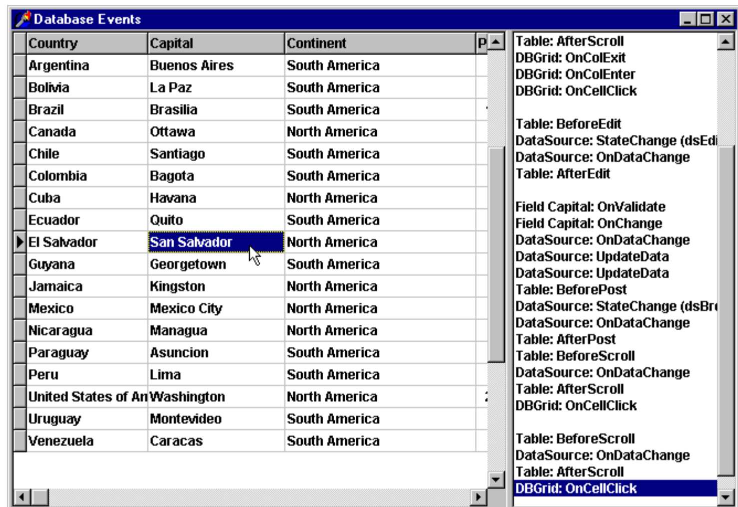


Figure 9.16: The output of the DbEvts program, which logs all the events related to database components. Image from the original book.

Most of the event handlers simply display the name of the component and that of the event, as in

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
  AddToList ('Table: AfterEdit');
end;
```

The field events are slightly more complex, but they use a single handler for the various field components:

```
procedure TForm1.FieldChange(Sender: TField);
begin
  AddToList ('Field ' + Sender.FieldName + ': OnChange');
end;
```

The form's `AddToList` method adds a new item to the list box and selects it, automatically scrolling the list if required:

```
procedure TForm1.AddToList(Str: string);
begin
  // add item and select it
  Listbox1.ItemIndex := Listbox1.Items.Add (Str);
end;
```

Finally, the program has a pop-up menu connected to the list box to clear the list or save the items to a file. The menu also has a command you can use to add a blank line, thus separating blocks of events. This operation is also done automatically by a timer, which adds a blank line to the list box unless the last item is already an empty string. This makes the output more readable, as you can see in Figure 9.16.

It is very important to study the output of this program as well as its code. You can try doing all the various operations on the table using the DBGrid, such as inserting, editing, and deleting records, and see the corresponding effect in terms of events fired by the VCL components. To see even more events, you can set the `Filtered` property of the table to `True`, define a calculated field, try to cause errors (for example, by duplicating the value of the name field), add a check box to open or close the table, and so forth.

Field Events

The DbEvts program shows the calls to the `OnChange` and `OnValidate` events of the field objects. Two other events, `OnSetText` and `OnGetText`, are not shown, because the handlers of these events are not simply called to indicate that an operation occurred. On the contrary, their event handler must perform the operation of getting data from or setting it to the corresponding field objects.

These two events are quite special, and their use is not as simple as it might seem at first sight. For this reason, they require a separate example, named FldText. This is only a slight revision of the DbAware example described earlier in this chapter, replacing the DBRadioGroup control with a DBListbox control. The problem is that a DBListBox control directly connects with a string field, while I want to connect it with an integer field, with each value indicating an option. Of course, I don't want a user to see or select a number, so I have to map the numbers stored in the database to the strings visible on the screen. In the earlier example, the DBRadioGroup control provided that mapping. Now I have to use an alternative approach.

In the FldText example, the Department field has two handlers for the `OnGetText` and `OnSetText` events. In the `OnGetText` event handler you can extract the numeric value of the `Sender` field and set the value of the `Text` reference parameter:

```
procedure TDbaForm.Table1DepartmentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  case Sender.AsInteger of
    1: Text := 'Sales';
    2: Text := 'Accounting';
    3: Text := 'Production';
    4: Text := 'Management';
  else
    Text := '[Error]';
  end;
end;
```

note In the code of the `OnGetText` event handler you cannot refer to the text of the field, for example, using the `DisplayText` property or the `GetData` method, since they would call the `OnGetText` event, in an infinite recursion.

In the `OnSetText` event handler you can examine the string and decide the value of the field, according to the conversion rule, in this case a simple mapping of values done with an `if-then-else` statement:

```
procedure TDbaForm.Table1DepartmentSetText(Sender: TField;
  const Text: String);
begin
  if Text = 'Sales' then
    Sender.Value := 1
  else if Text = 'Accounting' then
    Sender.Value := 2
  else if Text = 'Production' then
    Sender.Value := 3
  else if Text = 'Management' then
    Sender.Value := 4
  else
    raise Exception.Create (
      'Error in Department field conversion');
end;
```

The effect is that not only is the value visible in the DBListBox (as you can see in Figure 9.17), it also shows up in the DBGrid. By contrast, in the DbAware example, the grid displayed the numeric value.

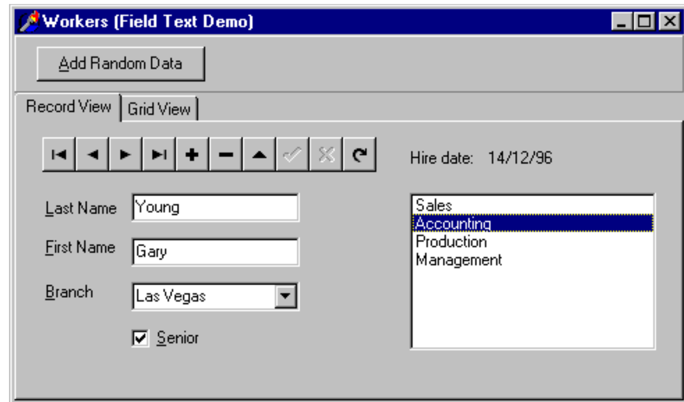


Figure 9.17: The output of the FldText example, which demonstrates the use of the OnGetText and OnSetText events of the field objects. Image from the original book.

Editing Dates with a Calendar

As a final example of the use of non–data-aware controls, the DbDates application shows how to use a MonthCalendar component to handle dates with a nice graphical component instead of a plain edit box. This example is based on the Events table from the DBDemos database, which lists Olympic events.

This example uses (for the first time) a DBImage control, with the following settings (whose effect is illustrated in Figure 9.18):

```
object DBImage1: TDBImage
  DataField = 'Event_Photo'
  DataSource = DataSource1
  Stretch = True
end
```

note Graphic, memo, and BLOB fields in Delphi are handled exactly like other fields. Just connect the proper editor or viewer, and most of the work is done behind the scenes by the system.

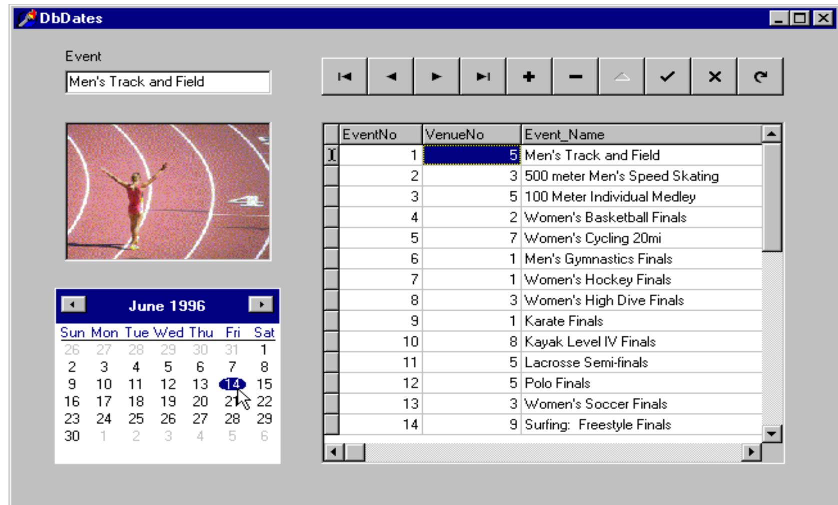


Figure 9.18: The selection of a date with the monthly calendar. Image from the original book.

Although the DBImage control works with no extra effort on our part, we must connect the MonthCalendar control with the corresponding field by handling two events of the DataSource control:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:
TField);
begin
  MonthCalendar1.Date := Table1Event_Date.Value;
end;

procedure TForm1.DataSource1UpdateData(Sender: TObject);
begin
  Table1Event_Date.Value := MonthCalendar1.Date;
end;
```

Besides copying the data back and forth, with the code listed above, the program must also put the table into edit mode as the user clicks the calendar control. The most obvious approach is to write a handler for the `OnClick` event of the control:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  Table1.Edit;
end;
```

However, this code doesn't work properly. As you set the table in edit mode, the `OnDataChange` event is executed once more, resetting the selection in the calendar. The overall effect is that the user's first click doesn't change the selection. To avoid this problem we can set a flag in the `OnClick` event handler and test it in the

`OnDataChange` event handler, or we can temporarily disconnect the second event handler. In the following code, I've taken the second approach:

```
procedure TForm1.MonthCalendar1Click(Sender: TObject);
begin
  // disconnect handler
  DataSource1.OnDataChange := nil;
  // set table in edit mode
  Table1.Edit;
  // reconnect handler
  DataSource1.OnDataChange := DataSource1DataChange;
end;
```

Exploring the Tables of a Database

In our examples so far, we have always accessed a database table by setting its name at design time. But what if you do not know which table your program will be connected to? At first, you might think that if you do not know the details of the database at design time, you won't be able to create forms and operate on the table. This is not true. Setting everything at design time is certainly easier. Changing almost anything at run time requires you to write more code. This is what I've done in the next example, called Tables, which demonstrates how to access the list of databases available to the Borland Database Engine[246], how to access the list of the tables for each database, and how to select which fields to view from a specific table.

Choosing a Database and a Table at Run Time

For the Tables example, I've prepared a form with a combo box you can use to select a database and a list box you can use to select a table of that database. The form also hosts a DBGrid, which can be connected with the selected database table. You can see the output of this program in Figure 9.19.

When the program starts, it fills the combo box, fills the list box (forcing the selection of the first item of the combo box), and then shows a table in the DBGrid (simulating the selection of the first item of the list box):

---

246 Something similar could be done by picking a stored FireDAC table file to use with the FD-MemTable component.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Session.GetDatabaseNames (ComboBox1.Items);
  // force an initial list in the listbox
  ComboBox1.Text := 'DBDEMOS';
  ComboBox1Change (Self);
  // force an initial selection in the DBGrid
  ListBox1.ItemIndex := 0;
  ListBox1Click (Self);
end;
```

Figure 9.19: The output of the Tables program, which shows the data of a table selected at run time. Image from the original book.

The key element is the call to the `GetDatabaseNames` procedure of the `Session` global object[247]. An object of class `TSession` is automatically defined and initialized by each Delphi database application (even if you don't define one), and to access its methods, you only need to refer to the DBTables unit in the `uses` statement. When the combo box is filled, the program immediately selects one of the databases and then triggers the `ComboBox1Change` event handler, which uses another method of the `TSession` class, `GetTableNames`. This method has five parameters: the name of a database, a filter string, two Boolean values indicating whether to include the table file extensions (for local tables only) and whether to include system tables in the list (for SQL databases only), and the `TStringList` that will be filled with the names of the tables. Here is the code the program executes when the user selects an item in the combo box:

```
procedure TMainForm.ComboBox1Change(Sender: TObject);
```

247 This isn't available any more, as it was specific to the BDE.

```
begin
  Session.GetTableNames (ComboBox1.Text, '',
    True, False, ListBox1.Items);
end;
```

In the `FormCreate` method, a further step is automatically executed at start-up; the program fills the DBGrid as if a list box item had been selected:

```
procedure TMainForm.ListBox1Click(Sender: TObject);
begin
  Table1.Close;
  Table1.DatabaseName := ComboBox1.Text;
  Table1.Tablename := Listbox1.Items [Listbox1.ItemIndex];
  Table1.Open;
  Caption := Format ('Table: %s - %s',
    [Table1.DatabaseName, Table1.Tablename]);
end;
```

Viewing Multiple Tables

The program allows a user to see the content of any table. As a further extension, when the user double-clicks the list box, the program displays the grid in a separate form. This allows the user to open multiple modeless forms and see different tables at once, as you can see in Figure 9.20.

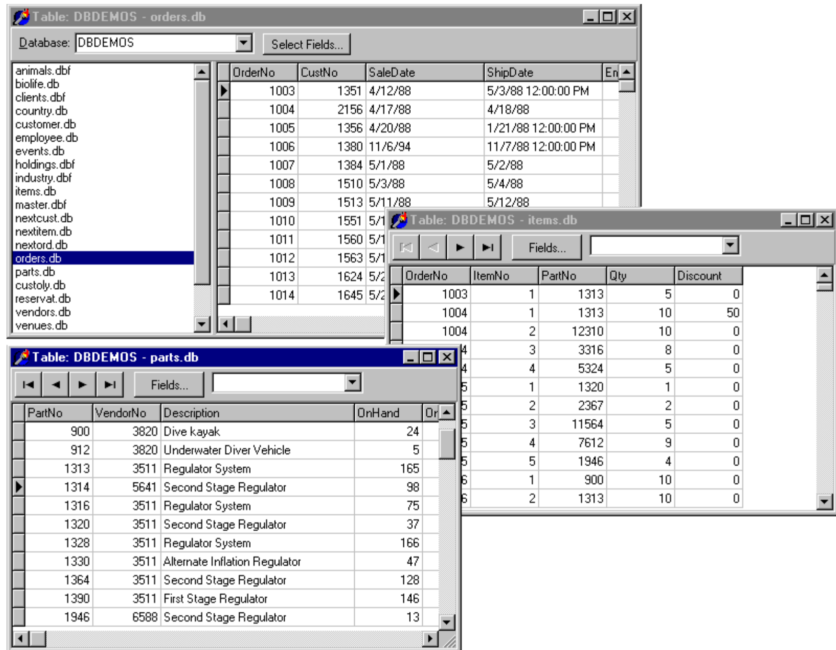


Figure 9.20: The Tables program can be used to open two or more grid-based table viewers. Images from the original book.

When the user double-clicks the list box in the main form, the code creates a `TGridForm` object, connects the `Table1` component of this form to the proper database and table, and shows the form:

```
procedure TMainForm.ListBox1DblClick(Sender: TObject);
var
  GridForm: TGridForm;
begin
  GridForm := TGridForm.Create (Self);
  {connect the table component to the selected
  table and activate it}
  GridForm.Table1.DatabaseName := ComboBox1.Text;
  GridForm.Table1.TableName :=
    Listbox1.Items [Listbox1.ItemIndex];
  try
    GridForm.Table1.Open;
    GridForm.Show;
  except
    GridForm.Close;
  end;
end;
```

note Notice that the code above simply creates the form and never destroys it. It is the responsibility of the form to delete itself in its `OnClose` event handler by setting the `Action` reference parameter to `caFree`.

When the secondary form is created, the program fills a combo box with the names of the fields of the table. However, this code can't go in the `OnCreate` event of the form, because the form is created before its `Table1` component is properly set up. Instead of adding a custom method and calling it, I've used the `OnShow` event handler, which also sets the caption of the form using the name of the table and the database:

```
procedure TGridForm.FormShow(Sender: TObject);
var
  I: Integer;
begin
  Caption := Format ('Table: %s - %s',
    [Table1.DatabaseName, Table1.TableName]);

  // fill the combo box with the names of the fields
  ComboBox1.Items.Clear;
  for I := 0 to Table1.FieldCount - 1 do
    ComboBox1.Items.Add (Table1.Fields[I].FieldName);
end;
```

note A possible extension to this program would be to generate a form based on data-aware controls, chosen depending on the type of field. You can find a Database Form Wizard capable of generating similar forms on my Web site, `www.marcocantu.com`.

What is the purpose of this combo box? Each time a user selects an element, the corresponding field is either shown or hidden, depending on its current state:

```
procedure TGridForm.ComboBox1Change(Sender: TObject);
begin
  // toggle the visibility of the field
  Table1.FieldByName (ComboBox1.Text).Visible :=
    not Table1.FieldByName (ComboBox1.Text).Visible;
end;
```

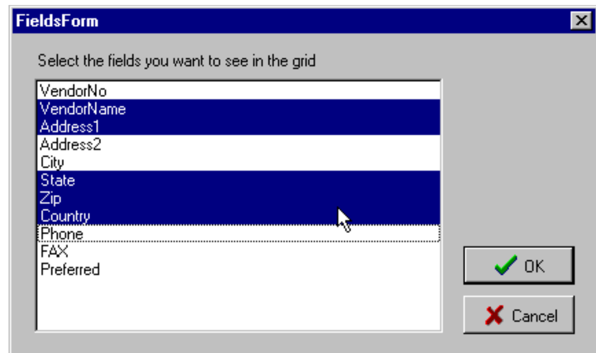
Notice the use of the `FieldByName` method to retrieve the field using the current selection of the combo box and the use of the `Visible` property. Once a field becomes invisible, it is immediately removed from the grid associated with the table. Therefore, by simply setting this property, we change the grid automatically.

The combo box I've placed in the toolbar of the GridForm works, but if you need to select several fields in a big table, it is slow and error-prone. As an alternative, I've

created a field-editor form, which is used both by the main and by the secondary form. This is the third form of the Tables example, named FieldsForm.

This form is displayed as a modal dialog box, so we can use a single global object every time. The new form has no code of its own. When the form is activated, its multiple-selection list box is filled with the names of the fields of the table. At the same time, the code selects the list box items corresponding to visible fields, as you can see in Figure 9.21.

Figure 9.21: The list box can be used to select the table fields to show in the grid. Image from the original book.



The user can toggle the selection of each item in this list box while the modal form is active. When it is closed, the other form retrieves the values of the selected items and sets the Visible property of the fields accordingly. Here is the complete code of this method:

```
procedure TGridForm.SpeedButton1Click(Sender: TObject);
var
  I: Integer;
begin
  FieldsForm.FieldsList.Clear;
  for I := 0 to Table1.FieldCount - 1 do
  begin
    FieldsForm.FieldsList.Items.Add (
      Table1.Fields [I].FieldName);
    if Table1.Fields [I].Visible then
      FieldsForm.FieldsList.Selected [I] := True;
  end;
  if FieldsForm.ShowModal = mrOK then
     for I := 0 to Table1.FieldCount - 1 do
       Table1.Fields.Visible [I] :=
         FieldsForm.FieldsList.Selected [I];
  FieldsForm.FieldsList.Clear;
end;
```

This code ends the description of this example. We have seen that you can write database applications that do most of the work at run time, although this approach is slightly more complex.

A Multi-Record Grid

So far we have seen that you can either use a grid to display a number of records of a database table or build a form with specific data-aware controls for the various fields, accessing the records one by one. There is a third alternative: use a multi-record object (a DBCtrlGrid[248]), which allows you to place many data-aware controls in a small area of a form and automatically duplicate these controls for a number of records.

Here is what we can do to build the Multi1 example. Create a new blank form, place a Table component and a DataSource component in it, and connect them to the COUNTRY.DB table. Now place a DBCtrlGrid on the form, set its size and the number of rows and columns, and place two edit components connected with the Name and Capital fields of the table. To place these DBEdit components, you can also open the Fields editor and drag the two fields to the control grid. At design time, you simply work on the active portion of the grid (see Figure 9.22, on the right), and at run time, you can see these controls replicated a number of times (see Figure 9.22, on the left).

Figure 9.22: The DBCtrlGrid of the Multi1 example at design time (on the right) and at run time (on the left). Images from the original book.

Here are the most important properties of the DBCtrlGrid object and the other components of this example:

248 As mentioned early in this chapter, this component still exists but it's not commonly used.

```
object Form1: TForm1
  Caption = 'Multi Record Grid'
  object DBCtrlGrid1: TDBCtrlGrid
    ColCount = 2
    DataSource = DataSource1
    RowCount = 2
    object DBEdit1: TDBEdit
      DataField = 'Name'
      DataSource = DataSource1
    end
    object DBEdit2: TDBEdit...
  end
  object Table1: TTable
    Active = True
    DatabaseName = 'DBDEMOS'
    TableName = 'COUNTRY.DB'
  end
  object DataSource1: TDataSource
    DataSet = Table1
  end
end
```

Actually, you can simply set the number of columns and rows. Then each time you resize the control, the width and height of each panel are set accordingly. What is not available is a way to align the grid automatically to the client area of the form.

Moving Control Grid Panels

To improve the last example, we might resize the grid using the `FormResize` method. We could simply write the following code (in the Multi2 example):

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This works, but it is not what I want. I'd like to increase the number of panels, not enlarge them. To accomplish this, we can define a minimum height for the panels and compute how many panels can fit in the available area each time the form is resized. For example, in Multi2, I've added one more statement to the `FormResize` method above, which now becomes

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBCtrlGrid1.RowCount :=
    (ClientHeight - Panel1.Height) div 100;
```

```
  DBCtrlGrid1.Height := ClientHeight - Panel1.Height;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

Instead of doing the same for the columns of the control grid component, I've added a TrackBar component to a panel. When the position of the trackbar changes (the range is from 2 to 10), the program sets the number of columns of the control grid and resizes it. In fact, if you simply set the number of columns, they'll have the same width as before. Here is the code of the trackbar's `OnChange` event handler:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  LabelCols.Caption := Format (
    '%d Columns', [TrackBar1.Position]);
  DBCtrlGrid1.ColCount := TrackBar1.Position;
  DBCtrlGrid1.Width := ClientWidth;
end;
```

This code and the `FormResize` method above allow you to change the configuration of the control grid at run time in a number of ways. You can see an example of a crammed version of the form in Figure 9.23.

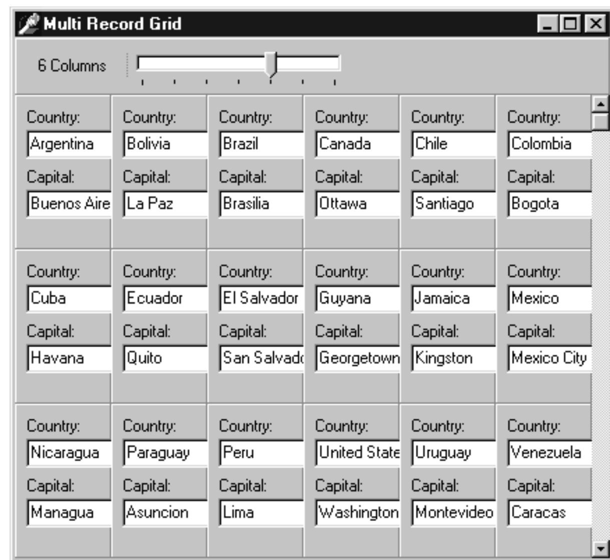


Figure 9.23: The output of the Multi2 example, with an excessive number of columns. Image from the original book.

Database Charts

Another interesting component you can use in database applications is the data-aware version of the TeeChart control built by David Berneda and available in the Professional and Enterprise versions of Delphi[249]. This component is very easy to use, particularly if your version of Delphi includes the corresponding TeeChart Wizard (found in the Business page of the File ➢ New dialog box).

To demonstrate the use of the DBChart control, I've added this component to the GridDemo example. The new application, called ChartDB, shows a DBGrid in the upper portion and a pie chart with the surface of each country at the bottom, as you can see in Figure 9.24.

The program has almost no code, as all the settings can be done using the specific component editor, which has a number of options but is quite easy to use. Here are some of the key properties of the component, taken from the form description:

```
object DBChart1: TDBChart
  Legend.Visible = False
  Align = alClient
  object Series1: TPieSeries
    Marks.ArrowLength = 8
    Marks.Visible = True
    DataSource = Table1
    XLabelsSource = 'Name'
    ExplodeBiggest = 3
    OtherSlice.Style = poBelowPercent
    OtherSlice.Text = 'Others'
    OtherSlice.Value = 2
    PieValues.ValueSource = 'Area'
  end
end
```

249 The light version of the component is still available in Delphi as an extra installer opotion.

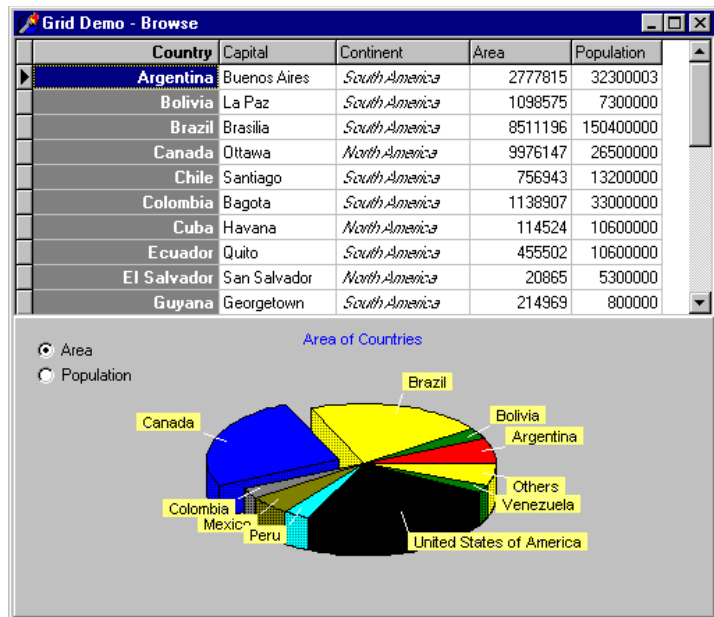


Figure 9.24: The output of the ChartDb example, which is based on the TDbChart control. Image from the original book.

note To understand these properties and the structure of the charts and the series, you can refer to the examples of the Chart component in the Chapter 22 “Graphics in Delphi”. This same chapter shows also how to dynamically export from a Web server application the graph produced by the DBChart, after converting it to a JPEG image.

What I’ve done was to show the area field as the data source for the pie chart (the `PieValues.ValueSource` property of the series), use the name field for the labels (the `XLabelsSource` property of the series), and condense all the countries with a value below 2 percent in a single section indicated as ‘Others’ (the `OtherSlide` sub-properties).

As a minor addition to the code, I’ve added two radio buttons you can use to toggle between the area and the population. The code of the two radio buttons simply sets the source of the series, after casting it to the proper series type, as in:

```
procedure TForm1.RadioPopulationClick(Sender: TObject);
begin
  DBChart1.Title.Text [0] := ‘Population of Countries’;
  (DBChart1.Series [0] as TPieSeries).
    PieValues.ValueSource := ‘Population’;
end;
```

What's Next?

In this chapter, we have seen a number of examples of database access from Delphi programs. I have covered the basic data-aware components as well as the development of database applications based on standard controls. We've explored the internal architecture of the field objects, created brand-new database tables at design time and at run time, and worked though many examples.

In particular, besides looking at the use of the data-aware controls, we've also used a couple of different manual approaches. You might wonder when the harder and lower-level approach might make sense. The short answer is to use the data-aware controls unless you need to do something unusual that conflicts with the default behavior of the data-aware controls. A typical example is the use of particular techniques for concurrency in multi-user applications, as we'll see in the next two chapters.

Is this all there is to say about Delphi database programming? Not at all. Delphi database support is very extensive and complete. The purpose of this chapter has been to give you an idea of what you can do, concentrating on the use of the Table component for database access. In the next chapter, we'll focus on the Query component, on working with multiple database tables (with joins and with master-detail and lookup structures), and on many other advanced features. We'll also cover the use of the new Data Module Designer in Delphi 5.