Delphi

Succinctly

by Marco Breveglieri

Syncfusion | Technology Resource Portal

Delphi Succinctly

By

Marco Breveglieri

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Zoran Maksimovic

Copy Editor: Morgan Weston, content producer, Syncfusion, Inc.

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marco Breveglieri is an Italian software and web developer. He started programming when he was 14 and got his first home personal computer, a beloved Commodore 16. There were not a lot of games to play with, so he started writing some simple programs using BASIC language. These included a bare calculator, a fake harmless OS, a text-based adventure game, and a music cassette database. Marco spent a lot of time in front of a white and black screen creating software routines, many of which might have seemed useless to everyone but him.

Going forward—and getting serious—years later he attended secondary school focusing on computer science, continuing that learning path and taking his first steps from BASIC to Pascal and C++. He earned his diploma and left school saying, “I am so sick, I don’t want to see another line of Pascal code!” Then he met Delphi, and he loved it. Cosmic irony.

In 1999, Marco started working for a software company using Delphi and its Object Pascal language to create HMI/SCADA industrial desktop applications for the Windows platform, expanding its knowledge to the emerging world of Web applications.

In 2003, he joined his former schoolmates Alessandro and Luca to start ABLS Team, an IT company that offers software development services for a wide range of systems, from desktop platforms to Web technologies. Their services include mobile devices, programming training courses, consultancy, and tutoring.

Today Marco continues to work on the ABLS Team, using Delphi to create desktop applications for Windows and Mac OS and mobile applications for Android and iOS. He also uses Visual Studio to build websites and applications leveraging the Microsoft Web stack based on the .NET Framework. He often takes part in technical conferences and holds training courses about programming with all these tools, especially Delphi, C# and the Web standards HTML5, JavaScript, and CSS3.

Last but not least, in his spare time Marco hosts an Italian podcast about Delphi programming called Delphi Podcast, and teaches programming to kids at his local CoderDojo.

Chapter 1 A First Glance at Delphi

Delphi was released in 1995 by Borland, the software company behind the well-known Turbo Pascal compilers.

The idea of Borland was to create an environment for rapid application development (RAD) based on components use, or better re-use, taking the well-known Turbo Pascal compiler a step forward.

At that time, Delphi was a direct competitor of Microsoft Visual Basic—at one point someone called it "VB-Killer." Thanks to the full support for object-oriented programming (OOP), multi-threading and COM (Component Object Model) support, and many other exclusive features, Delphi could achieve more complex and well-structured projects than Visual Basic 6. VB developers had to await the advent of the .NET Framework technology in order to have a comparable tool with a high-level language and modern enterprise-class business qualities.

Today, Delphi is owned and maintained by Embarcadero Technologies, a company focused on high-class enterprise data management and development tools. Delphi is sold as a standalone product or inside RAD Studio, a broader suite where Delphi is bundled with C++Builder (a different flavor of the same technologies and libraries available for Delphi but based on C++ language and compilers).

In this book we will refer to Delphi 10 Seattle, which is the latest available version at the time of writing, but everything we'll see should be fine for any older or newer versions.

The purpose of this book is to provide an overview of what you can do with Delphi and how, facing the working principles of its environment and the high potential of Object Pascal language to build native solutions for a wide number of heterogeneous devices and platforms.

A Look Inside the Box

Delphi is available either as a separate product or as part of RAD Studio, a more complete developer suite that includes a C++ language environment sharing the same libraries of Delphi.

Delphi includes:

- An Integrated Development Environment (IDE).
- A set of compilers for different target platforms.
- Core runtime and design-time libraries (RTL, VCL, FMX).
- Several third-party components and packages.
- Offline product documentation and API references.

- A lot of examples and demos with source code.

In the following chapters, we will be examining each of these parts in detail to see how you can build any kind of application with them.

Libraries and Frameworks

The development process in Delphi leverages three libraries for runtime and design-time purposes.

These libraries include full source code, so you can have a peek and get to know how some features are implemented, or sometimes put a workaround in place if there is a bug.

The Run Time Library (RTL) is a portable piece of software filled up with routines, types and classes for general purposes like string and date/time manipulation, memory and I/O management, lists and other containers and RTTI (known as Reflection in .NET and Java) just to name a few.

The Visual Component Library (VCL) leverages the RTL to provide a set of both visual controls and non-visual components aimed at creating applications and services for the Windows platform. VCL exists in the product since its inception in 1995 and after more than 20 years of improvements, it has grown up and became a solid and mature framework. It is actively maintained to embrace the most prominent features available on each version of Windows released by Microsoft.

FireMonkey is similar to the VCL, meaning that it provides components and visual controls, but it is different from the former in many aspects. First of all, it is a graphic library: the elements that make up the user interface are designed from scratch leveraging the GPU and using vectors. It is also fully cross-platform: every application created with it can be compiled to run on different operating systems, including Windows, Mac OSX, Android and iOS.

Delphi offers many other libraries to address common tasks like network and Internet communication, data access, and cloud support. To name a few, **FireDAC** is a library that lets you connect to a wide variety of data sources and databases in different formats, including NoSQL databases (like MongoDB). **Indy** is a set of classes and components that enable the creation of clients and servers for the most popular network protocols; **HTTP Client Library** provides tools and components to call and exchange data with RESTful APIs, **Cloud API** is an extensible framework to connect and consume cloud services with up-to-date components for Microsoft Azure and Amazon EC2 Services.

One Language, Many Projects

Delphi can create a wide range of projects. Here is a table that summarizes them all, but new kinds are added after each release based on the new platforms the product embraces.

Table 1: Delphi Main Project Types

Project type	Description
Console Application	A basic cross-platform application, with no GUI, runnable from the Command Prompt or a Terminal window.
VCL Forms Application	A classical Windows native desktop application based on Visual Component Library (VCL).
Multi-Device Application	A rich cross-platform client application targeting multiple devices and different form factors based on the FireMonkey library (FMX).
Dynamic-link Library (DLL)	A cross-platform native shared library, useful for exporting functions to other applications and programming languages, like old-school plain Windows DLL files.
Control Panel Application	Adds a new icon to the Windows Control Panel that launches a fully-customizable application when clicked.
Android Service	Perform background tasks on the Android platform.

Installing Delphi

Before starting our tour around all the marvelous Delphi features, you have to download and install it on your PC.

Delphi can produce applications for many platforms, however it actually is a Windows native executable, so you must have a PC equipped with Windows (Vista or later) in order to run it, or a Windows virtual machine if you use a Mac.

You can download a free trial version of Delphi from Embarcadero's website.

Once you have downloaded the setup package, launch it to begin the installation process that involves the steps described below.

1. Before anything else, you must put a check for the **License Agreement** and acceptance of **Privacy Policy**, then you can optionally join the **Embarcadero Customer Experience Program**, an initiative that provides data to help improve the product.

2. The **Setup Personalities** step lets you choose the programming languages (called "Personalities") to install. You can elect to install both Delphi and C++Builder or only one of them; if you are going to start a trial period, you can evaluate both products, otherwise you should stick with the languages included in the license bought from Embarcadero.

3. The **Setup Languages** step allows you to install different languages of the product (the default is English).

4. The **Select Features** step lets you choose what features you would like to install with the product (i.e. libraries, help, samples, add-ons). You can select a feature and read a short description below the tree.

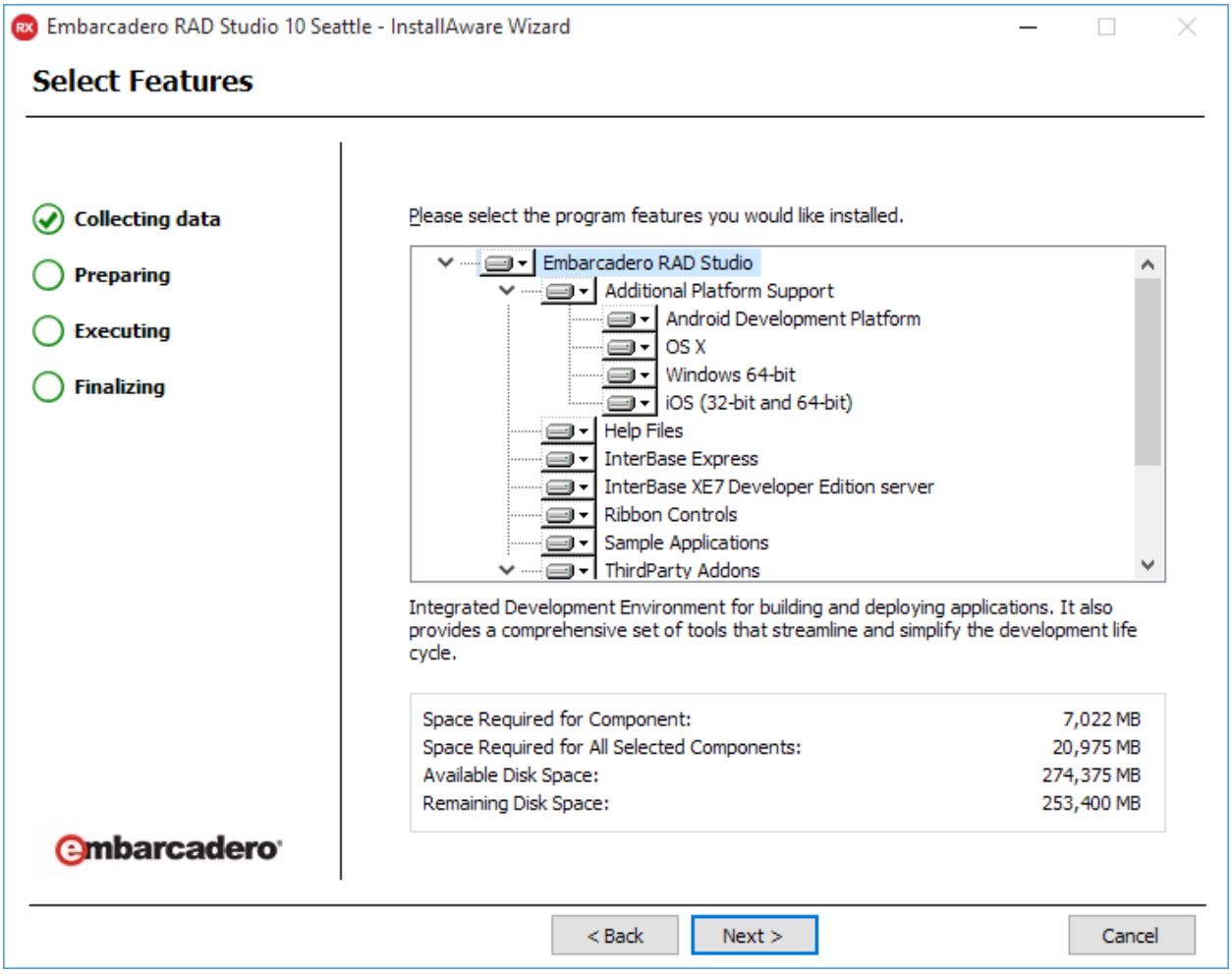

Figure 1: Feature Selection

5. The **Android SDK** step lets you install the Android NDK/SDK with the product; if you already have them installed and want to develop for the Android platform, uncheck these and configure them later, following the detailed instructions in the official documentation.

6. The **Start Menu** step is where you can edit the name of the Start Menu group and decide if you want to create it just for you or for all users of your machine.

7. The **Destination Folder** step allows you to change the destination path of program files, demos, and common resources.

8. The **Update File Associations** permits you to select which file extensions should be associated to Delphi; you can select all file types or deselect the ones that are also used by other programming tools and environments (like Visual Studio) as you wish.

9. The **Download File Location** step requires you to enter a path where the installer can save the media packages that must be downloaded according to selected features; you can also order “media kits” (DVDs) or download a full ISO distribution if you have an active Update Subscription with Embarcadero.

10. The final step is the installation itself. Wait for it to finish and then you are ready to start using Delphi.

Launching Delphi

Once the installation procedure is complete, you can launch Delphi by clicking the corresponding link in the Windows Start Menu.

If you see a registration window appearing, you can choose to create a new Embarcadero Developer Network (EDN) account or logging in if you already have one. You should also have received further instructions by e-mail when you have downloaded the installer.

A splash screen will appear to show information about the loading process of both core components and add-ons where available.

When everything has been loaded, the full-screen Delphi IDE main window will open.

At the center of the main window you can find the **Welcome page**, a useful resource to deepen your knowledge with tutorials, examples and how-to procedures you can look at once finished reading this book, surrounded by many dockable panels, and each of them has its own scope.

Summary

In this chapter we introduced the libraries available in Delphi: Visual Component Library (VCL) for Windows development, and FireMonkey (FMX) for multi-device, cross-platform applications. We will dig into these libraries in the rest of the book to see how you can use them successfully.

We also had a look at the installation process. We have just covered some broad information, but remember, you can always check the Installation Notes from the product’s official documentation for more details.

Chapter 2 Your First Application

I hope you are now looking forward to creating your first project with Delphi. We could skip a "Hello World" demo, but why not follow tradition?

Creating a New Project

To start, we'll create a Windows desktop application from scratch. To do that, move the mouse pointer to the main menu, point to the **[File|New]** item and select **VCL Forms Application**.

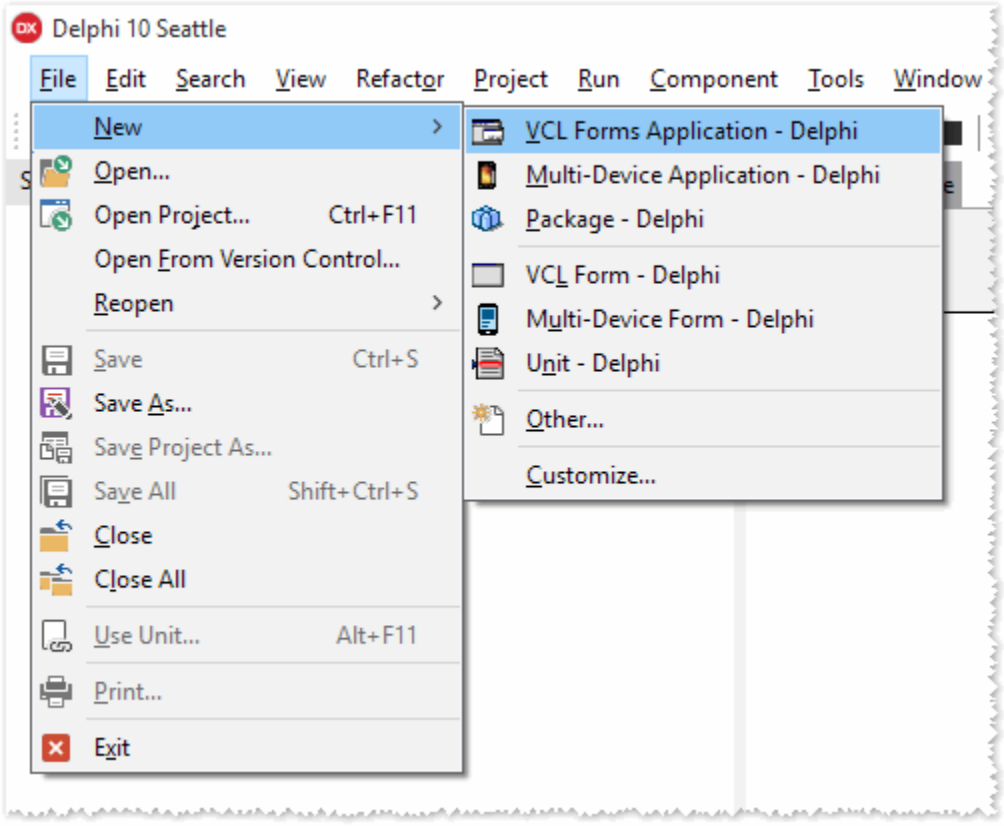

Figure 2: Delphi Main Menu

Delphi will create the new project and prepare the environment to start working.

When you create a new project, Delphi automatically inserts a new empty form (known as the Main Form) into the project and opens so you can begin designing the UI of your application.

You are free to delete the project or add new forms to it later. For now, we will try some basic operations, and explore the details in the next chapters.

Adding Controls

From the **Tool Palette** window, expand the “Standard” category by clicking the plus (+) sign, then click the **TEdit** control and drag it to your form, drop it on the empty window area. Next, repeat the procedure selecting the **TButton** control.

If you want to change the text on the button you just added, left-click the control you placed inside the Form and point to the **Object Inspector** window. Scroll through the grid to find the **Caption** property and click on it to enter new text, replacing the default value (Button1) with a new label. Press Enter or click elsewhere outside the Object Inspector panel to complete the change.

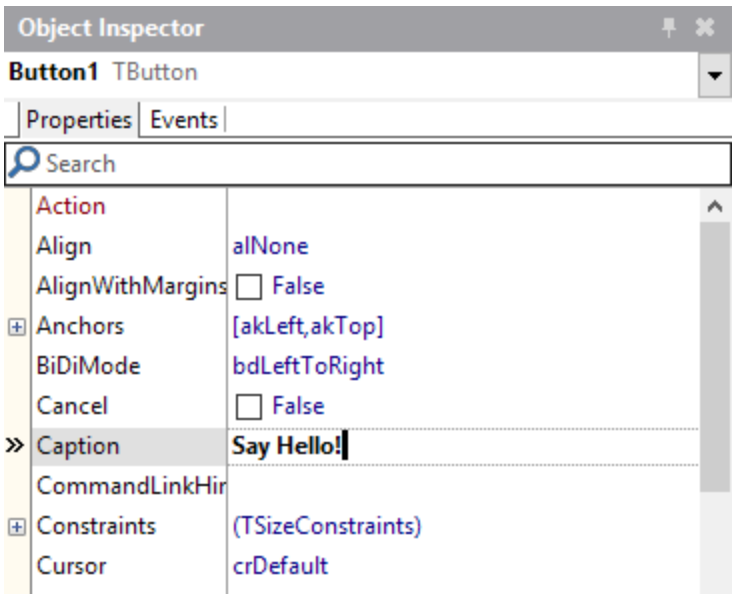

Figure 3: Button Properties

You can do the same to clear the contents of the textbox **TEdit**: select the control and change the property **Text** to an empty value.

If you want to move a control, you can click and move it around by holding the left mouse button down, or use the blue dots to change its size. Our final Main Form should look like this:

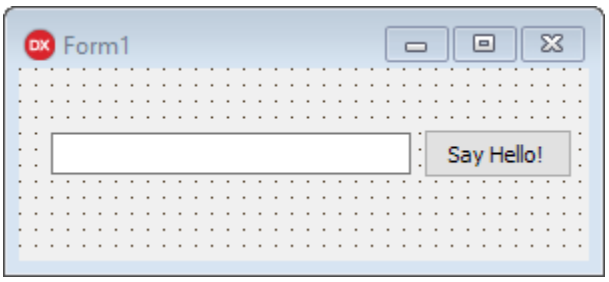

Figure 4: “Hello World” Form

Responding to Events

If you double-click the **TButton** control, Delphi will switch to the *Code Editor* and create a routine that handles the event raised when the user clicks the button. Delphi takes care of writing lot of the boilerplate code, leaving the specific implementation up to you.

Suppose we want to say “hello” when the user inserts his (or her) name and clicks the button. You could write something like the following:

Code Listing 1: Hello World

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Hello, ' + Edit1.Text + '!');
end;
```

Don’t pay too much attention to the syntax—we’ll spend a full chapter on Object Pascal.

Run Your Application

To run the application, select **[Run|Run Without Debugging]** (or press CTRL+F9) and Delphi will start building your project.

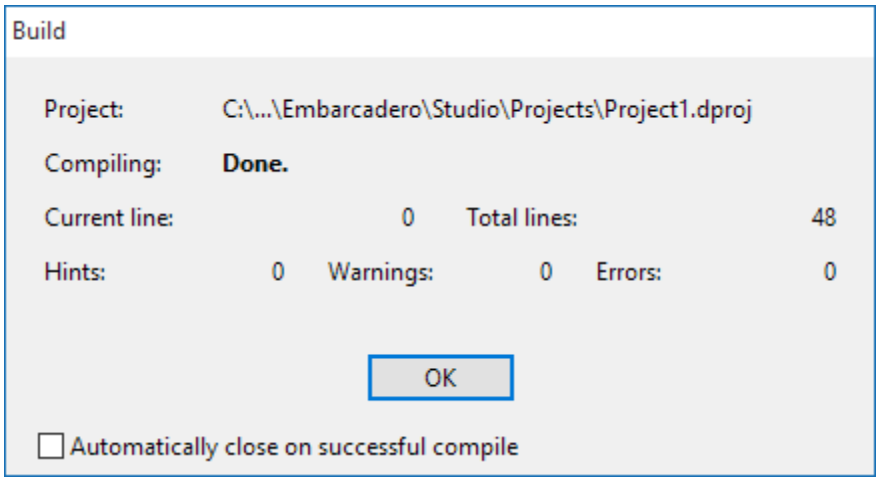

Figure 5: Build Progress Window

If no errors are detected during the building process, Delphi launches the application executable file (which has the “.exe” extension on Windows).

Note: Delphi is a native compiled language, so it always creates an executable file when you need to run your application and see if it works as you expect. You can’t avoid that. You can only choose to run the program without debugging so Delphi does not attach to the process once the build is completed and both processes continue to live separately.

If you followed all the steps correctly, the Main Form should appear and you should see the following message when you enter your name and click the button.

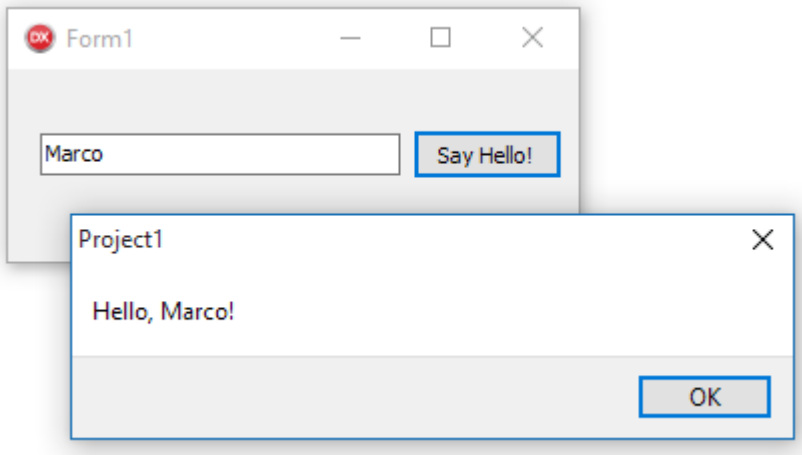

Figure 6: "Hello World" Running

The message box invoked by calling the **`ShowMessage`** function is a modal dialog. This means the execution will not proceed until you close the window by clicking OK or the close button at the upper right corner.

Note: The application keeps running as long as the Main Form is visible on the screen. You can minimize it or move it around on your desktop, and when you close it, the application terminates.

Congratulations, you have just created your first Delphi program!

Deploy Your Application

When you run or build the project, Delphi produces a native application.

The Visual Component Library (VCL) we have used to create our "Hello World" application targets the Windows platform, and the compiler for this platform actually writes an executable (.exe) program file.

The term "native" means that your program contains machine code that can be executed by the target CPU and OS without the need for runtimes and virtual machines.

An installer or a packaging tool is not required to deploy your program: you can simply copy the executable file to the destination machine and launch it.

Tip: Sometimes your application may need to use external files, resources or databases; in these cases, I recommend using an installer tool to ensure that all required files are copied to the destination machine. This lets the user customize the process, like the program target directory, the choice of optional components,

the help manual, and samples. If you want a simple, easy-to-use tool, look at Inno Setup (which, by the way, was built with Delphi!)

Summary

Here we have only scratched the surface of what you can achieve with Delphi, and I think you can already get a clear idea of how much the Rapid Application Development (RAD) approach speeds up the development process.

Using Delphi, you can focus exclusively on the key parts of your application, such as the user interface and the business logic, without concentrating too much on the internal mechanisms.

I know this example may seem overly simple, but the next chapters will illustrate how to use Delphi to make applications that are far more complex. Before that, we should familiarize ourselves with the tools available in the IDE.

Chapter 3 Exploring the IDE

Before we delve into each kind of project you can build with Delphi, you should get in touch with the fundamental tools available in the IDE and see how they work on a concrete circumstance.

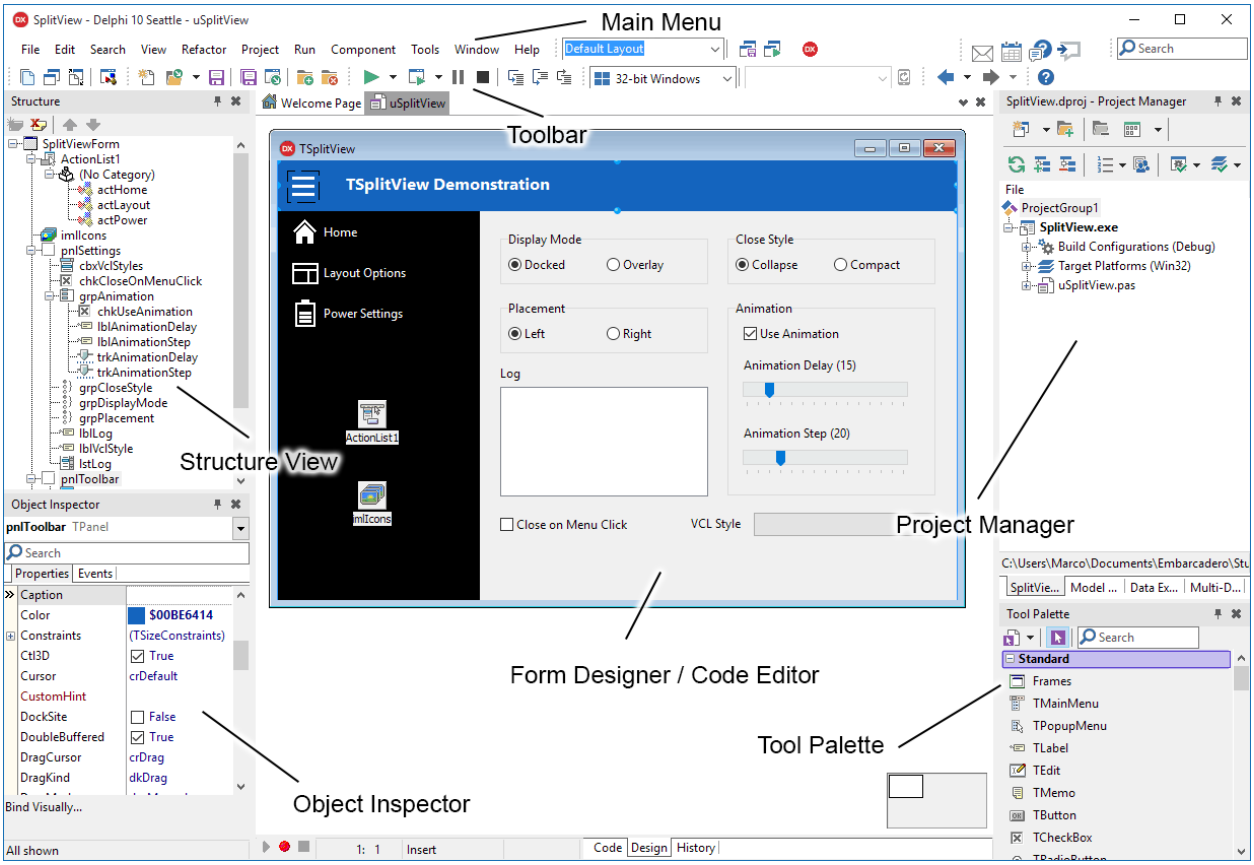

Figure 7: The Delphi IDE

The Tool Palette

The Tool Palette hosts all of the available components and visual controls.

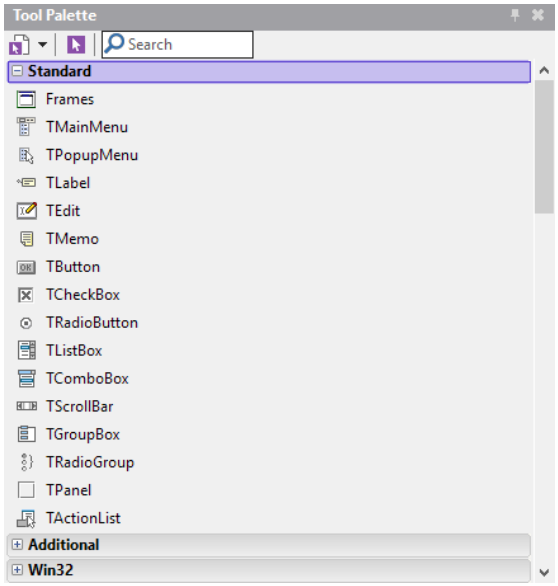

Figure 8: Tool Palette

You can drag components from the Tool Palette to the Form Designer to add functionality to your application, or drag and drop visual controls to build a user interface. Some of them are installed "out of the box" with Delphi, but you can also install new components from third parties or build and install your own components.

All the items inside the Tool Palette are categorized according to their usage context or the library they belong to. Just click the name of the category you want to see the list of components and controls it contains.

Tip: The Tool Palette supports "incremental search." When the panel is in focus and the cursor is flashing inside the Search box, start typing the name of the desired component. The palette displays only the items that match the text you enter.

The Form Designer

The Form Designer occupies the main area of the IDE window, and it is where most of the action of designing an application takes place. Here is where you drop components and visual controls like you just did in the "Hello World" sample.

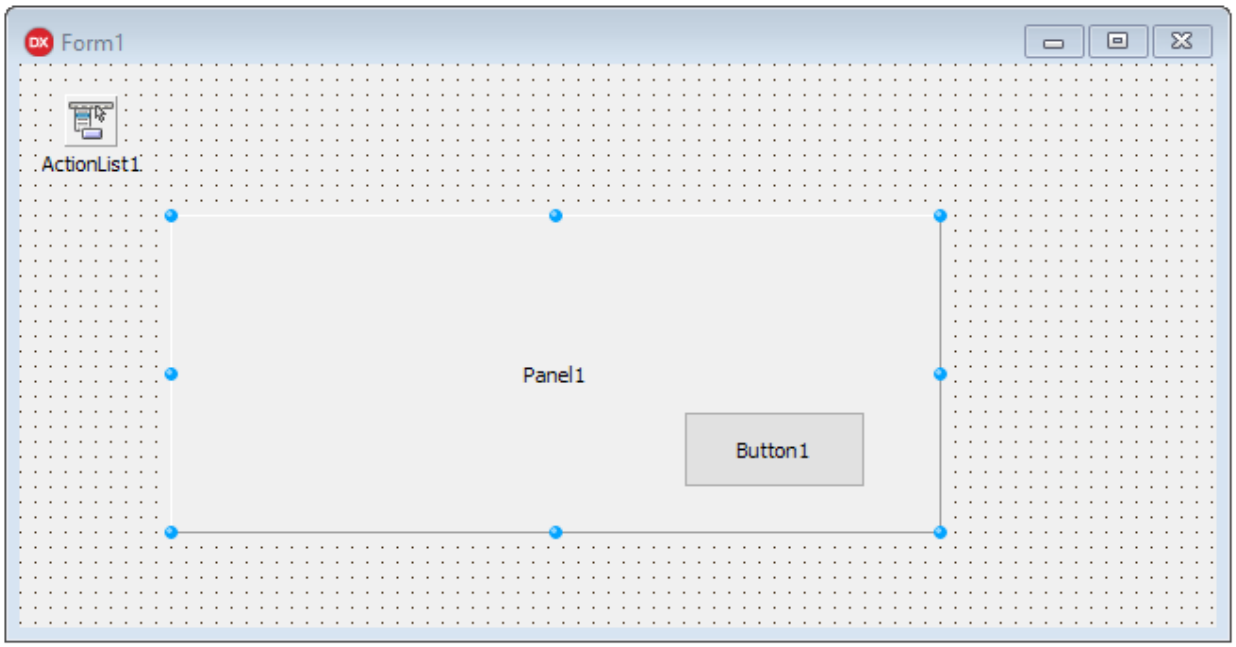

Figure 9: Form Designer

Each control has a default size and position when dropped. You can change the latter by clicking the control dragging it around, or resize the control with the handles that appear when you click the blue dots.

Some controls can act as parents for other controls. Try it yourself by dragging a **TButton** control from the Tool Palette to the panel you previously dropped. If you move the panel around, the button will follow it, retaining its relative position.

Now try adding a non-visual component: select and drag the **TActionList** component to your Form. Any component will be displayed as an icon on the Form at design time but you will never see it at runtime since it provides only business logic and behavior, hence it does not have a visual representation.

Tip: Sometimes you will end up with a lot of components on your forms, and this can make it difficult to work with visual controls as components keep getting in your way. When this happens, you can select [Edit|Hide Non-Visual Components] from the main menu, or press CTRL+H to hide them from your view. When you are done, select that menu item or press the shortcut again to restore them.

The Code Editor

If you click on the Form Designer and select **[View|Toggle Form/Unit]** from the main menu (or press **F12**), Delphi switches immediately to the Code Editor window.

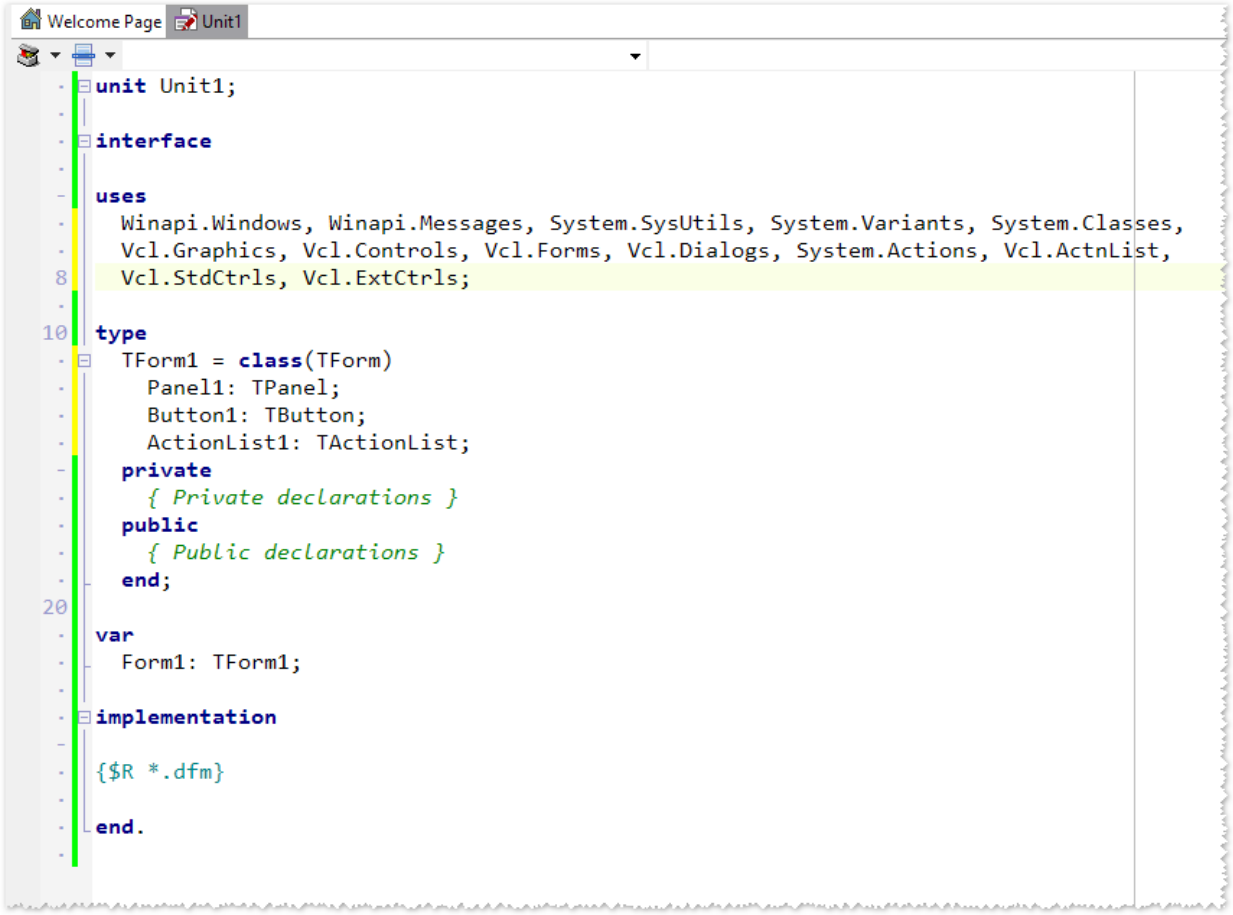

Figure 10: Code Editor

Here is where you write the source code of your application. Each form is usually linked to a source code file that contains a type declaration of the form itself. Delphi keeps that declaration in sync with your changes at design time, writing the code for you, while you can add custom code manually as a response to events, modifying the behavior of your application.

The Code Editor is also used for units of Pascal code that are not bound to a form, and, more generally, to edit any kind of text file.

If you press the **F12** key or select **[View|Toggle Form/Unit]**, Delphi switches again to the Form Designer.

Tip: The Tool Palette supports "incremental search." This means that when the panel is in focus and cursor is flashing inside the Search box, the palette displays only those items that match the text you enter.

Structure View

The Structure View displays a tree diagram that changes depending on the active document.

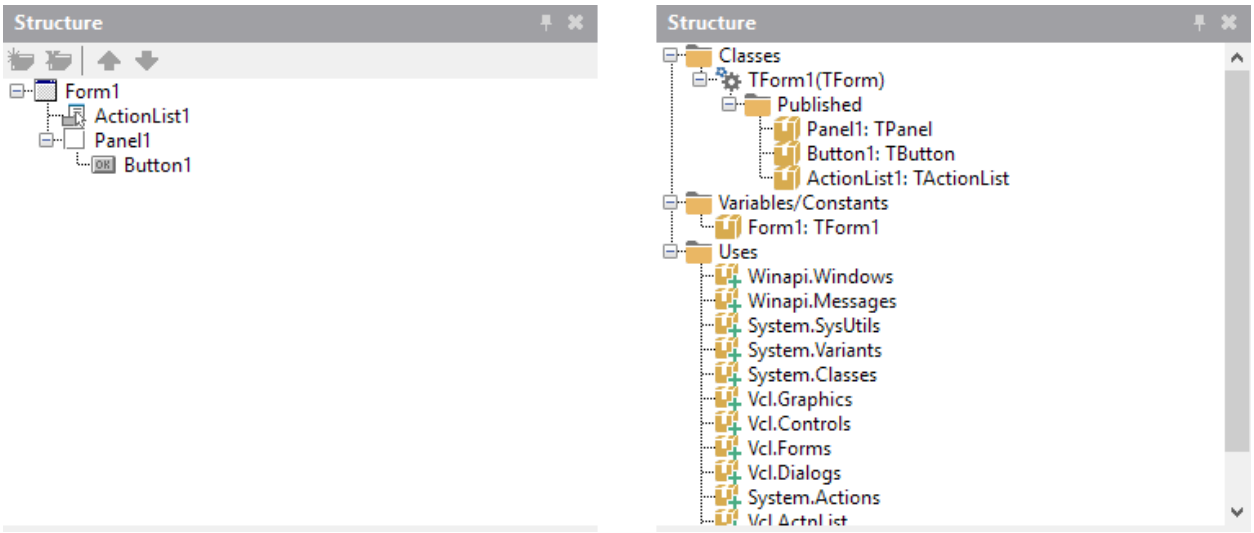

Figure 11: Structure View

When you work with the Form Designer, the Structure View displays all the components and visual controls placed on your form, and you can select them from this view.

If you switch to the Code Editor, you'll see the hierarchy of classes, types, variables, constants and so on, and you can double-click a node in the tree to quickly move to where that item is defined in the code file.

The Object Inspector

The Object Inspector is a panel where you can edit all the properties of selected forms, components, and visual controls.

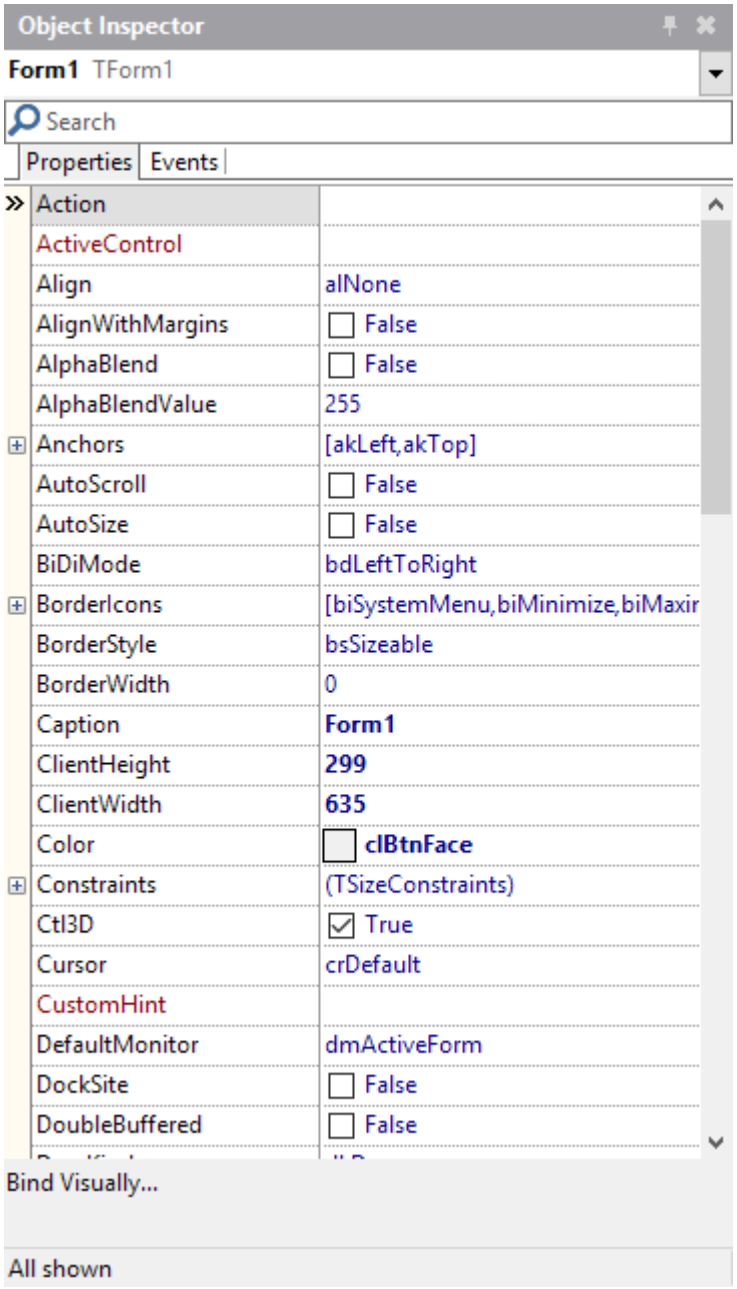

Figure 12: Object Inspector

Once you have selected a component or a control, either from the Form Designer or by using the Structure View, the Object Inspector lists all the properties that are available at design time.

For example, you can change the caption of a **TButton** control by selecting it, clicking the **Caption** property, and writing the new text you want to display inside the button.

Properties can affect the behavior of components and visual controls, and alter the visual aspect of the latter.

The Object Inspector has an Instance List on the top. You can select the component for which you want to change property values from this list, as you have already done using the Form Designer or Structure View.

You can even filter the list of properties using the **Search** box and entering the name of a property (or part of it).

In the bottom section, you can find the **Quick Action** panel. When you select a component that has quick actions associated with it, you can click a hyperlink that triggers a wizard or an automated procedure that performs a complex setup in seconds.

The Object Inspector has the following tabs:

- **Properties**: Use this to set the property values that affect the visual aspect or behavior of controls and components.

- **Events**: Use this to assign the reference to a handler method; double-click a row to create a new empty handler.

The Project Manager

The Project Manager shows the files that are part of the project, or a group of projects, displaying them in a hierarchical form, and includes some “special nodes,” which refers to build configurations, target platforms, and other resources.

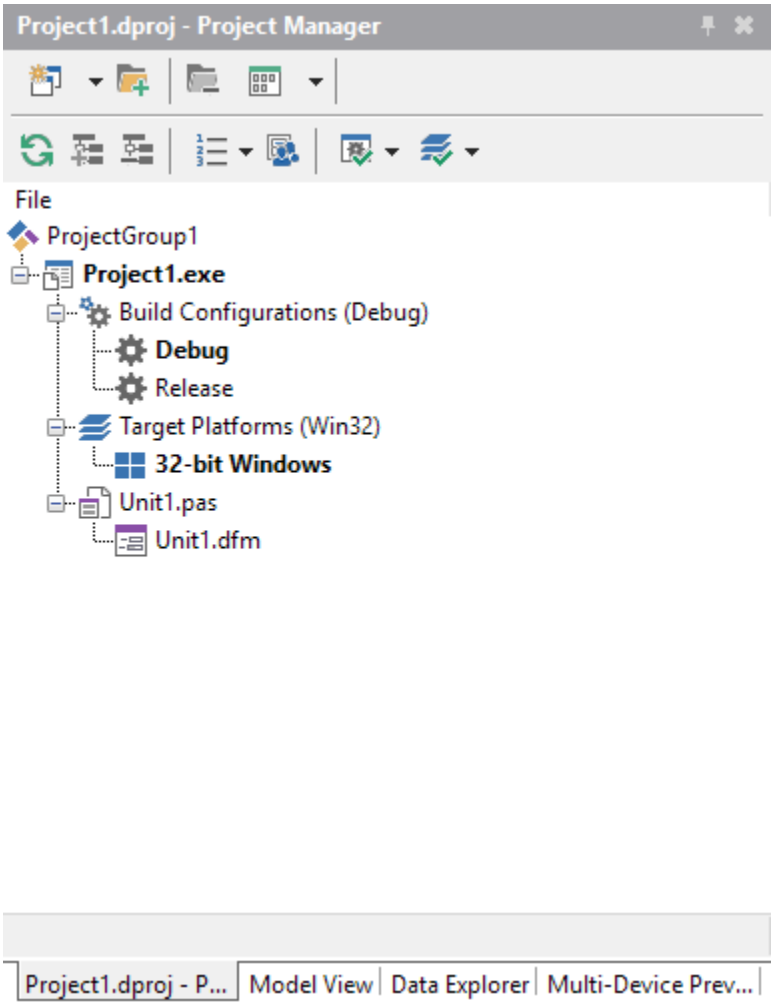

Figure 13: Project Manager

This panel has a toolbar that lets you add new files to your project, change the arrangement of the items listed in the tree, change the target platform for your application, or quickly activate a different build configuration—just point to the toolbar to see what each button does.

Note: If you look carefully at the project tree, you will notice that the main form has two files linked together. One file contains the definition of the form, i.e., the values you set in the Object Inspector for the form and the components inside of it. The other file contains the Pascal code that guides its behavior (the code you can see and edit using the Code Editor).

Other Panels

There are many other panels available inside the IDE. Some of them are hidden unless there is relevant content to display. Others share the same tab group of the panels we have already discussed.

Data Explorer

The Data Explorer shows ready-to-use database connections and lets you access data live at design time.

You can add new connections or edit and delete existing ones.

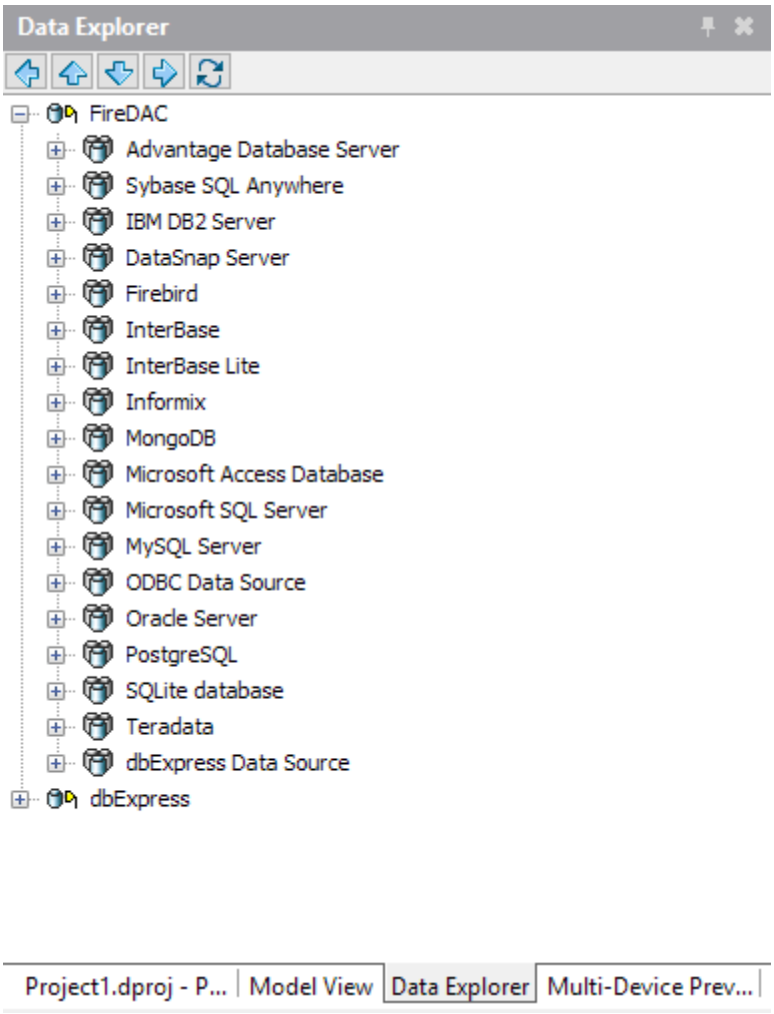

Figure 14: Data Explorer

We will dive in to the data access libraries supported by Delphi in the following chapters.

Model View

The Model View panel allows you to add support for modeling to the project. This creates a logical representation of its contents, where the project is a package with a root namespace and each type is an element node.

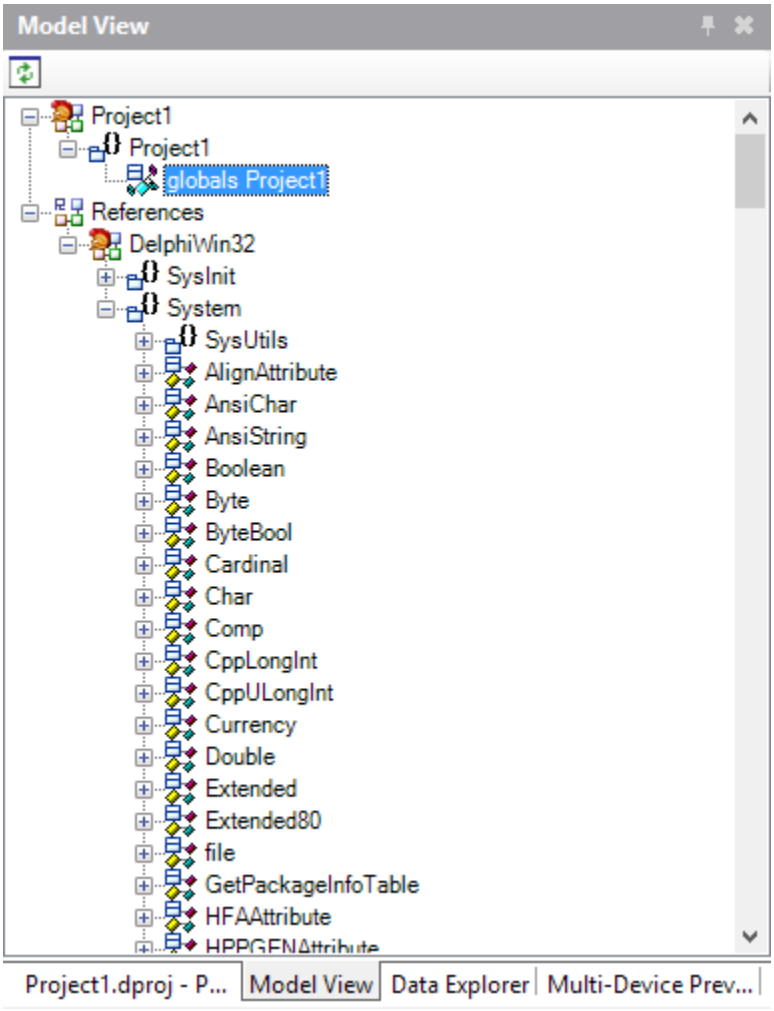

Figure 15: Model View

You can use this view to add UML diagrams to your project auto-generating the code, or vice versa, keeping them coordinated.

Class Explorer

The Class Explorer panel shows a hierarchical view of the classes in your project similar to the Structure View, but with features that are more powerful. For example, you can apply several refactoring options, add new members to classes (i.e. methods, fields, properties), and search for usages for a symbol.

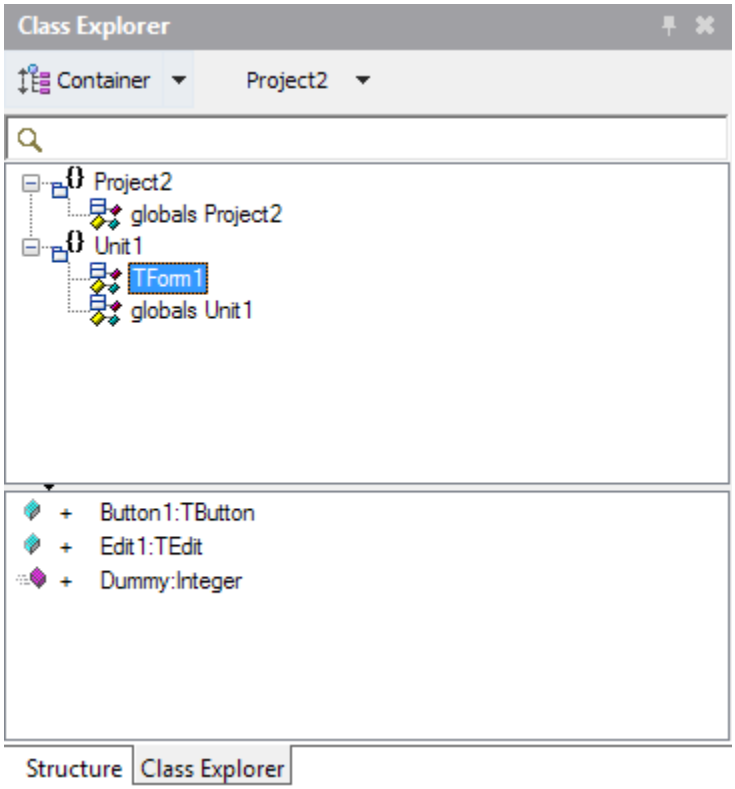

Figure 16: Class Explorer

To-Do List

This panel lets you take care of the tasks needed to complete your work.

Here you can see a list of tasks defined using **`// TODO`** comments in code and manually add new global tasks to your project, saved in a separate ".todo" file.

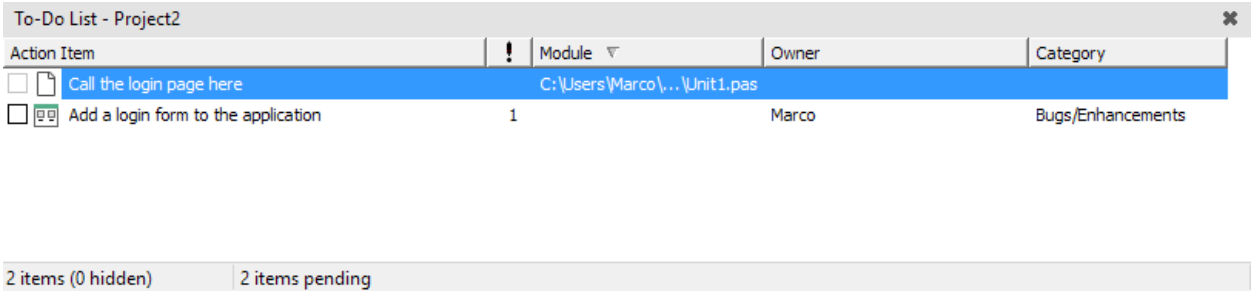

Figure 17: To-Do List

When you are finished with your task, just add a check to mark it complete.

Multi-Device Preview

The Multi-Device Preview is one of the most recent exclusive additions to the product, and is available only for Multi-Device Applications.

You can think of it as a virtual desk where all the most popular devices (both desktop and mobile) are laid out to see how your app will look on them. This includes the different form factors for each device, so you can fine tune your UI and ensure it is working effectively on your selected target device.

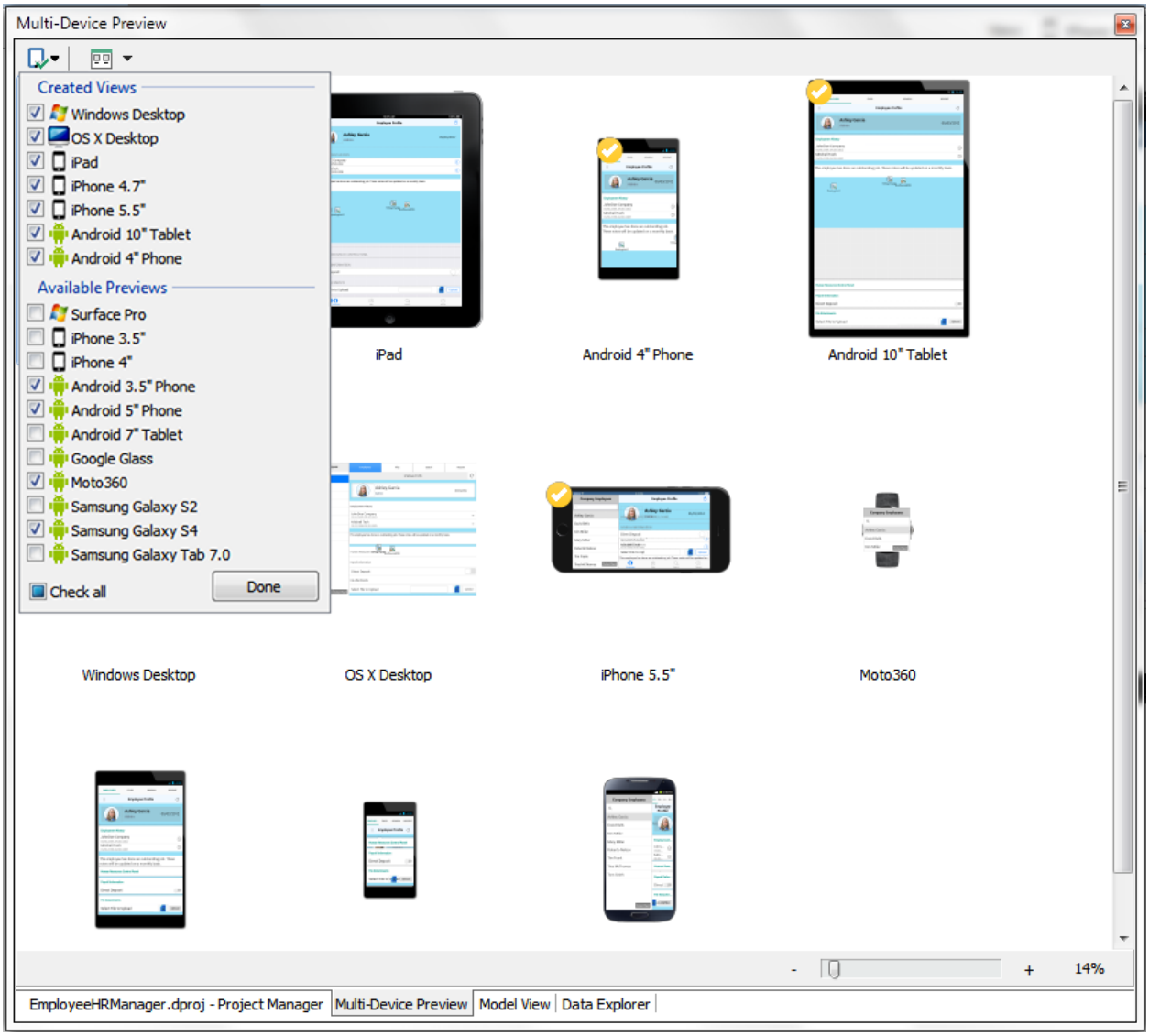

Figure 18: Multi-Device Preview

Messages

The Messages panel is hidden by default, but becomes visible when there are errors after you build a project, showing warnings or hints that need your attention.

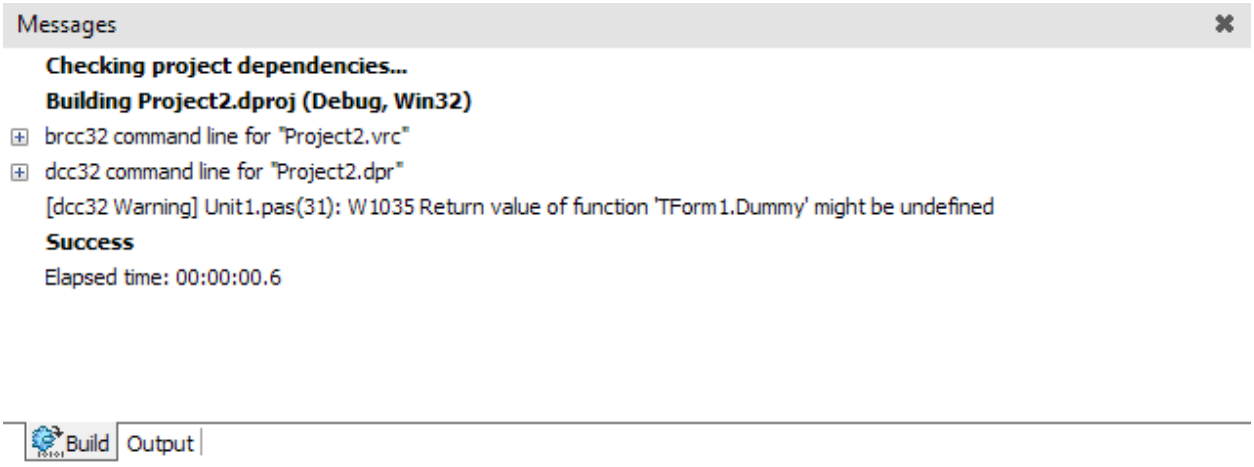

Figure 19: Messages

Refactorings

This panel is usually hidden by default, and Delphi expands it when you apply refactorings to your code, displaying a preview of all the expected changes. You can then choose to proceed with applying them or cancel the operation.

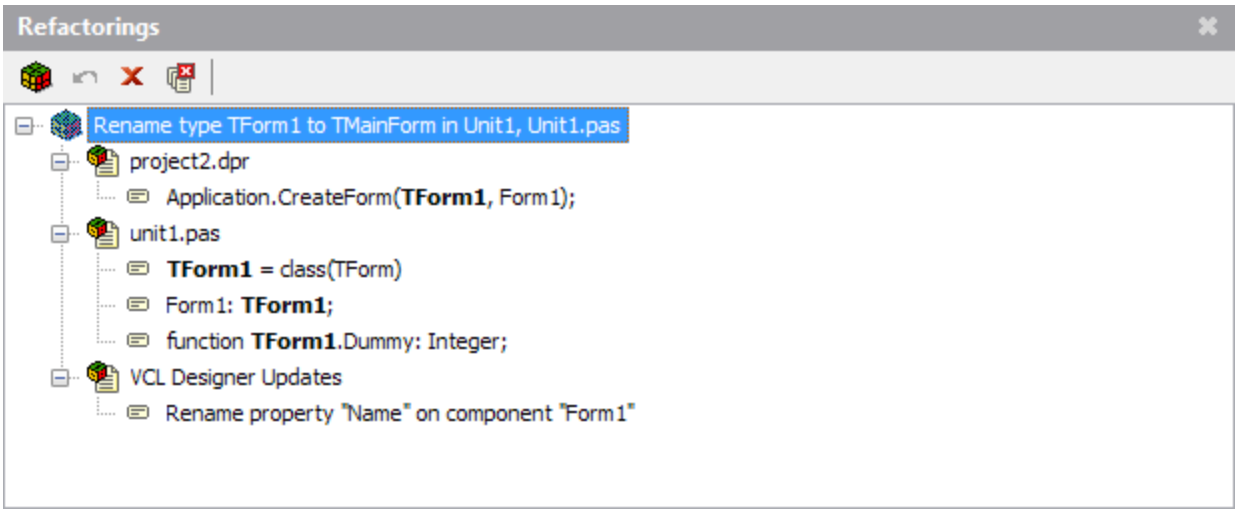

Figure 20: Refactoring Preview

Summary

In this chapter, we have given a purpose to the panels available in the IDE.

You can use the Tool Palette to select components and controls and drag them to your visual form in the Form Designer. You can change the design time properties through the Object Inspector, switching to the Code Editor to write the business logic.

You can find your project files inside the Project Manager window, then open them and navigate their contents using the hierarchical tree offered by the Structure View.

Most of the panels have a toolbar, and you can right-click them to display a menu. Make the toolbars visible using the [View] menu item and feel free to experiment with any command and option available, because you will use them all the time!

Chapter 4 The Object Pascal Language

Let's take a journey into the wonderful world of the Object Pascal language.

Pascal is a strongly-typed language, and it tends to be simple, elegant, and clean. It is often used as a learning language thanks to these features. Nevertheless, it is also extremely flexible, since it supports different coding styles and paradigms like procedural and object-oriented programming.

The Delphi incarnation of Object Pascal language is powerful and stuffed with many features, like anonymous methods, generics, and operator overloading. If you have used Pascal at school, nowadays the language—especially in Delphi—is a lot more than that.

Note: Always keep in mind that Pascal is a case-insensitive language, like Visual Basic. However, I recommend that you maintain consistency in the names used as identifiers. Delphi will help in this task through the Code Insight technology and by proposing a list of identifiers valid in the given context. This helps you quickly complete the instructions in code.

Fundamental Elements

Comments

Even if every compiler in the world always ignores them or strips them away, comments are essential to document the most delicate parts of your code, and the first element explained in any programming book.

Delphi supports single-line comments that begin with `//`, or classic Pascal multi-line comments using curly braces `{…}`, or the `(*…*)` character sequence:

Code Listing 2: Comments

```
// This is a single line comment

(*
  This is a multi-line comment
  that spans more than one row.
*)

{
  Hey, this is a multiline comment, too.
  Choose your favorite style
}
```

Compiler Directives

Sometimes you will see some comments with a strange form like this:

Code Listing 3: Compiler Directives

```
{$APPTYPE CONSOLE}
{$R MyResourceFile.res}
```

These are not comments, but directives. They are options to influence the compilation process. For example, **`{$R MyResourceFile.res}`** tells the compiler to embed the specified resource file and its contents in the executable.

Program structure

Each type of project in Delphi has a main program file with the basic organization shown in Code Listing 4.

Code Listing 4: Program Structure

```
program ProjectName;
begin
  // TODO: Put your main logic here…
end.
```

The keyword **`program`** denotes the presence of a runnable application, but you can also see something like **`library`** (for dynamic-link libraries) or **`package`** (for Delphi packages that are bundles of classes and components with runtime and design time support).

Note: Regardless of the type of project, you can always select [Project|View Source] from the main menu to open up the main program source file.

The **`begin…end`** code block contains the instructions that are executed at program startup and are actually the entry point of your application.

Tip: I do not recommend putting a lot of code inside the main program file. Your logic should be implemented using the best, most popular programming practices. This means creating separate modules, where each contains the data types and classes that take care of a specific aspect of your model (e.g. data storage, logging, security, etc.)

So, how do you create code modules in Delphi?

Units as Modules

You must place every line of Pascal code inside a **unit**. Aside from the main program file, a unit is the minimal structure necessary to accommodate any line of code, and each unit is saved in a separate Pascal file (meaning a file with a ".pas" extension).

Code Listing 5: Basic Unit Structure

```
unit MyModule;

interface

implementation

end.
```

The **unit** keyword is followed by an identifier that acts as a sort of namespace for all the elements the unit contains.

Note: The name of the unit must always coincide with the name of the file. For example, if the unit is named "MyModule," it must be saved in a file called "MyModule.pas." You can always use a dotted notation if you prefer, like "MyCompany.MyModule," to create a sort of namespace.

Though it seems strange, the keyword **end** actually ends the unit.

Inside there are two main sections, **interface** and **implementation**. Each element in the **interface** section is visible outside the unit so other modules (read "units") can access it, while everything in the **implementation** represents a "black box" not available outside the unit.

Functions and Procedures

Suppose you want to create a unit that contains some shared utility (global) routines using a procedural coding style. You can declare and implement your routines inside the implementation section:

Code Listing 6: Global Functions

```
unit MyUtilities;

interface

// Function declaration
// (makes it visible to other units)
function DoubleString(const AValue: string): string;

implementation

// Function implementation
function DoubleString(const AValue: string): string;
begin
  Result := AValue + AValue;
end;
```

```
end.
```

This example shows a static linked routine that takes a **string** as a parameter, then concatenates the string to itself and returns the value as a result.

Note: If you are curious about the meaning of the const keyword before the AValue parameter declaration, it prevents the function to change the value of that variable. This constraint allows the compiler to apply an optimization by trusting the function not to change the value and so passing safely a pointer to the string value, without creating a copy of the whole string.

You use the **function** keyword if your routine has a return value, otherwise you must use **procedure**. When you call a function that returns a value, you have the freedom of choosing whether to use the returned value or not.

In the example code, the function **DoubleString** is visible and can be used in other units since its declaration appears also in the **interface** section. It is, to all effects, a global function. To make the function private to the unit, simply remove it from the **interface** section.

Importing Units

To use an element that is already declared in another unit, you must add a **uses** clause to your unit, followed by a comma-separated lists of the units you want to import.

Note: Always remember that you can import only what is declared in the interface section of a unit; everything in the implementation section will remain hidden.

Look at Code Listing 7 to see how this works.

Code Listing 7: Uses Clause Sample

```
unit MyBusinessLogic;

interface

uses
  MyUtilities;

implementation

// Routine implementation
procedure DoSomething();
var
  SomeText: string;
begin
  SomeText := DoubleString('Double this string');
  WriteLn(SomeText);
end;
```

```
end.
```

As you can see, the **DoSomething()** procedure can call **DoubleString()**because the unit that contains the function is listed above in the **uses** clause; if omitted, you would get one of the most frequent errors at build time: “Undeclared identifier.”

Variable Declarations

Every time you declare a variable in Delphi, you must assign it a type, since Pascal is a strong-typed language.

We have already used a **string** variable in some previous example to save a sequence of characters.

If variables are declared inside a procedure or a function, they are local, and therefore not visible outside that routine. If you put a variable declaration in the **implementation** section of a unit, you make them global to the whole unit. If you place a declaration in the **interface** section, every other unit that imports it can also access the variable.

Local variables must be declared before the **begin…end** block that marks the body part of procedures and functions, so you cannot declare variables “inline” (like you do in C#, for example).

Note: Every identifier must be valid to be accepted by the Delphi compiler without complaints. To do that, never start a name with a number or use a reserved word for your units, routines, types and variables.

Variable Assignments

Before using any variable, you must assign a value to it using the **:=** operator.

Here are some samples of variable declarations and assignments:

Code Listing 8: Variable Declarations and Assignments

```
// Variable declarations
var
  Number: Integer;
  Flag: Boolean;
  Letter: Char;
  Text: string;
  Instance: TObject;
begin
```

```
  Number := 123;
  Flag := True;
  Letter := 'D';
  Text := 'abc';
  Instance := TObject.Create();
  // TODO: Put your implementation here...
end;

end.
```

Each variable assignment is required to be type-safe: you must assign a variable only values that belongs to the same type, or a descendant type (if it is an object), or a value that cannot lead to data loss. If the code does not meet these conditions, Delphi will issue a compiler error when you build your code.

Basic Data Types

Delphi provides a set of primitive types. The name comes from the fact that these are built-in types the compiler knows, and they can be used to declare variables or create more complex types, like records and classes.

Whenever you want to introduce a new type, use the **`type`** keyword. You will see some usage of it in the rest of this chapter.

Integer Types

Delphi uses Integer types to store non-floating point values. I have summed up them in Table 2.

Table 2: Integer Data Types

Type Identifier	Range	Size
ShortInt	-128..127	Signed 8-bit
SmallInt	-32768..32767	Signed 16-bit
Integer	-2147483648..2147483647	Signed 32-bit
Int64	-2^63..2^63-1	Signed 64-bit
Byte	0..255	Unsigned 8-bit
Word	0..65535	Unsigned 16-bit
Cardinal	0..4294967295	Unsigned 32-bit
UInt64	0..2^64-1	Unsigned 64-bit

Boolean Types

Delphi provides a **`Boolean`** type to express Boolean values and perform standard Boolean logical operations.

Note: There are also other Boolean types available (ByteBool, WordBool, LongBool) but they exist for backward compatibility.

Code Listing 9: Boolean Type

```
// Variable declaration
var
  Proceed: Boolean;

// Usage
Proceed := True;
Proceed := False;

if Proceed then
```

Enumerated Types

Enumerated types are an ordinal integer-based type that associates a set of identifiers to integer values. Here is a sample demonstration:

Code Listing 10: Enumerated Type

```
// Type declaration
type
  TSeason = (Spring, Summer, Fall, Winter);

// Variable declaration
var
  Season: TSeason;

// Usage
Season := TSeason.Summer;
```

Characters and Strings

Delphi provides a **`Char`** type that is an alias for the **`WideChar`** type and represents a 16-bit Unicode character. The **`String`** type is an alias for a **`UnicodeString`** and it contains zero or more Unicode characters.

Code Listing 11: Character and String Types

```
// Variable declaration
var
  SomeChar: Char;
  SomeText: String;

// Usage
SomeChar := 'D';
SomeText := 'Delphi rocks!';
```

If you have to work with ANSI code pages, you can use the **AnsiChar** and **AnsiString** types; this happens often if you have to import functions from third-party DLLs or other legacy systems.

Delphi lets you assign an **AnsiString** to a **UnicodeString** doing an implicit conversion. The opposite could lead to data loss, since Delphi could be unable to convert a Unicode character if the current ANSI code page misses it.

The rest of the world is (or should be) Unicode compliant by now, so if you are not dealing with these scenarios and don't have to provide some backward compatibility, stick with the common **Char** and **String** types.

Note: Be warned, Delphi string indexing is actually different on desktop and mobile compilers: it is 1-based on the desktop and 0-based on mobile. If you use RTL string functions to access characters in a string, pay attention to this aspect or, even better, use the TStringHelper static class and its methods to perform string manipulations in a safe mode.

Subrange Types

You can also define your own range-limited ordinal types specifying the lower and upper bounds, as shown in Code Listing 12.

Code Listing 12: Subrange Types

```
// Type declaration
type
  TMonthDay = 1..31;
  TYearMonth= 1..12;
  THexLetter= 'A'..'F';

// Variable declaration
var
  MonthDay: TMonthDay;
  YearMonth: TYearMonth;
  HexLetter: THexLetter;

// Usage
```

```
MonthDay := 31;
YearMonth := 12;
HexLetter := 'B';
```

Real Types

Real types are used to store approximated floating-point values. Here is a complete list of the types that belong to this family:

Table 3: Real Data Types

Type Identifier	Range	Significant digits	Size (bytes)
Single	1.5e-45 .. 3.4e+38	7-8	4
Double	5.0e-324 .. 1.7e+308	15-16	8
Real	5.0e-324 .. 1.7e+308	15-16	8
Extended	3.4e-4932..1.1e+4932 (on 32-bit platforms) 5.0e-324..1.7e+308 (on 64-bit platforms)	10-20 (32-bit) 15-16 (64-bit)	10 (32-bit) 8 (64-bit)
Comp	$-2^{63}+1$ .. $2^{63}-1$	10-20	8
Currency	-922337203685477.5808 ..922337203685477.5807	10-20	8

Note: The Currency data type is stored as a 64-bit integer to minimize rounding errors when used for calculations that involve money values.

Array Types

You can declare arrays that contain elements of any kind, whether static, with a fixed length and a specified range, or dynamic, with a variable length that you can adjust at runtime by calling the **`SetLength`** function.

Code Listing 13: Arrays

```
var
  // Fixed length array.
  IntArray: array[1..10] of Integer;
```

```
  // Dynamic array.
  DynArray: array of Char;

// Basic usage
IntArray[1] := 20;
IntArray[10] := 200;

// Dynamic array usage
SetLength(DynArray, 50); // allocate memory for 50 elements
DynArray[0] := 'D';
```

Set Types

Object Pascal has a specific syntax support for **`set`** types, which is a collection of elements all belonging to the same type. The number of elements you can have in a **`set`** variable depends on the base type: if you create a set of **`Byte`** elements, you can store up to 255 byte-value elements in it.

The order of the values is not significant, and elements can only be added to the same set once. Think of it as the software representation of a Venn diagram.

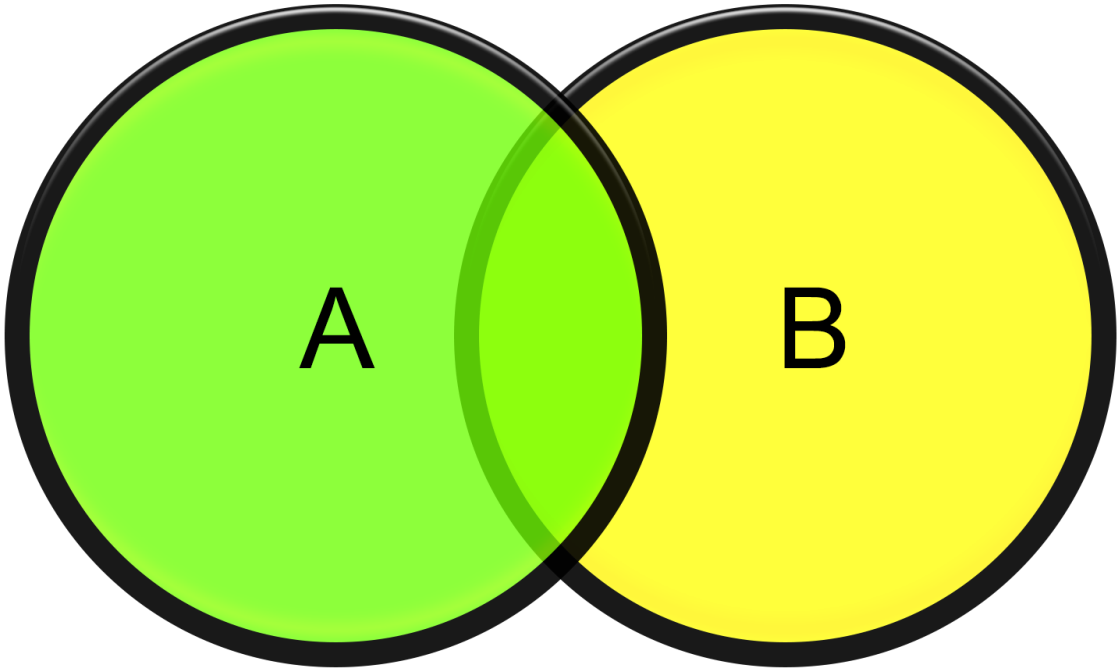

Figure 21: Venn Diagram

After declaring your set, you can initialize it, add and remove a subset of elements, find the intersection, and more.

Code Listing 14: Set Types

```
var
  CharSet: set of Char;

// Defines the initial set.
CharSet := ['a', 'b', 'c'];

// Add a subset of items to the initial set.
CharSet := CharSet + ['x', 'y', 'z'];

// Removes a subset of items.
CharSet := CharSet - ['a', 'x'];
```

We'll examine which operators can be applied to **`set`** values later in this chapter.

Record Types

Object Pascal allows you to create record types. Records are a way to group a set of variables into a single structure when their values represent a single piece of information.

For example, we can think of a **`TDateTime`** value as a group of two variables that respectively contain date and time pieces of information. You can also declare a **`TPoint`** record type to hold the X and Y values that represent a position.

Code Listing 15: Record Types

```
// Defines the record type.
type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;

// Declares a record variable.
var
  P: TPoint;
```

Tip: I recommend using a record type when the number of fields is limited and you can keep the structure as simple as possible. If you have to represent the state of a more complex object or move around references, records become inefficient and you should create a class instead.

Operators

As any other traditional programming language, Object Pascal supports many operators you can use to make calculations, concatenations and logical comparisons.

Arithmetical Operators

The following tables summarizes all the operators available in Object Pascal language, grouped by category.

Table 4: Arithmetical Operators

Operator Name	Symbol	Example of Usage
Addition	`+`	`A + B`
Subtraction	`-`	`A–B`
Multiplication	`*`	`A * B`
Division (real values)	`/`	`A / 2`
Division (integer values)	`div`	`Width div 2`
Remainder (integer values)	`mod`	`Height mod 2`

If you use integer values for addition, subtraction, and multiplication and integer division, the result type of the expression is again an integer value. If you apply a real division, the expression returns a floating point (real) value.

Comparison Operators

This set of operators allows you to express a condition that compares two type-compatible values and returns a Boolean result:

Table 5: Comparison Operators

Operator Name	Symbol	Example of Usage
Equality	`=`	`A = B`
Inequality	`<>`	`A <> B`
Less or equal than	`<=`	`A <= B`

Less than	`<`	`A < B`
Greater than	`>`	`A > B`
Greater or equal than	`>=`	`A >= B`

You can use them for integer and real values but also for chars and strings: the alphabetical order will determine the result.

Boolean Operators

You can implement Boolean logic using the operators listed in Table 6.

Table 6: Boolean Operators

Operator Name	Symbol	Example of Usage
Negation	`not`	`not (Proceed)`
Logical AND	`and`	`Valid and NotEmpty`
Logical OR	`or`	`A or B`
Exclusive OR	`xor`	`A xor B`

Note: Delphi uses something called Boolean short-circuit evaluation to determine the resulting value of a Boolean expression. Evaluation goes from left to right in the expression, and as soon as the result can be determined, evaluation stops and the expression value is returned. You can toggle this mode on (which is the default) or off (using "Complete Boolean Evaluation" instead) through the compiler directive {$B+}.

Set Operators

Some comparison and arithmetic operators can be used in expressions that include **`Set`** values and elements.

Table 7: Set Operators

Operator Name	Symbol	Example of Usage
Union	`+`	`SetA + SetB`
Difference	`-`	`SetA–SetB`

Intersection	`*`	`SetA * SetB`
Subset	`<=`	`SmallSet <= LargeSet`
Superset	`>=`	`LargeSet >= SmallSet`
Equality	`=`	`SetA = SetB`
Inequality	`<>`	`SetA <> SetB`
Membership	`in`	`Element in SetA`

Pointer Operators

You make use of Pointer Operators when you need to specify the memory address of some value, including the address of procedures and functions. Alternately, you can use pointer operators to remove a reference to an address to get to the effective value.

Table 8: Pointer Operators

Operator Name	Symbol	Example of Usage
Address of	`@`	`@MyValue` `@MyRoutine`
Value of	`^`	`^MyPointer`

Note: When you work with objects in Delphi, you are actually using references, which is nothing more than a pointer variable that contains the address to the object you want to manipulate. To keep the code clear and readable, you can omit the ^ operator to access object members (properties, fields, and methods).

Special Operators

There are further operators to mention that become handy in some special occasions.

Suppose you have to use a reserved word for the name of a procedure you want to export, because you are required to, or you want to give a meaningful name to something that coincides with a keyword. How can you do that without getting any errors?

Simply put an ampersand (**&**) symbol before the identifier.

Code Listing 16: "&" Special Operator

```
procedure &Begin;
begin
  // TODO: Put your code here...
end;

// Here we call the procedure.
&Begin();
```

Most programming languages support "escape sequences" to put some special characters into strings. In Delphi you can achieve this using the **#** character:

Code Listing 17: Character Codes

```
// We put a line break using the ASCII codes for CR+LF sequence.
Text := 'This is the first line,'#13#10'and this is the second one';

// Here we put a TAB character.
Text := 'Field Name:'#9'Field Value';
```

Structured Statements

Object Pascal supports many structured statements to let you implement loops, jumps, and conditional branching in your code.

Simple and Compound Statements

Object Pascal uses semi-colons (;) to terminate each statement. If you want to group one or more instructions together, you must enclose them in a **begin…end** block, creating a compound statement.

Code Listing 18: Simple and Compound Statements

```
// A single instruction.
WriteLn('Some text...');

// A block of more instructions.
begin
  WriteLn('The first line.');
  WriteLn('The second line.');
  WriteLn('The third line.');
end;
```

If-Then-Else Statement

The basic **If-Then-Else** statement has the following form.

Code Listing 19: If-Then-Else Statement

```
if Condition then
  statement
else
  statement;
```

The **`Condition`** part can be a Boolean variable or an expression that leads to a Boolean result.

When the condition is met, Delphi executes the simple or compound statement that follows the **`then`** keyword, otherwise it executes the statement after the **`else`** keyword. The **`else`** part is optional.

Case Statement

If you need to chain more **`if`** statements to specify what should be done when an expression assumes a set of values, use a **`case`** statement to avoid making your code cluttered and unreadable.

Code Listing 20: Case Statement

```
case Choice of
  1:
  begin
    // TODO: Instructions here...
  end;
  2:
  begin
    // TODO: Instructions here...
  end;
  3:
  begin
    // TODO: Instructions here...
  end;
  else begin
    // TODO: Instructions here...
  end;
end;
```

Note: Only scalar type variables can be used with the Case statement, and any case must be labelled with a constant value, so you cannot use object

references or floating point values. If you think this is a big limitation, be aware that using many case statements is discouraged and considered an "anti-pattern." It is better to use object-oriented programming and virtual methods to model a similar business logic.

Loop Statements

Object Pascal provides different statements to implement different kinds of loops in your code.

If you want to **repeat** a statement until a condition is met, you can write something similar to the following code.

Code Listing 21: Repeat Loop

```
repeat
   WriteLn('Enter a value (0..9): ');
   ReadLn(I);
 until (I >= 0) and (I <= 9);
```

If you want to check the condition before entering the loop statement, you can use the **while** structure.

Code Listing 22: While Loop

```
while not Eof(InputFile) do
begin
  ReadLn(InputFile, Line);
  Process(Line);
end;
```

If you have an explicit number of iterations to do, you can use the **for** loop structure.

Code Listing 23: For Loop

```
// Forward loop.
for I := 1 to 63 do
   if Data[I] > Max then
     Max := Data[I];

// Backward loop.
for I := 63 downto 1 do
   if Data[I] < Min then
     Min := Data[I];
```

If you have a data type that provides an iterator, like arrays, strings, or collections, you can use the **for…in** loop.

Code Listing 24: Enumerator Loop

```
var
  SomeChar: Char;
  SomeText: String;

// Iterates each char in a string.
for SomeChar in SomeText do
   WriteLn(SomeChar);
```

Exception Handling

Delphi includes support for exception handling. Exceptions let you write neat code while dealing with runtime errors.

Wrap the code to protect from errors between **`try...except`** and exception handling between **`except...end`**.

Code Listing 25: Exception Handling

```
try
  R := A div B;
except
  on E:EDivByZero do
  begin
    Writeln('Division by zero - ', E.ClassName, ': ', E.Message);
  end;
  on E:Exception do
  begin
    Writeln('Unexpected error - ', E.ClassName, ': ', E.Message);
  end;
end;
```

Through the **`on...do`** construct you can filter specific hierarchies of exceptions and store a reference to the **Exception** object that holds information about the issue.

Summary

If you thought that Object Pascal was the MS-DOS Turbo Pascal language you learned at school many years ago, you have to think again.

As you can see, Object Pascal—and especially the dialect used in Delphi—has been extended over the years with a lot of new features that are typical of any modern programming language. Like C# or Java, you can find support for operator overloading, anonymous methods, closures, generics and so on, and many more features are added in every new release of Delphi.

Object Pascal offers a balanced mix of powerful tools, a clear and readable syntax, without getting too verbose.

If you want to become an expert of Object Pascal and learn all of the most interesting details, check out the Object Pascal Handbook by Marco Cantù, the Delphi Product Manager.

Chapter 5 Object-Oriented Programming with Delphi

Delphi has full support for the object-oriented programming paradigm (OOP).

When you start writing your business logic, you can choose to follow an imperative procedural way of coding, therefore declaring global types, variables and routines, or leverage the support for object orientation and model your business logic creating classes with fields, methods and properties, interfaces and other typical elements of any OOP-language. You can even mix and match both styles, using the best parts of each.

This chapter shows how to do OOP programming with Object Pascal but it assumes that you are already familiar with its concepts.

Classes and Objects

The fundamental elements of OOP are classes and objects. A **class** defines a structure that contains data and logic: data is represented by fields, and logic is contained inside methods.

Each class can be seen as a template that defines the state and the behavior of any **object** that belongs to it.

Here is the full code of a unit containing a sample class declaration to represent an abstract two-dimensional shape.

Code Listing 26: Full Sample Class Declaration

```
unit Shape2D;

interface

uses
  System.Classes, Vcl.Graphics;

type

{ TShape2D }

  TShape2D = class abstract
  private
    FHeight: Integer;
    FWidth: Integer;
    procedure SetHeight(const Value: Integer);
    procedure SetWidth(const Value: Integer);
```

```
  protected
    function GetArea: Integer; virtual; abstract;
  public
    constructor Create;
    procedure Paint(ACanvas: TCanvas); virtual;
    property Area: Integer read GetArea;
    property Height: Integer read FHeight write SetHeight;
    property Width: Integer read FWidth write SetWidth;
  end;

implementation

{ TShape2D }

constructor TShape2D.Create;
begin
  inherited Create;
  FHeight := 10;
  FWidth := 10;
end;

procedure TShape2D.Paint(ACanvas: TCanvas);
begin
  ACanvas.Brush.Color := clWhite;
  ACanvas.FillRect(ACanvas.ClipRect);
end;

procedure TShape2D.SetHeight(const Value: Integer);
begin
  if Value < 0 then
    Exit;
  FHeight := Value;
end;

procedure TShape2D.SetWidth(const Value: Integer);
begin
  if Value < 0 then
    Exit;
  FWidth := Value;
end;

end.
```

We'll use this sample to explore some of the Object Pascal OOP features.

Classes, like any other Pascal type, can be declared in the **interface** section of a unit to make them available to other units. To use a class in the same unit where it is declared, put it in the **implementation** section.

Tip: You don't have to manually write every single line of a class implementation. Once you have completed the declaration of your class in the interface section, put the editor cursor inside of it and press CTRL+SHIFT+C to automatically create its implementation.

Ancestor Type

If you don't specify an ancestor class, Delphi assumes that you are extending **`TObject`**, which is the mother of all the classes and provides the basic support for memory allocation and generic object management.

Here is a sample descendant for our **TShape2D** class.

Code Listing 27: Sample Descendant Class Declaration

```
unit ShapeRectangle;

interface

uses
  Shape2D, Vcl.Graphics;

type

{ TShapeRectangle }

  TShapeRectangle = class (TShape2D)
  protected
    function GetArea: Integer; override;
  public
    procedure Paint(ACanvas: TCanvas); override;
  end;

implementation

{ TShapeRectangle }

procedure TShapeRectangle.Paint(ACanvas: TCanvas);
begin
  inherited Paint(ACanvas);
  ACanvas.Rectangle(ACanvas.ClipRect);
end;

function TShapeRectangle.GetArea: Integer;
begin
  Result := Height * Width;
end;

end.
```

The **TShapeRectangle** class is a descendant of **TShape2D**: this means that the class inherits everything that is part of the ancestor class, like the **Height** and **Width** properties, and it has the faculty of extending it by adding more members or change the logic. However, this applies only where the base class allows it and through specific extension points, like virtual (overridable) methods.

Delphi does not support multiple inheritance; you cannot inherit from more than one class.

Visibility Specifiers

The visibility of members inside the class does not depend on the section of the unit (**interface** or **implementation**) where they are declared or implemented, but there are specific keywords to assign visibility.

Table 9: Visibility Specifiers

Keyword	Description
private	Members are visible only to the class that contains them and by the unit where the class is declared.
protected	Members are visible to the class that contains them and all its descendants, including the unit where the class is declared.
public	Members are visible to the class that contains them, the unit where it is declared and to other units.
published	This has the same effect of the **public** specifier, but the compiler generates additional type information for the members and, if applied to properties, they become available in the Object Inspector.

Delphi also supports a couple of additional specifiers, **strict private** and **strict protected**. If applied to a member, this means members will be visible only to the class itself and not inside the unit where the class is declared.

Fields

The **TShape2D** class has some fields:

Code Listing 28: Instance Fields

```
TShape2D = class abstract
private
  FHeight: Integer;
  FWidth: Integer;
  // ...
end;
```

These fields are declared under the **private** section, therefore they are not accessible to client code outside the class. This means that changing their values must be done by calling a method like **SetHeight()** and **SetWidth()**, or by changing the value of a property linked to a field.

Note: You are seeing some coding conventions in action here: all the types usually start with the letter "T," and the fields start with the letter "F." If you follow these widespread conventions, you will be able to read code written by other people without much effort.

When an instance of this class is created, fields are automatically initialized to their default value, so any integer field becomes 0 (zero), strings are empty, Boolean are false, and so on.

If you want to set an initial value to any field, you must add a constructor to your class.

Methods

Here is an excerpt of the **TShape2D** method declarations taken out from our sample code.

Code Listing 29: Method Declarations

```
  TShape2D = class abstract
  private
    procedure SetHeight(const Value: Integer);
    procedure SetWidth(const Value: Integer);
  protected
    function GetArea: Integer; virtual; abstract;
  public
    procedure Paint(ACanvas: TCanvas); virtual;
  end;
```

Methods are functions and procedures that belong to a class and usually change the state of the object or let you access the object fields in a controlled way.

Here is the body of **SetWidth()** method that you can find in the **implementation** section of the same unit.

Code Listing 30: Method Implementation

```
procedure TShape2D.SetWidth(const Value: Integer);
begin
  if Value < 0 then
    Exit;
  FWidth := Value;
end;
```

Each instance of a non-static method has an implicit parameter called **Self** that is a reference to the current instance on which the method is called.

Our sample includes a virtual method marked with the **virtual** directive.

Code Listing 31: Virtual Method Declaration

```
  TShape2D = class abstract
  protected
    procedure Paint(ACanvas: TCanvas); virtual;
  end;
```

A descendant class that inherits from **TShape2D** can override the method using the **override** directive.

Code Listing 32: Overriden Method Declaration

```
  TShapeRectangle = class (TShape2D)
  protected
    procedure Paint(ACanvas: TCanvas); override;
  end;
```

The implementation in the descendant class will be called in place of the inherited method.

Code Listing 33: Overridden Method Implementation

```
procedure TShapeRectangle.Paint(ACanvas: TCanvas);
begin
  inherited Paint(ACanvas);
  ACanvas.Rectangle(ACanvas.ClipRect);
end;
```

The descendant method has the ability to call the inherited implementation, if needed, using the **inherited** keyword. The **Paint()** method in the base class fills the background with a color; the descendant classes inherit and call that implementation adding the instructions to draw the specific shape on the screen.

Properties

Properties are means to create field-like identifiers to read and write values from an object while protecting the fields that store the actual value, or using methods to return or accept values.

Code Listing 34: Properties

```
  TShape2D = class abstract
  private
    property Area: Integer read GetArea;
    property Height: Integer read FHeight write SetHeight;
    property Width: Integer read FWidth write SetWidth;
  end;
```

In the sample above, when you read the **Height** property, Delphi takes the value from the private **FHeight** field, which is specified after the **read** clause; when you set the value of **Height**, Delphi calls the **SetHeight** method specified after the **write** clause and passes the new value.

Note: Both the `read` and `write` accessors are optional: you can create read-only and write-only properties.

Properties give the advantages of fields and their ease of access, but keep a layer of protection to passed-in values from client code.

Constructors

Constructors are special static methods that have the responsibility to initialize a new instance of a class. Declare them using the **constructor** keyword.

Here is a sample constructor declaration taken from our sample class.

Code Listing 35: Constructor Declaration

```
  TShape2D = class abstract
  public
    constructor Create;
  end;
```

Let's examine the implementation of the constructor method.

Code Listing 36: Constructor Implementation

```
constructor TShape2D.Create;
begin
  inherited Create;
  FHeight := 10;
  FWidth := 10;
end;
```

The **inherited** keyword calls the constructor inherited from the base class. You should add this statement to almost every constructor body to ensure that the all the inherited fields are correctly initialized.

Some classes also define a **destructor** method. Destructors are responsible for freeing the memory allocated by the class, the owned objects, and releasing all the resources created by the object. They are always marked with the **override** directive so they are able to call the inherited destructor implementation and free the resources allocated by the base class.

Abstract Classes

When you think about a class hierarchy, there are times when you are unable to implement some methods.

Consider our ancestor **TShape2D** sample class: how would you implement the **GetArea()** method? You could add a virtual method that returns 0 (zero), but the zero value has a specific meaning. The definitive answer is that it does not make sense to implement the **GetArea()** method in **TShape2D** class, but it has to be there, since every shape logically has it. So, you can make it **`abstract`**.

Code Listing 37: Abstract Class

```
TShape = class abstract
protected
  function GetArea: Integer; virtual; abstract;
end;
```

Note: Each class that has abstract methods should be marked `abstract` itself.

The abstract method is devoid of implementation; it has to be a virtual method, and descendant classes must override it.

Code Listing 38: Implementing an Abstract Class

```
// Interface
TShapeRectangle = class
protected
  function GetArea: Integer; override;
end;

// Implementation
function TShapeRectangle.GetArea: Integer;
begin
  Result := Height * Width;
end;
```

Obviously you must not call the inherited method in the base class, because it is abstract and it would lead to an **AbstractError**.

Tip: The compiler issues a warning when you create an instance of abstract classes. You should not do that to avoid running into calls to abstract methods.

Static Members

Object Pascal also supports static members. Instance constructors and destructors are intrinsically static, but you can add static fields, members, and properties to your classes.

Look at an extract of the **TThread** class declaration from the System.Classes unit.

Code Listing 39: Static Members

```
  TThread = class
  private type
    PSynchronizeRecord = ^TSynchronizeRecord;
    TSynchronizeRecord = record
      FThread: TObject;
      FMethod: TThreadMethod;
      FProcedure: TThreadProcedure;
      FSynchronizeException: TObject;
    end;
  private class var
    FProcessorCount: Integer;
    class constructor Create;
    class destructor Destroy;
    class function GetCurrentThread: TThread; static;
  end;
```

The **ProcessorCount** variable is a **private** static member, and its value is shared by all the instances of **TThread** class thanks to the **class var** keywords.

Here you can also see a sample of **class constructor** and **class destructor**, a couple of methods aimed at the initialization (and finalization) of static field values, where **class function** is a static method.

Instantiating Objects

We have spent a lot of time on classes, but how can we create concrete instances?

Here is a sample of code that creates and consumes an instance of **TMemoryStream**.

Code Listing 40: Creating an Instance

```
var
  Stream: TMemoryStream;
begin
  Stream := TMemoryStream.Create;
  try
    Stream.LoadFromFile('MyData.bin');
  finally
```

```
    Stream.Free;
  end;
end;
```

The constructor called using the form **TClassName.Create** allocates the necessary memory to hold object data and returns a reference to the object structure.

The variable that holds the reference to the created object must be of the same type or an ancestor type.

Tip: If you want to make a reference type variable point to no object, you can set it to `nil`, a keyword that stands for a null reference value, like `null` in C#.

Call the method **LoadFromFile** to bring in the **TMemoryStream**. This will buffer all the data contained in the specified file (the path is specified as the first parameter to the method).

The object is then destroyed calling the **Free** method.

Note: Why not call the Destroy() method instead of Free()? After all, Destroy() is the "official" destructor. You must call Free() because it is a non-virtual method and has a static address, so it can be safely called whatever the object type is. The Free() method also checks if the reference is not assigned to `nil` before calling Destroy().

You might ask yourself what the **try…finally** construct stands for.

Delphi does not have a garbage collector; you are responsible for freeing every object you create. The **try…finally** block ensures that the resource is freed even if an error occurs during its use phase. Otherwise, a memory leak occurs.

Type Checking

You can check if an object belongs to a class using the **is** operator. If you get a positive result, you can cast the object to the specified type and access its members safely.

*Code Listing 41: Using the **is** Keyword*

```
if SomeObject is TSomeClass then
  TSomeClass(SomeObject).SomeMethod();
```

You can use the **as** operator to do type checking and casting at the same time.

*Code Listing 42: Using the **as** Keyword*

```
(SomeObject as TSomeClass).SomeMethod();
```

If **SomeObject** is not of **TSomeClass** type or a descendant of it, you get an exception.

Note: Never use the `is` and `as` operators at the same time, since they perform the same type-checking operations and that has some cost in terms of computing. If `is` operator checking is successful, use always a direct cast. However, never do a direct cast without checking before.

Interfaces

Interfaces are the thinner layer you can put between implementations, and are a fundamental tool for achieve high decoupling in code.

An interface is the most abstract class you can create in Delphi. It has only abstract methods with no implementation.

Code Listing 43: Interface

```
ICanPaint = interface
  ['{D3C86756-DEB7-4BF3-AA02-0A51DBC08904}']
  procedure Paint;
end;
```

Any class can only have a single ancestor, but it can implement any number of interfaces.

Note: Each interface declaration must have a GUID associated to it. This requirement has been introduced with COM support, but there are also lot of RTL parts where interfaces are involved that work with GUIDs. It could appear as an annoyance, but adding the GUID to the interface is really fast and simple: just press the CTRL+SHIFT+G key combination inside the Code Editor.

While classes have **TObject** as a common ancestor, interfaces all have **IInterface**.

Code Listing 44: Base IInterface Declaration

```
IInterface = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

The **IInterface** methods add support for reference counting. A class that implements an interface should inherit from **TInterfacedObject**, since this class provides the implementation for **IInterface** methods that are responsible for reference counting.

Code Listing 45: Interface Implementation

```
TShape = class (TInterfacedObject, ICanPaint)
  procedure Paint;
end;
```

Class Reference Types

Delphi includes a unique and powerful feature called class reference types (also called “meta-classes”).

Here is a sample declaration to clarify the concept.

Code Listing 46: Class Reference Type Declaration

```
TShapeClass = class of TShape;
```

Short and simple, but what does it mean? You can use this type for variables, fields, properties, and parameters, and you can pass the **TShape** class or one of its descendants as a value.

Suppose you want to declare a method that is able to create any kind of **TShape** object you want by calling its constructor. You can use the class reference type to pass the shape class as a parameter, and the implementation would be similar to the one shown in Code Listing 49.

Code Listing 47: Class Reference Type Usage

```
function TShapeFactory.CreateShape(AShapeClass: TShapeClass): TShape;
begin
  Result := AShapeClass.Create;
end;
```

Meta-classes allow you to pass classes as parameters.

Code Listing 48: Class Reference Type Value

```
MyRectangle := ShapeFactory.CreateShape(TRectangleShape);
```

Summary

In this chapter we have seen the basics of object-oriented programming with Object Pascal in Delphi. Since this is a *Succinctly* series book, we do not have all the space needed to explore every little detail of class implementation in Delphi.

If you really want to delve into all the possibilities of Object Pascal language and apply the most advanced programming techniques, like GoF Design Patterns, Inversion of Control, Dependency Injection and many more, I recommend the *Coding in Delphi* and *More Coding in Delphi* books from Nick Hodges.

Chapter 6 Making Real-World Applications

We have seen the fundamental IDE tools, built the “Hello World” application, and explored the basics of Object Pascal language and its syntax.

Now the time has come to delve into the intriguing aspects of building a real-world application. We’ll explore the main steps of creating a fully-functional, WordPad-like application with almost no code!

You can try to build the demo project using this chapter as a step-by-step tutorial, or you can download the full sample from GitHub.

Introducing the VCL

The **Visual Component Library** (VCL) is part of the product since its first version and has been designed mainly to wrap the Windows native APIs into a hierarchy of reusable and extensible classes, components and visual controls.

Applications that are built using VCL target the Windows platform only. If you want to go cross-platform, you must use the FireMonkey library (FMX), and the next chapter is dedicated to that. Fortunately, the most frequently used VCL components have already been made cross-platform, so they are available for both VCL and FMX.

The “V” of VCL acronym stands for “visual,” not only as a reference to visual controls, but also to Delphi’s “visual way” of building user interfaces and adding business logic.

Setting Up the Project

Since we want to build a VCL Forms Application, select **[File|New|VCL Forms Application]** from the main menu to create a new project and click **[File|Save all]** to save all the files generated by Delphi.

I usually tend to create a separate folder for my project. For this sample, you may create a folder named “DelphiPad.” Then save the Main Form as “Main.pas,” replacing the default meaningless proposed name (usually “Unit1.pas” or similar) and save the main project file with the name “DelphiPad.dproj,” putting them inside the previously created folder.

You can access the project files from the Project Manager window and open them with a double-click, but if you navigate to the project directory using Windows Explorer you will see that Delphi has stored several files in it.

Here is a summary of each file type and a brief description of its aim.

Table 10: Delphi Project File Types

Name	Name / Extension	Description
Delphi Project File	.dpr, .dpk	This file contains the main source code for a program or a library (.dpr) or a package (.dpk) and it usually is the entry point of your application
Delphi Project File (extended)	.dproj	This file type has been introduced to store all the project metadata that cannot fit inside the ".dpr" file. If you want to open a project, you can select either a ".dproj" or a ".dpr" file. If ".dproj" file is missing, e.g. when you open a quite old project, Delphi creates it automatically.
Delphi Form File	.dfm	This file contains the definition of a single Delphi Form or Data Module (which is a sort of non-visual container). Here is where Delphi stores the property values of forms and components. Data are usually stored using a human-readable text format, but you can switch to a more compact binary format. Delphi reads this file when you load the form inside the IDE at design time, and embeds it as a resource into the executable so it can load it and restore the form at runtime.
Pascal Code File	.pas	This plain text file contains Object Pascal source code and can optionally link a ".dfm" file if it contains the code-behind for a Form. As we have seen in the Object Pascal chapter, this file simply contains a Unit.
Resource File	.res	This kind of binary file contains resources like bitmaps, icons, strings, and cursors. It is a widely-adopted standard format created by a specific compiler, and it is directly embedded into executable files at build time. Delphi creates one to store Version Information about your project (title, company name, version and build number, etc.), but you can include additional resource files.
Temporary and Cache Files	.local .identcache	These files are used by Delphi to store fast access information during the development process. If you use a Version Control System (and I hope you are really using one!) add these files to the "ignore list" and delete them (when Delphi is not running) before distributing your source code solution to others.

Statistic file	.stat	This file contains statistical information about your project, like how many seconds you spent writing code, compiling, and debugging.

Note: Remember that you can always see the main files that make up your project inside the Project Manager window.

The first thing you might want to do is assign a main title to your project. Select **[Project|Options]** (or press Ctrl+Shift+F11) to bring up the Project Options dialog.

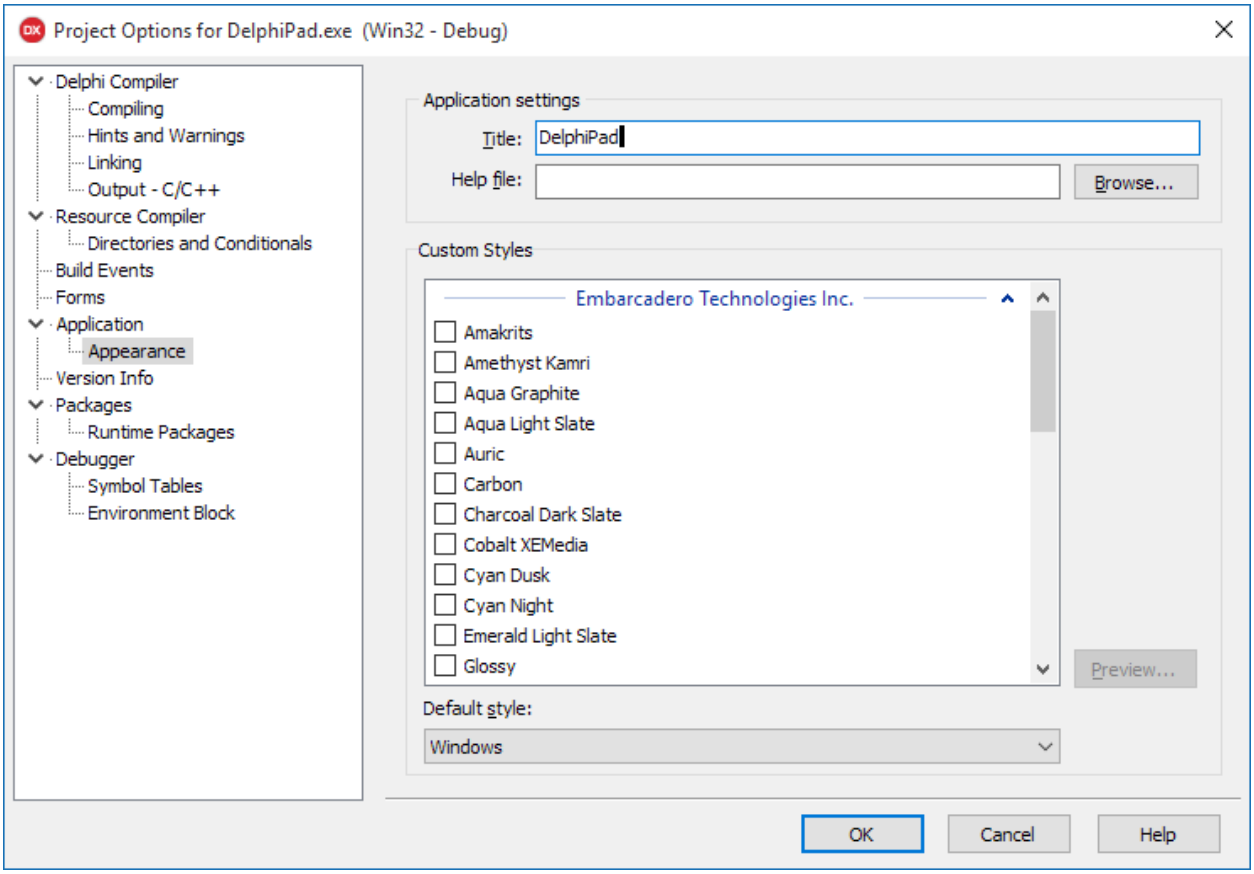

Figure 22: Project Options Dialog

You can use the navigation tree on the left to choose a page from the dialog and change the settings of your project. Select **[Application|Appearance]** page to affect the visual appearance of your application. For example, I have typed “DelphiPad” in the Title box, so that title will be displayed on the Windows task bar, task list, and everywhere there is an evidence of your running application.

Tip: Delphi has full support for themes. From the “Appearance” page you can choose one or more themes to embed in your project. One of them can be selected

as default at startup, but you can switch to another loaded theme in code or load it from an external file.

But where does that Title information go once submitted? Try to peek the Project File (.dpr) source code selecting **[Project|View Source]** from the main menu. You should see a piece of come similar to this:

Code Listing 49: VCL Application Main Program

```
program DelphiPad;

uses
  Vcl.Forms,
  Main in 'Main.pas' {Form2};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.Title := 'DelphiPad';
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

See that **`Application.Title`** assignment? Delphi automatically inserts it once you have typed the title and confirmed the Project Options dialog.

There are many circumstances where Delphi modifies the code in response to an action performed inside the IDE. To change the application title again, you can either edit the string directly in the Project source file or use the Project Options dialog.

Tip: I suggest you to stick as often as possible to the tools in the IDE where available, because directly editing the code by hand is much more prone to errors, even when you have become an expert.

Customize the Main Form

The Main Form has a central role for our application, and that goes for most of the projects created with Delphi. Let's see what options are available to customize it.

Double-click the "Main.pas" file from the Project Manager to open the Main Form if not visible. If Code Editor is displayed, press F12 (or select **[View|Toggle Form/Unit]**) to switch to the Form Designer window. You should already be familiar with it.

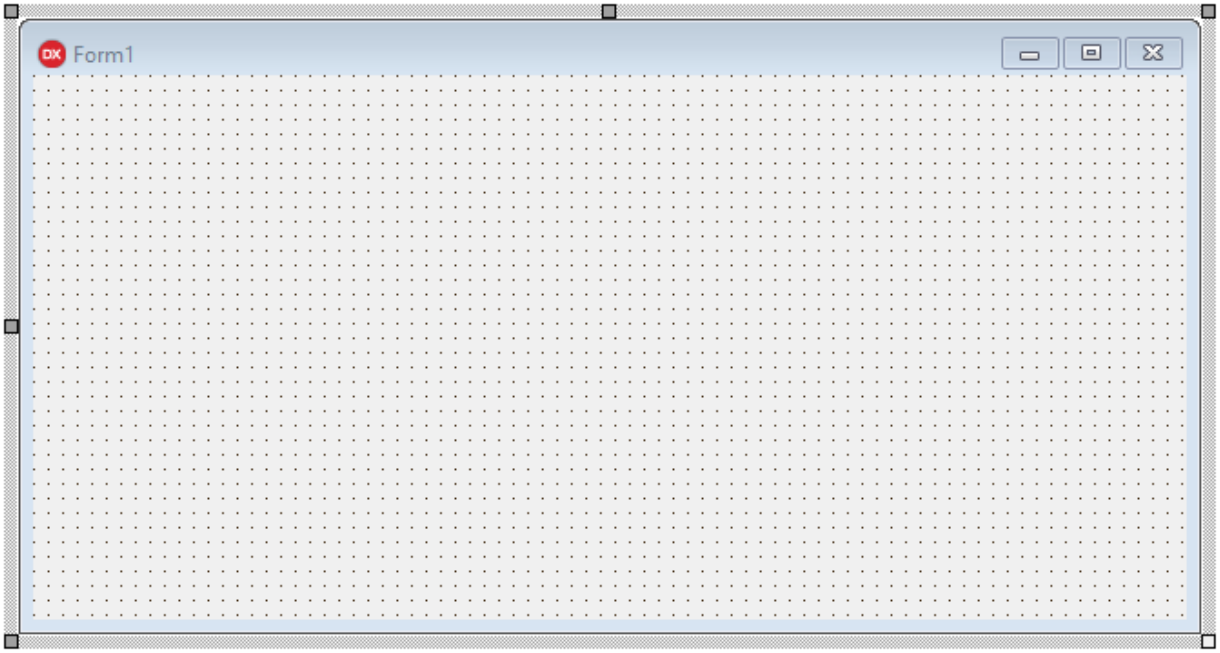

Figure 23: Empty Main Form

First, change the name of the form to make it more meaningful by going to the Object Inspector and setting the Name property to “MainForm.”

Note: If you switch to the Code Editor, you will notice that Delphi has renamed the form class name for you changing it to `TMainForm`. It is another situation where Delphi keeps the code in sync with your changes inside the IDE.

Then we change the caption replacing the default text (“Form1” or something similar) with the name of our application. Click on the form empty area, move to the Object Inspector panel, click Caption, and insert your title.

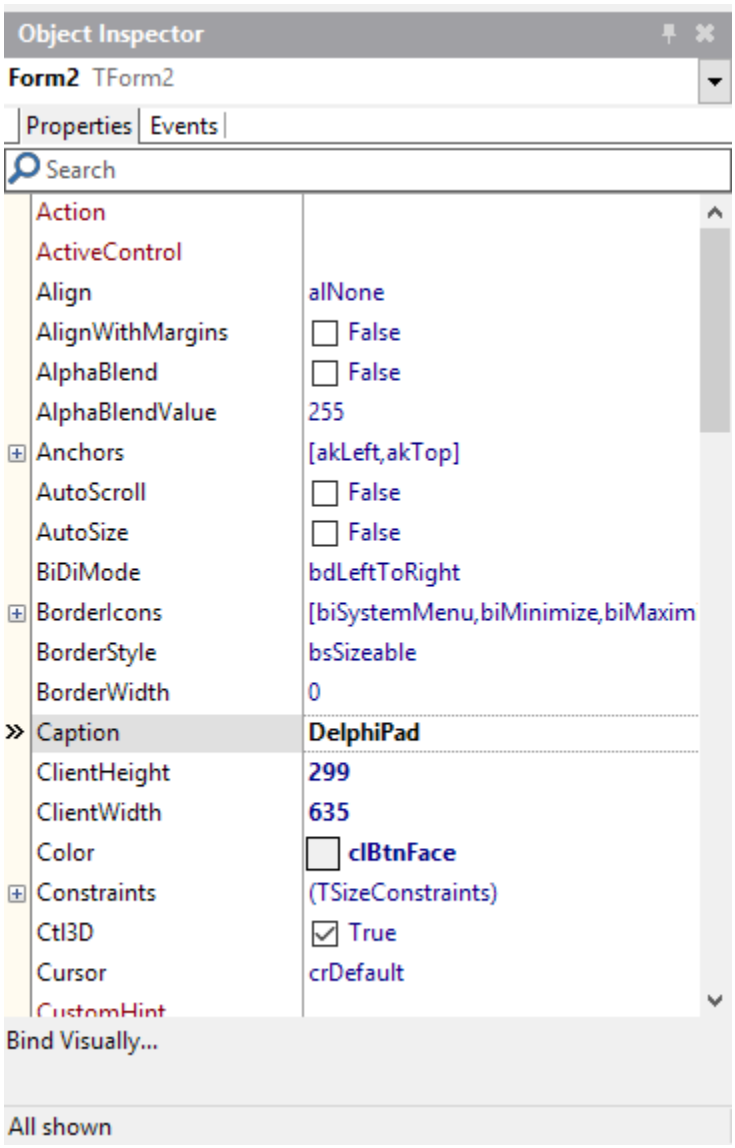

Figure 24: Form Properties

You can change many interesting properties on every form of your project, including the Main Form. Here is a list of the most frequently used ones.

Table 11: Form Properties

Name	Description
BorderStyle	Changes the appearance and the behavior of the border. You can make the form fully resizable by selecting the **`bsSizeable`** value (default) or restricting it with **`bsDialog`**; use **`bsToolWindow`** and **`bsSizeToolWin`** to achieve the same effects but make the title bar smaller, like a tool window.
KeyPreview	When enabled (set to **`true`**), key pressing events are raised before on the Form and later on the focused control.

Position	Sets the default position when the Form is displayed on screen. You can keep the default position dynamically assigned by Windows or center your Form on the screen.
ScreenSnap	When set to **`true`**, Form snaps to screen borders when you drag near them; you can adjust the effect using the **`SnapBuffer`** property.
Windows State	Sets the windows state that affects the size of the form on the screen; e.g. set it as **`wsMaximized`** to maximize the window at startup.

There are many other properties available. They are actually shared by many visual controls since they are introduced in a common ancestor class, **`TControl`**. Table 12 lists a few of them.

Table 12: Control Properties

Name	Description
Align	Setting this property, you can dock any control to the upper, lower, left and right border of its parent; the control retains its position while the parent is resized.
AlignWithMargins	Affects how the Align property works, applying a margin between the target control and the adjacent ones.
Anchors	Lets you put a virtual “pin” to the left, top, bottom, or upper side of the control so it follows the parent equivalent border when the latter is resized.
Constraints	Defines the minimum and maximum values for control size (width and height).
Cursor	Allows you to specify which cursor must be displayed when the mouse points to the control.
Height / Width	Sets the size of the control.
Hint	You can specify a hint string that will be displayed in a tooltip if you keep the mouse over the control client area.
Left / Top	Sets the position of the control.
Margins	Defines the spacing between the border and the outer controls when using alignment and the **`AlignWithMargins`** property is enabled.
Padding	Defines the spacing between the border and the contained controls when these uses alignment.
ParentColor	Dictates the control to use the same background control of its parent.
ParentFont	Dictates the control to use the same font of the parent control.

We are now ready to build the user interface of our custom Wordpad-like application.

To complete our demo we need:

- A main menu.
- A toolbar.
- A rich text editing area.

Creating a Main Menu

To add a main menu to the form, select the **TMainMenu** component from the “Standard” page of the Tool Palette and place it on the form.

Note: Even if the main menu has a visual representation, TMainMenu is not a control but a simple component that stores the menu configuration.

To define the contents of the main menu, double-click the **TMainMenu** component instance to recall the **Menu Designer** editor.

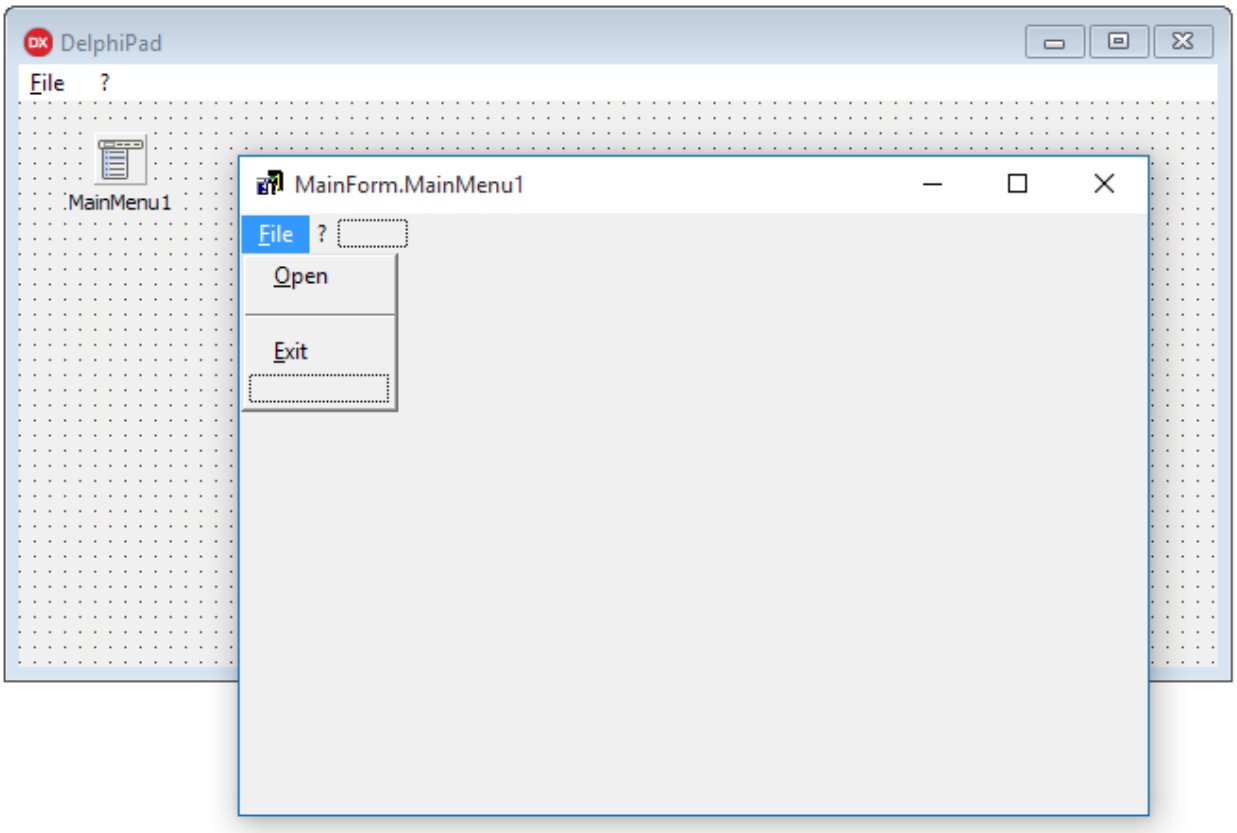

Figure 25: Menu Designer

Note: Many components support "Component Editors," which are special wizards that can be launched by double-clicking components and controls once they are added to the form. They are useful to speed up the setting of more property values at a time, automatically generate child components or perform high-level customizations using a more suitable and intuitive interface.

Click the blank item from the **Menu Designer** and enter a caption using the Object Inspector, you can then add further items above or below it. If you right-click any menu item, a popup menu will appear to load or save predefined menu as a template or transform any item into a submenu.

If you want to create a menu separator, create a menu item and insert "-" in the Caption property.

Tip: By the way, don't confuse TMainMenu with TPopupMenu component: both of them use Menu Designer to define menu items, but while TMainMenu creates a menu in the upper side of the form, TPopupMenu can be "attached" to some visual controls and components to appear when the user right-clicks on the target.

You can always launch the by program pressing Shift+Ctrl+F9, or selecting **[Run|Run Without Debugging]** to see if the main menu actually works as expected.

Adding a Main Toolbar

Many applications offer a toolbar that gives a fast way to execute the most frequently used commands.

You can do the same in Delphi: just select the **TToolBar** control from the *Win32* page of the Tool Palette and add it to your form. Notice that the control is aligned to the top by default, it's empty and ready to be configured.

Right-click the toolbar to show the popup menu. Select **New Button** and **New Separator** to quickly add command buttons and group separators. Add a couple of items just to test it: we'll add more buttons later with further configurations.

Set the **AutoSize** property to `true` to adjust the toolbar to the same height of its buttons and enable the **ShowCaptions** property to make button captions visible.

Defining Commands

Support for **Actions** is one of the Delphi features I like the most, and you will learn why in a minute.

Visual applications usually make "commands" available to the end user (i.e. open a file, make the selected text bold, etc.). Many developers often put the command logic directly inside the event handlers that respond to user actions on visual controls.

This kind of approach leads to code duplication or, in the best case, increases the complexity when you consider the need to execute the same command by clicking on both a menu item and a toolbar button. Things can get even worse if you must enable (or disable) these command controls when some commands should not be available to the user.

This is where Delphi “actions” come to the rescue.

Drag the **TActionList** component from the **Standard** page of the Tool Palette and drop it into your form. This component acts as a central point where you can define all the commands that are available to the user when the window is displayed and in focus.

Double-click the component to bring up the **Action List** Editor.

Note: The Action List component editor is actually a standard Collection Editor. It looks similar and works the same way for every component that has a property that inherits from the TCollection type.

Click the **New Action** button (or press the Insert button) to add a new action to the **Actions (VCL)** list box. Each action is an instance of **TAction** type and represents a single executable command.

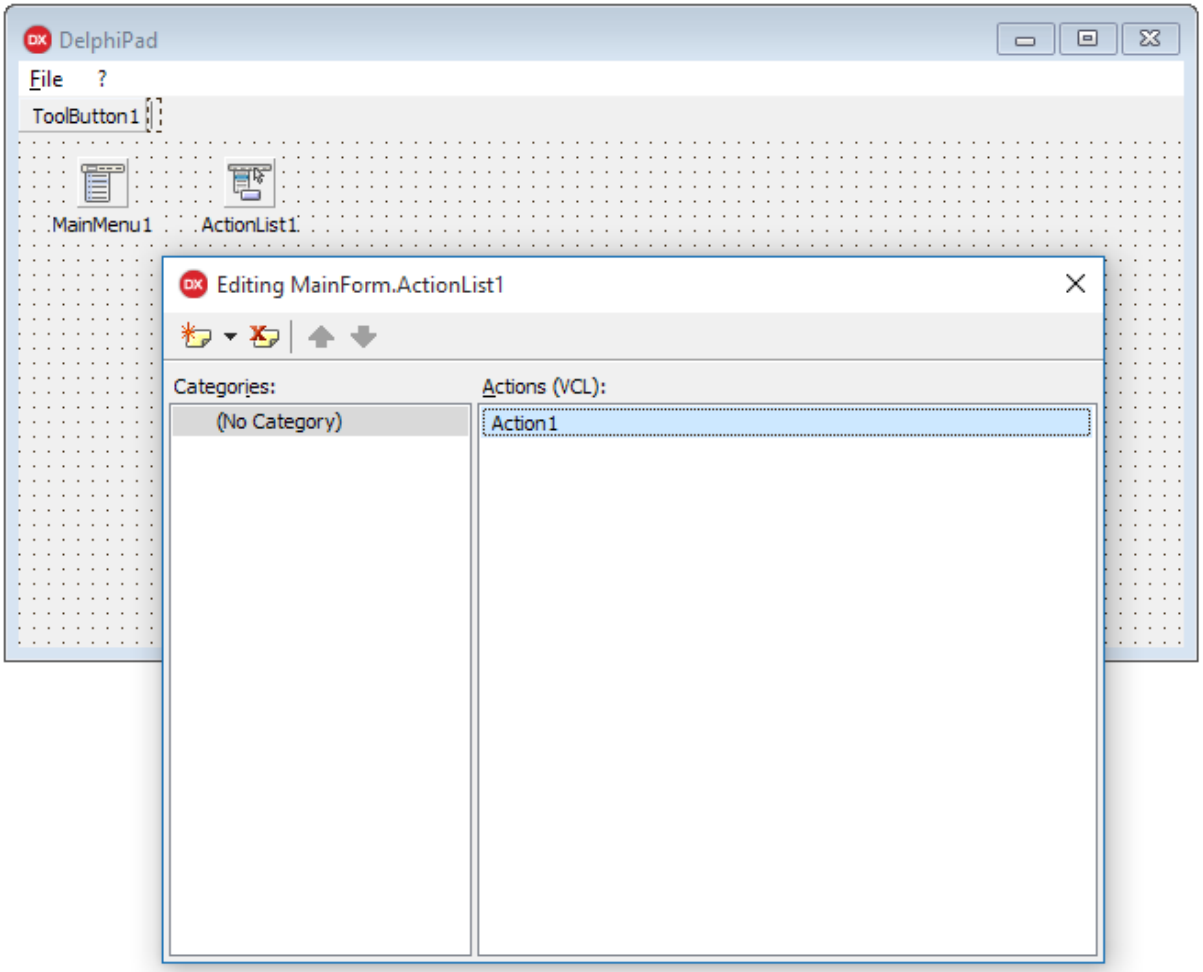

Figure 26: Action List Editor

Here are some **TAction** properties that are worth mentioning:

Table 13: Action Main Properties

Property Name	Description
Caption	This property contains the display text for any control that is bound to the action and is able to execute it (like menu items, standard buttons, and toolbar buttons).
Category	This property allows you to categorize the actions, dividing them into groups based on their context. For example, you can assign a “File” category to commands like “Open File” and “Close File.” The category name can be entered manually, or picked up from a list of categories already assigned to other actions.

Checked	This property holds the “checked state” for an action. You can change the value either at design time or at runtime. When the action is linked to a checkbox or a menu item, the state is displayed by the control and it will reflect any change. If **`AutoCheck`** property is enabled, the action automatically toggles the **`Checked`** property when it is executed.
Enabled	This property allows the developer to enable or disable the action. When the action is disabled, the user cannot execute it. Every control bound to this action will appear disabled too.
Hint	This property sets the text of the tooltip that will be displayed when the user points and holds the mouse on a linked control.
ImageIndex	This property contains the index of an image that will be displayed on linked controls, such as buttons and toolbars. The index refers to the offset of an image stored inside the **TImageList** component.
ShortCut	This property assigns a keyboard shortcut to the action. The user can execute the action through the selected key combination if the action has not been disabled.

For example, suppose we want to add an action to let the user exit our application, either by clicking the **[File|Exit]** main menu item or the “Exit” button on the toolbar. How could we accomplish this task?

First, add a new action to the list and set the following values for its properties:

Property Name	Value
Caption	Exit
Category	File
Hint	Exits the application
Name	FileExitAction
ShortCut	Ctrl+Alt+X

Switch to the **Event** tab in the Object Inspector and double-click the **OnExecute** event to create a new handler method. The method will contain the following code, which acts a response when the user executes the action.

Code Listing 50: Exit Action Execute Code

```
procedure TMainForm.FileExitActionExecute(Sender: TObject);
begin
  // Closes the main form and ends the application.
  Self.Close;
```

```
end;
```

The event handler is a method of **TMainForm** and is attached as a reference to the **OnExecute** event of **FileExitAction**.

Note: VCL Forms Applications normally terminate when Main Form is closed. You should always resort to this practice, avoiding any other brutal or exceptional way to close your application.

How can we link our **FileExitAction** to the visual controls that must execute it when clicked?

Open the Menu Designer on the **TMainMenu** component, select the **[File|Exit]** menu item and set the **Action** property by selecting **FileExitAction** from the dropdown list displayed in the Object Inspector. The job is done!

Repeat this procedure for the main toolbar. Add a new tool button to the **TToolBar** control using its pop-up menu and change the **Action** property like you just did for the main menu item.

You will notice that either the menu item or the tool button mirrors the **Caption** specified in the **FileExitAction** component. If you change your mind and decide to set a different caption, the controls will reflect the change. That is the hidden power of actions!

Using Images

A toolbar is not a real toolbar without any icons, and menu items deserve some nice graphics too.

You can add images using the **TImageList** component from the **Win32** page of the Tool Palette.

TImageList is a component that holds a set of images with a predefined size expressed through the **Height** and **Width** properties. The default size is 16x16 pixels, but you can increase it to 24x24, 32x32, 48x48, or any size value you want.

Double-click the component to recall the **ImageList Editor**. This dialog lets you choose images from your file system and assign them to actions, menus, toolbars, buttons, and other visual controls.

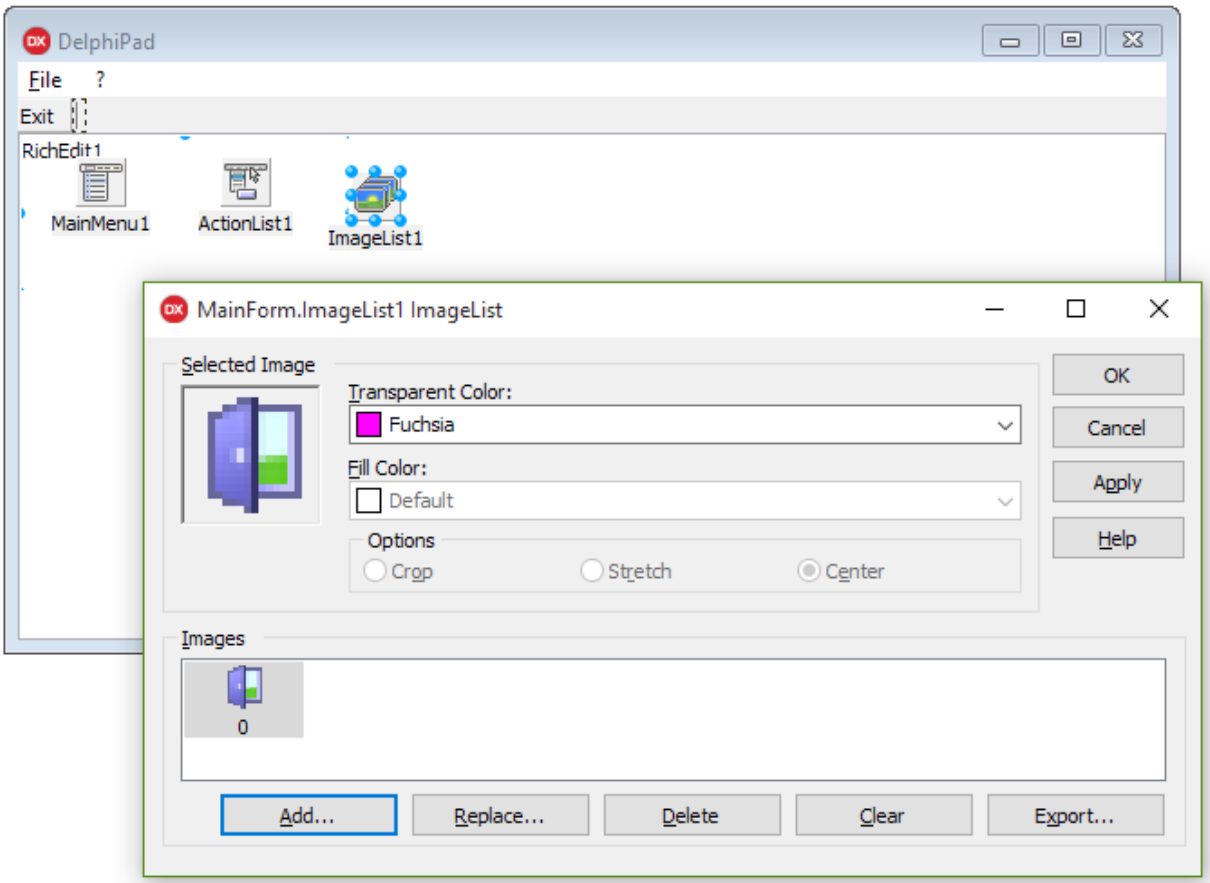

Figure 27: Image List Editor

The editor usually recognizes the background color on loaded images and considers it as a transparent color.

If you want to display the icons, you should link the **Images** property of **TActionList**, **TToolBar** and **TMainMenu** to the **TImageList** component using the Object Inspector as usual.

Each image is assigned to an index. Set the **ImageIndex** property of **TMenuItem** components, **TToolButton** controls, and **TAction** instances to the index of the image you want to assign to each element.

Adding a Rich Text Editor

The Visual Component Library provides a rich text editor out of the box called the **TRichEdit** control.

You can find it inside the **Win32** Tool Palette page. Drag it to your form and drop the control somewhere in the empty area.

Using the Object Inspector, set the **Align** property to **`alClient`** and the control will expand itself, filling up all the available space in the client area. When the form is resized, each control will retain its position and adapt to the new size according to the value specified in the **Align** property.

The **TRichEdit** control offers an editable area similar to the one you can find in Wordpad. You can input a line of text, change the size of the font, format words and paragraphs as bold, italic and underline, create bulleted lists, and so on.

At this point, do we really have to encode any single command we want to support on our Wordpad application? Thanks to **Standard Actions**, you don't have to!

Using Standard Actions

Standard Actions are **TAction** descendants that implement ready-to-use commands that are widely used in many business applications.

Go back to the **TActionList** components and call the ActionList Editor with a double-click.

Notice that **New Action** button has a dropdown menu that lets you choose from **New Action** (the command we have already used) and **New Standard Action**.

Selecting **New Standard Action**, Delphi will display a dialog to select one or more built-in actions you can add to your project.

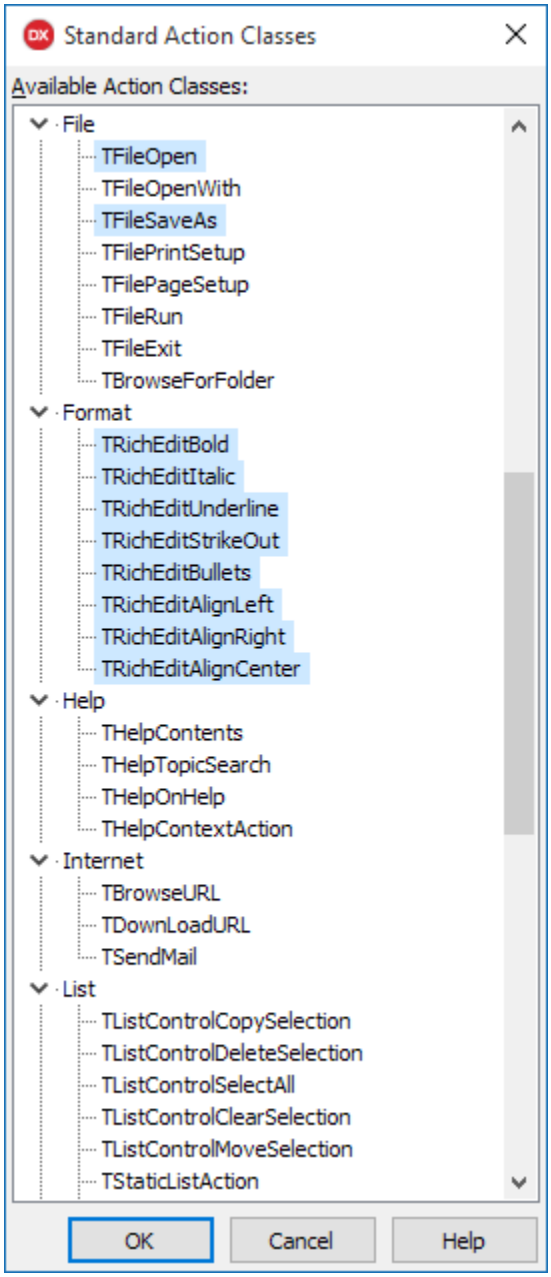

Figure 28: Standard Action Classes

Use Ctrl+Click to select more actions from the available action class list and press OK to add them to the ActionList component. Now they will become part of your project and assignable to menu items and tool buttons.

Tip: If you link an ImageList to an ActionList, when you add Standard Actions default icons are automatically inserted to the ImageList. You can always replace the new icons with images of your choice.

Standard actions come with a set of default values assigned to their properties, and they have more properties and events than default actions to customize their behavior and meet your specific business needs.

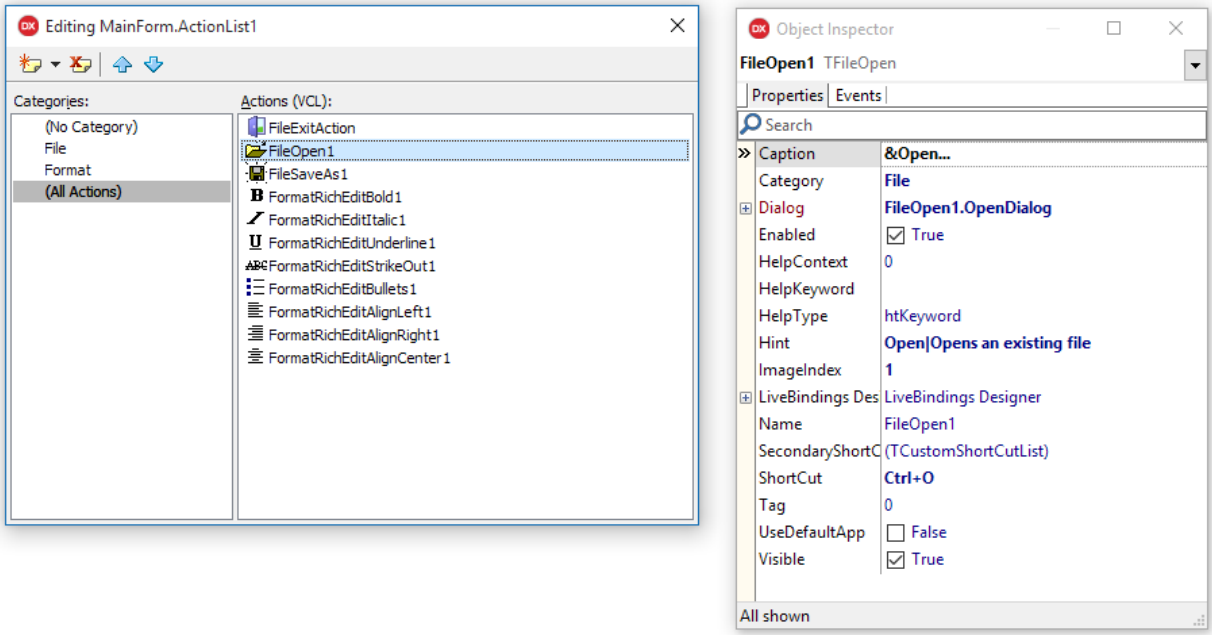

Figure 29: Standard Action Properties

Here is a screenshot of the finished application running.

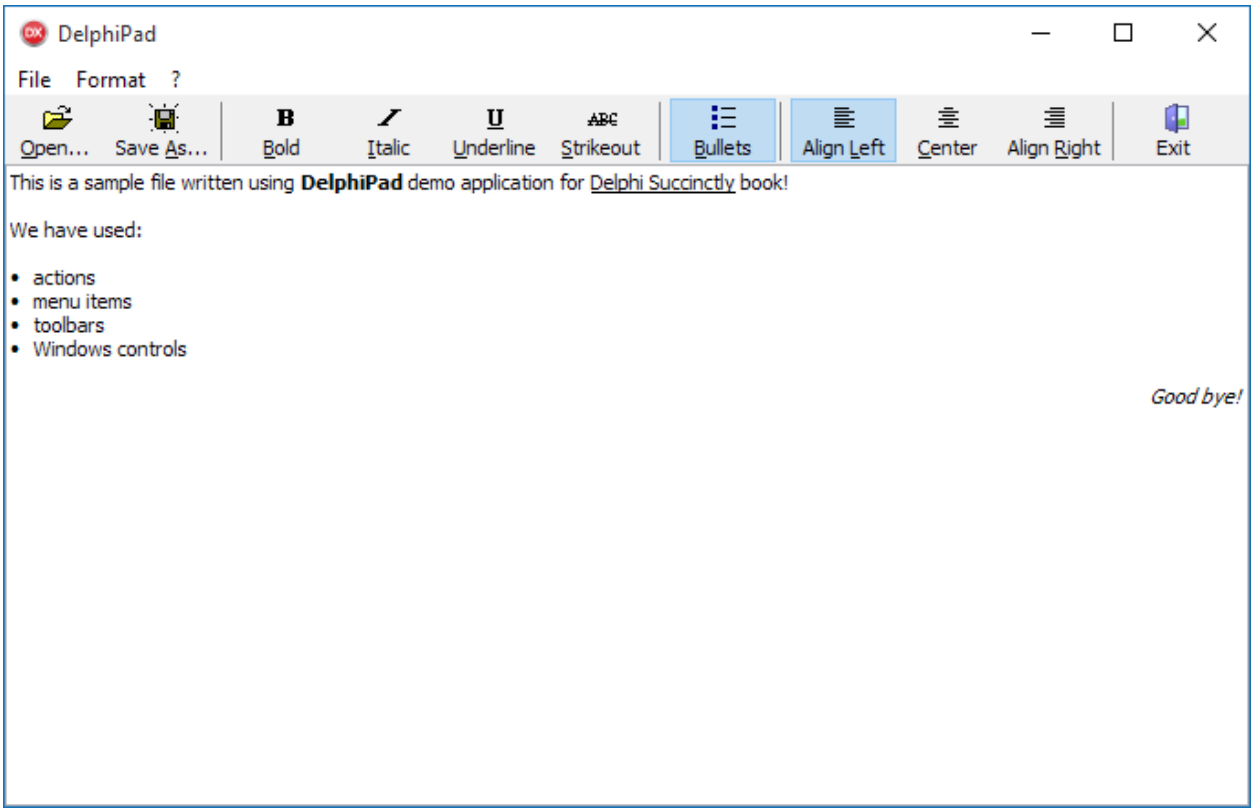

Figure 30: DelphiPad Sample Running

Summary

In this chapter we have just begun to explore the real potential that Delphi and the VCL can provide to develop rapidly a complete and fully functional application in just a few minutes.

We have created only a main form to keep the demo as simple as possible, but you can add any number of forms you want and create Data Modules (non-visible forms) to host your components. Examples include data-access components, ActionLists, ImageLists, and any other component, and sharing them with all the modules and forms that make up your application.

You can also add Frames from the **New Items** window to insert reusable pieces of user interface dragging them from the Tool Palette to the Form Designer when you need them.

Remember that Forms, Data Modules, and Frames are based on an Object Pascal class type, so you can add your own fields, properties, and methods to them as you would with other classes. Delphi even allows you to create new descendants from them visually.

Read the official Delphi documentation to expand your knowledge about all the smart tools you can use to build VCL Forms Applications.

Chapter 7 Cross-Platform Development with FireMonkey

FireMonkey is the name of the library that makes it possible to create cross-platform applications in Delphi, but it is effectively much more than that.

Actually, FireMonkey (abbreviated FMX) is a rather new framework born to develop next generation business applications and:

- It is a *graphics library*, since the elements that make up the user interface are vectors and leans to the GPU.
- It is *native*, because its code is compiled directly into the executable file and doesn't requires any runtime to work.
- It is *multi-platform*, because applications created with it can be compiled for different operating systems, such as Windows, Mac OSX, Android, and iOS.
- It is *abstract*, meaning that it is not merely a wrapper of the APIs exposed by targeted platforms.
- It comes with *source code*, unlike other competitor libraries with similar features, like Windows Presentation Framework (WPF), Silverlight, and Flex.

FMX also provides an abstraction layer for features such as windows, menus, dialogs, controls, timers, and sensors.

You can use FireMonkey to create 2D (or HD) applications and 3D applications, providing access to specific objects of a powerful vector graphics engine, supporting advanced capabilities like anti-aliasing in realtime, resolution independence, alpha blending, solid and gradient fills, brushes, pens, graphical effects, animation and transformations.

FMX facilitates the construction of complex user interfaces thanks to an extensive range of ready-to-use primitive objects and visual controls (shapes, buttons, text boxes, list and combo boxes, etc.) that you can compose by putting each one inside the others.

Like the VCL, FireMonkey embraces Unicode and supports skinning, theming, and the most popular image formats (JPEG, PNG, TIFF and GIF). However, it is incompatible with the VCL, since the latter is tied and coupled with the Windows native API, while RTL is the cross platform foundation of both FMX and VCL.

Creating a Multi-Device Application

Multi-Device Applications are how Delphi lets you build single-source cross-platform projects that target different devices (desktops, tablets and smartphones) and operating systems (Windows, Mac OS, Android and iOS) using the FireMonkey library.

In the rest of the paragraph, we will build a sample shopping list application.

To create a new Multi-Device Application, select **[File|New|Multi-Device Application]** from the Delphi main menu. This opens a wizard dialog that allows you to select an application type to use as a starting point. You can choose from the following:

- *Blank Application*, to start a new empty HD (2D) raw project.
- *3D Application*, to start with a 3D Form ready to host tridimensional objects.
- *Header/Footer*, to start with a 2D Form with top and bottom toolbars.
- *Header/Footer with navigation*, which is similar to the previous template but includes a page control.
- *Master-Detail*, to start a 2D project that uses the TMultiView control and supports switching between a list of items and the details of the selected item.
- *Tabbed*, to start a 2D project with tabs and the gesture support to switch between them.
- *Tabbed with Navigation*, which is similar to the previous template but adds navigational buttons to move to the next and previous tabs.

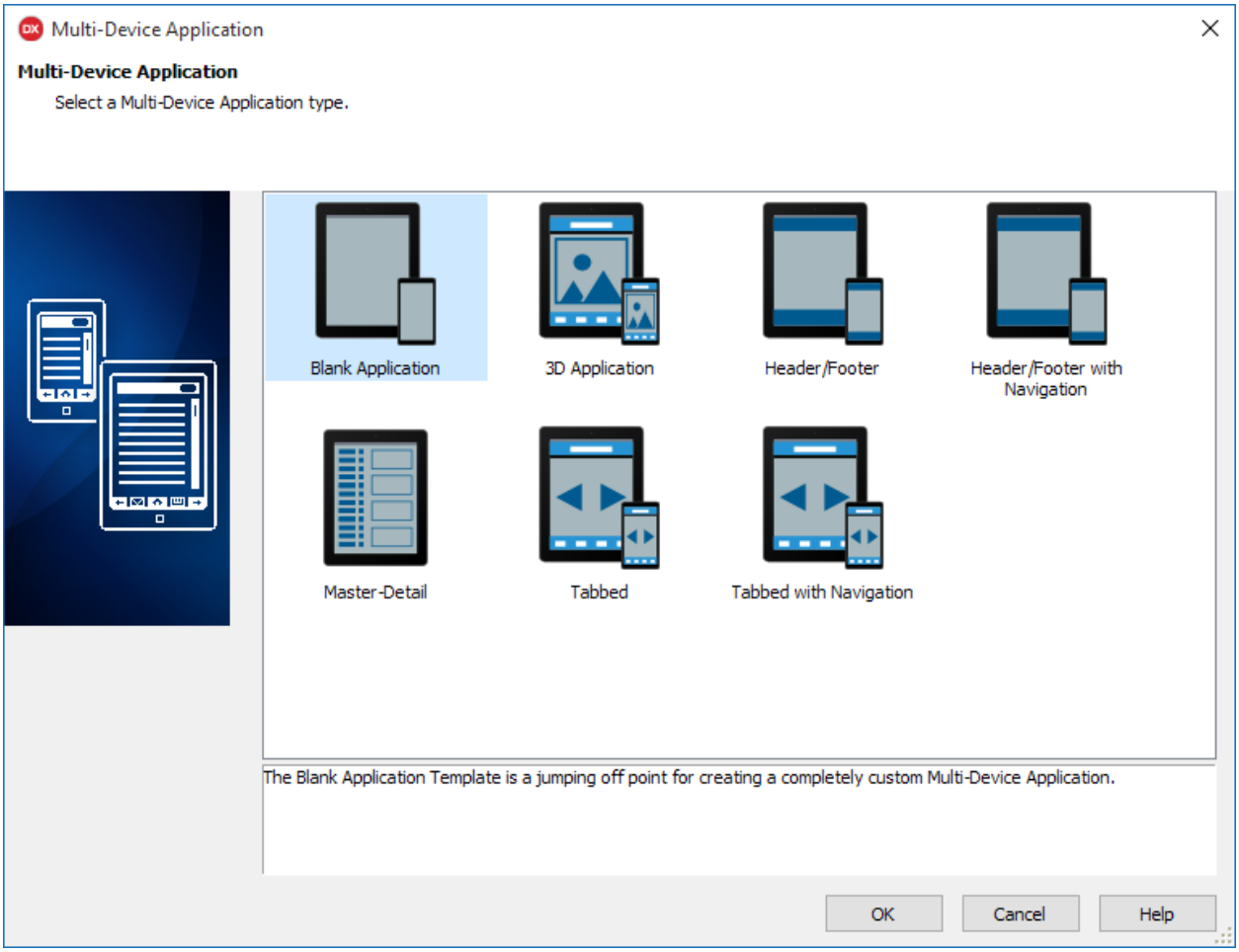

Figure 31: Multi-Device Application Templates

For our sample application, select the **Blank Application** template and click OK.

As with VCL Forms Applications, Delphi will create a default main empty form.

Displaying a List

Now we need to add a visual control to show our shopping list items. You should already be familiar with the Tool Palette. Locate the **`TListView`** component from the **Standard** page and drag it to your empty form. Move to the Object Inspector and set its name to **`ShoppingListView`**.

The **`TListView`** control can be used to display a list of items of any kind once the appearance has been configured. Items can be added through code at runtime or by binding the **`ListView`** to a data source.

Once you have added the control to the form, set the **Align** property to **Client** to make the control fill up the available client space. Then enable the **SearchVisible** property and a little search box will appear on the top of the list control. This will allow the user to enter a string to filter the contents displayed in the ListView.

Tip: TListView supports a "pull to refresh" feature. To enable it, set the `PullToRefresh` property to `True` and handle the `OnPullRefresh` event to implement your custom response to the gesture (i.e. reloading the updated items).

Defining a Data Source

Now that we have set up our list of items, we should bind it to a data source to see something inside our control and test how the application works.

It would be nice to have some auto-generated sample data and bind the **TListView** control to it. The **TPrototypeBindSource** component from the **LiveBindings** page can help you with this.

Add the **TPrototypeBindSource** to the main form and name it **ShoppingBindSource**.

Now it is time to add some fields. Right-click the component and select **Add Field…** to insert a new field in the virtual table of data produced by the component. The **Add Field** dialog will pop up.

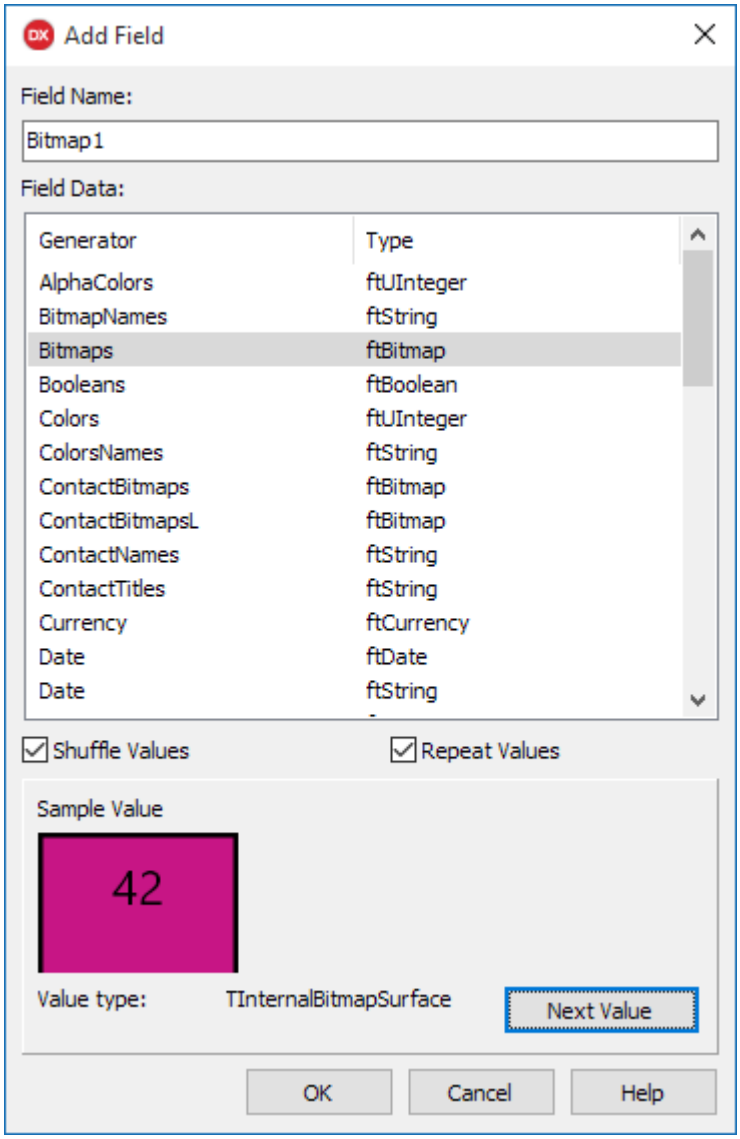

Figure 32: PrototypeBindingSource New Field

Browse the fields available in the **Field Data** list to get an idea of how many type of fields you can prototype data for, including colors and bitmap images.

Select the field associated to the **LoremIpsum: ftString** generator and click OK to add it. Then, set the **Name** property of the field to **Title**.

Using LiveBindings

LiveBindings is a technology available for both the VCL and FireMonkey frameworks.

It is based on binding expressions that link object properties together. For example, you can bind the **Checked** property of a toggle switch button to the **Enabled** property of any visual control to enable or disable it depending on the state of the button, without writing a single line of code.

Note: The VCL includes a special family of standard visual controls called "Data Controls" that supports data binding, while FireMonkey relies on LiveBindings for that.

You can access LiveBindings features through the set of components available in the LiveBindings page of the Tool Palette, but a better way to access them is through the **LiveBindings Designer**. You can open it selecting the **[View|LiveBindings Designer]** item from the main menu.

Drag the **Title** field from the **TPrototypeBindingSource** element to the **Item.Text** field of the **TListView** element.

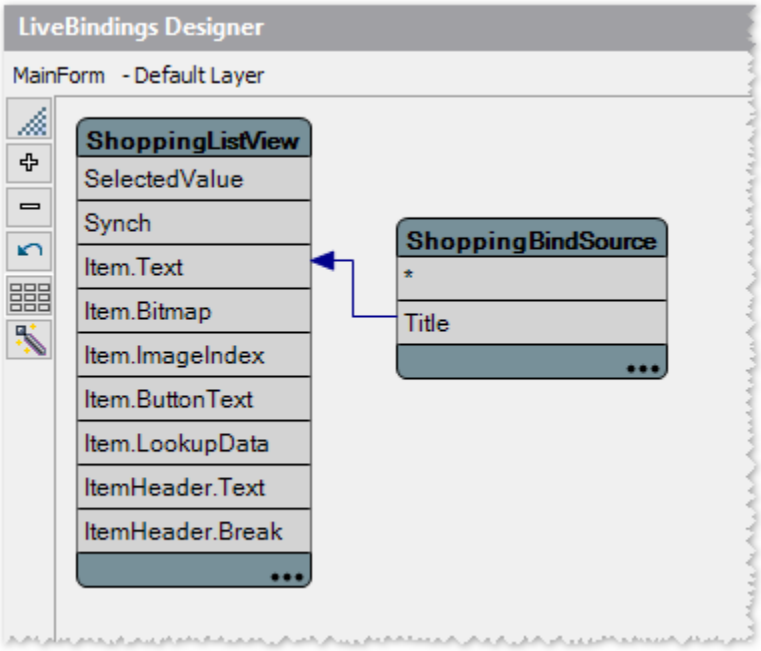

Figure 33: LiveBindings Designer

This will tell the control to use the **Title** property value of each prototype record as the text of each list item. You should already see a preview in the Form Designer and inside the Multi-Device Preview panel.

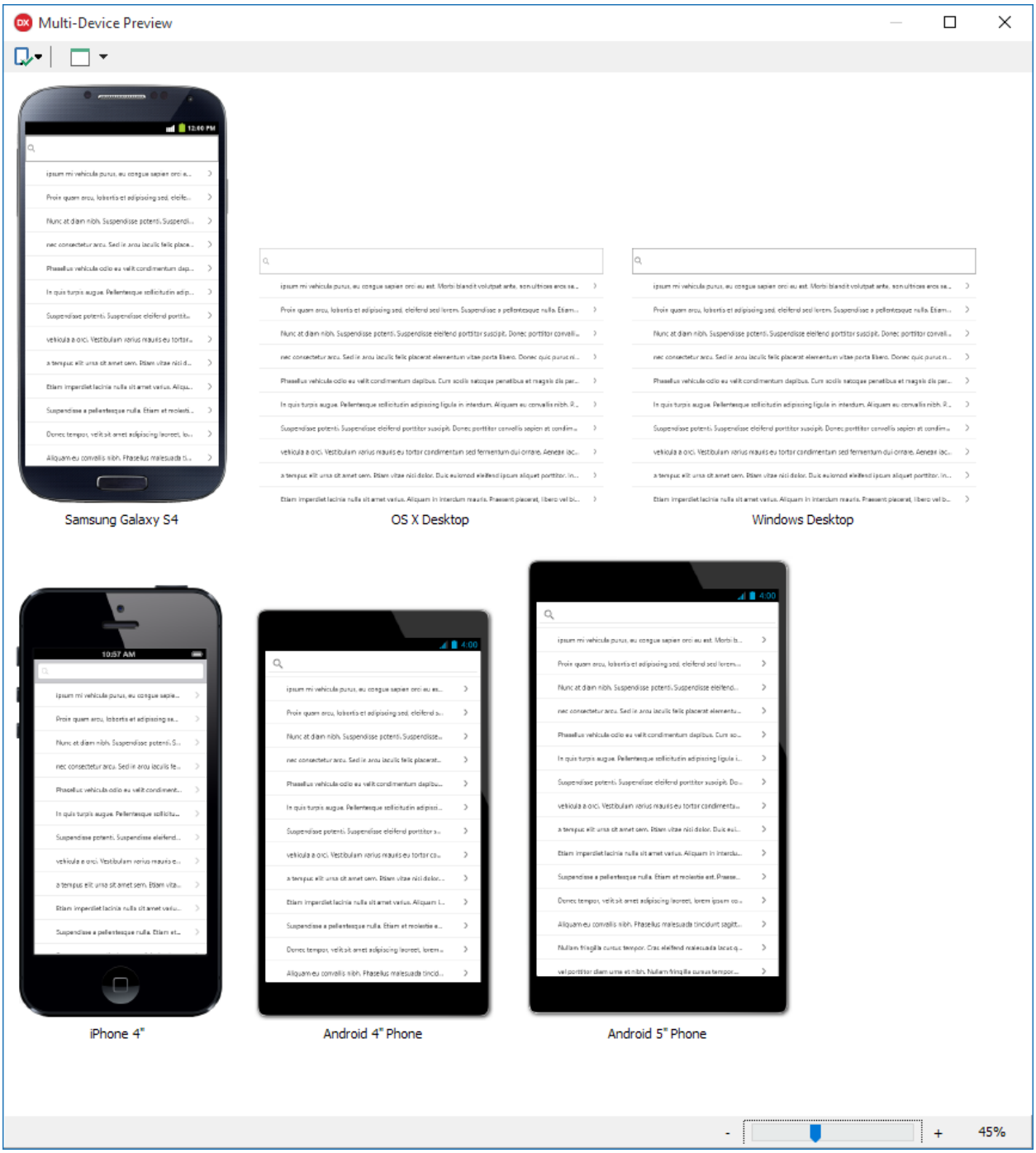

Figure 34: Multi-Device Preview

Accessing Data Using FireDAC

The **TPrototypeBindingSource** component—as the name implies—is good when you build a prototype of your application, but sooner or later you will have to use a real database to store your data if you want to release your product to the market.

FireDAC is a cross-platform data access library that provides a set of components and drivers for a wide range of database systems and formats, including:

- Microsoft SQL Server.
- MySQL.
- Oracle.
- SQLite.
- MongoDB.
- PostgreSQL.
- IBM DB2.
- InterBase (a database server from Embarcadero).
- FireBird (an open-source database born from a fork of InterBase).
- SQL Anywhere.
- Informix.
- Access.

FireDAC supports SQLite and InterBase (ToGo edition) for both Android and iOS, while it requires native client libraries for the other databases on client platforms.

Since we want to build a mobile application, we are using SQLite for the sake of simplicity and large availability.

The first step is creating a connection to the database. Select the **TFDConnection** component from the **FireDAC** page of the Tool Palette, then double-click the component to open the **FireDAC Connection Editor**.

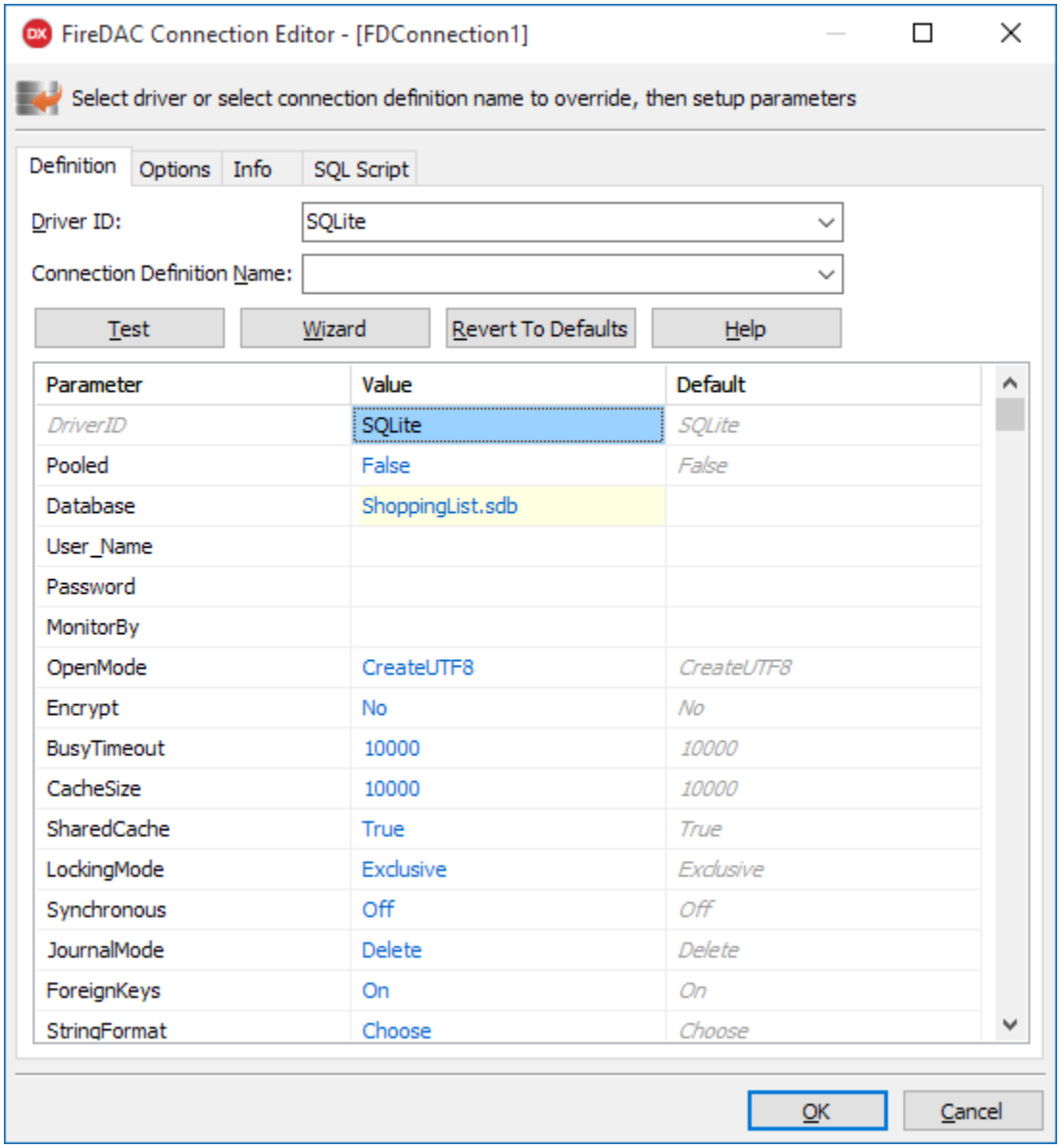

Figure 35: FireDAC Connection Editor

Select *SQLite* from the **Driver ID** list and enter the path "ShoppingList.sdb" in the **Database** field. Set the **LockingMode** parameter to **Normal**, allowing more than a connection to the database. Select OK to confirm the connection parameters. Remember to disable the login prompt, setting the **LoginPrompt** property to `False` from the Object Inspector.

To open the connection, set the **Connected** property to `True`. Don't worry if the database file does not exist; the component will create it for you automatically.

Now that we have a connection to our database, we can execute SQL queries on it using the **TFDQuery** component. You can even execute SQL statements live at design time.

Drop a **TFDQuery** component on your form and insert the following statement in the **SQL** property.

Code Listing 51: CREATE TABLE SQL Statement

```
CREATE TABLE IF NOT EXISTS ShoppingItem (Title TEXT NOT NULL)
```

Right-click the query component and select the **Execute** menu item to execute the statement immediately.

Now it's time to connect our **TListView** to the **ShoppingItem** table of the sample SQLite database.

The LiveBindings Designer can help again. Click the **LiveBindings Wizard** button to open a special wizard that creates instances of required binding components through a step-by-step guided process.

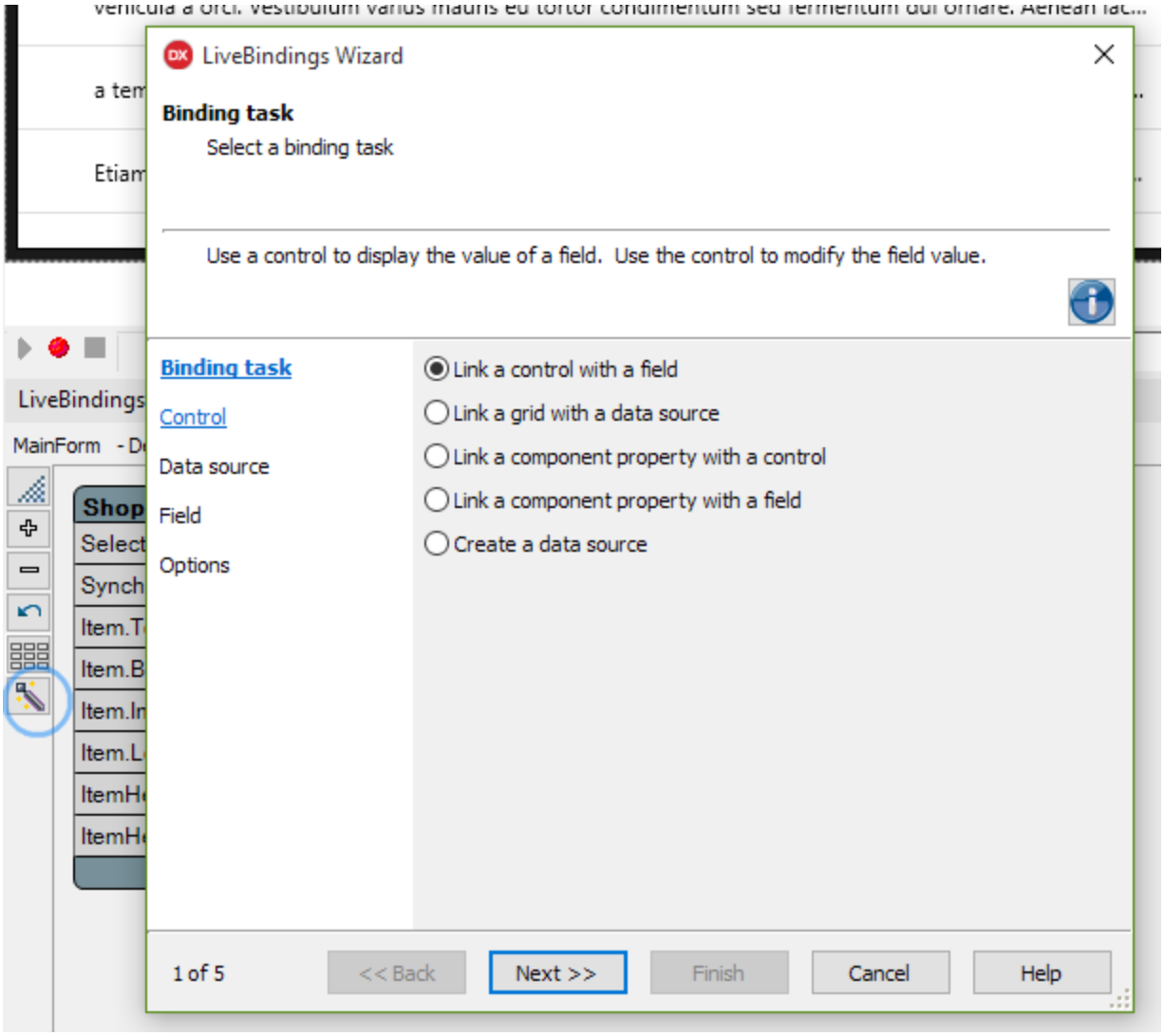

Figure 36: LiveBindings Wizard

After clicking the **LiveBindings Wizard**:

1. Select **Create Data Source** as the **Binding Task** to execute and click **Next**.

2. Select **FireDAC** as the **Data Source** and click **Next**.

3. Select **Query** as the **Command Type**, then enter **SELECT * FROM ShoppingItem** as the **Command Text** and click **Test Command** to check if the SQL statement is right. If everything is OK, then click **Next**.

4. Click **Finish** to end the wizard.

If everything has been done correctly, the wizard will add to your form:

- A **TFDQuery** component that is able to execute the SQL command entered before against the database.

- A **TBindSourceDB** component that makes the data retrieved by using the query available to LiveBindings and hence to the **TListView** control.

The only thing left to do is linking the data source to the **TListView** control. To perform this operation, call the **LiveBindings Wizard** again and:

1. Select **Link a control with a field** as the **Binding Task** to execute and click **Next**.

2. Select **Existing Control** and click the TListView instance in the list since it is the **Control** the data source must be linked to, then click **Next**.

3. Select the **TBindSourceDB** component as the **Data Source** that must be linked to the previously selected control, then click **Next**.

4. Select **Title** as the field name and click **Next** again.

5. Click **Finish** to end the wizard.

Note: The wizard might ask you to delete an existing link. This happens because our TListView is already linked to the TPrototypeBindSource component, and only one link is allowed to bind the control to any data source.

Adding Commands

To complete our sample application, we should add some command buttons to allow the user add new items to the list or delete existing ones.

Select the **TToolBar** control from the **Standard** page of the Tool Palette and drop it on the main form. We'll add a couple of buttons and a title label to it.

Drag a **TButton** control from the **Standard** page and configure it through the Object Inspector to act as a command to add a new item to the shopping list, setting:

- The **Align** property to **Left**.

- The **Name** property to **AddButton**.
- The **StyleLookup** property to **addtoolbutton**.

Add another **TButton** to the **TToolBar** control. This will delete the currently selected item, and set:

- The **Align** property to **Right**.
- The **Name** property to **DeleteButton**.
- The **StyleLookup** property to **deleteitembutton**.

Tip: FireMonkey comes with a set of predefined styles you can assign to controls through the `StyleLookup` property.

At the end, add a **TLabel** control to the toolbar to act as a title bar and set:

- The **Align** property to **Client**.
- The **Name** property to **TitleLabel**.
- The **StyleLookup** property to **toollabel**.
- The **Text** property to **Shopping List**.

Responding to User Actions

We are done with the user interface design. You must add the code that responds to user actions.

Suppose you want to add a new item to the list and save the data into the table we have previously created. You will need to execute an INSERT command in order to do that.

Add a **TFDQuery** component to your form, name it **InsertQuery** and set the **SQL** property to the following statement.

Code Listing 52: INSERT SQL Statement

```
INSERT INTO ShoppingItem (Title) VALUES (:Title)
```

When you write SQL statements using FireDAC, you can use the colon (:) character to represent parameters. You must remember to assign a value to parameters before executing the SQL statement.

Select the **TFDQuery** component and click the ellipsis button beside the **Params** property in the Object Inspector to open the Collection Editor. Select the **Title** parameter to configure its properties. Because we will pass a text string to this parameter—the title for a new shopping list item—set the **DataType** property to **ftString** before executing the **`INSERT`** query.

Add the following private method to the main form class.

Code Listing 53: Input Query Event Handler

```
procedure TMainForm.OnInputQueryClose(const AResult: TModalResult;
  const AValues: array of string);
var
  Title: string;
begin
 Title := string.Empty;
  if AResult <> mrOk then
    Exit;
  Title := AValues[0];
  try
    if (Title.Trim <> '') then
    begin
      InsertQuery.ParamByName('Title').AsString := Title;
      InsertQuery.ExecSQL();
      SelectQuery.Close();
      SelectQuery.Open;
      DeleteButton.Visible := ShoppingListView.Selected <> nil;
    end;
  except
    on e: Exception do
    begin
      ShowMessage(e.Message);
    end;
  end;
end;
```

When asked to insert the title of the new item, this piece of code will handle the response to the user input.

Double-click the **AddButton** control to create a handler method for its default event, **OnClick**, and insert the following code.

Code Listing 54: Add Button Click Event Handler

```
procedure TMainForm.AddButtonClick(Sender: TObject);
var
  Values: array[0..0] of string;
begin
   Values[0] := String.Empty;
   InputQuery('Enter New Item', ['Name'], Values,
    Self.OnInputQueryClose);
end;
```

This will display an input text box to let the user enter the title for a new item. If the user confirms the operation, the callback method **OnInputQueryClose** will be called and add the item to the list by executing the **INSERT** statement wrapped into the **TFDQuery** component.

In order to delete an item from the list, the first step is to add a new **TFDQuery** component. This is similar to the previous instance, but replaces the **INSERT** command with **DELETE**.

Code Listing 55: DELETE SQL Statement

```
DELETE FROM ShoppingItem WHERE Title = :Title
```

Then you must handle the event that is raised when the user clicks **Delete**.

Code Listing 56: Delete Button Click Event Handler

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
var
  Title: String;
begin
  Title := TListViewItem(ShoppingListView.Selected).Text;
  try
    DeleteQuery.ParamByName('Title').AsString := Title;
    DeleteQuery.ExecSQL();
    SelectQuery.Close;
    SelectQuery.Open;
    DeleteButton.Visible := ShoppingListView.Selected <> nil;
  except
    on e: Exception do
    begin
      ShowMessage(e.Message);
    end;
  end;
end;
```

You should avoid displaying the Delete button when no items are selected from the list. To ensure that, just add the following event handler when the user clicks on the **TListView** control and you are done.

Code Listing 57: ListView Click Event Handler

```
procedure TMainForm.ShoppingListViewClick(Sender: TObject);
begin
  DeleteButton.Visible := ShoppingListView.Selected <> nil;
end;
```

Running the Application

While you are developing your Multi-Device Application, you can always run it on your Windows desktop to test if it works as expected. The Delphi compiler for the Windows platform is fast and lets you quickly see your result.

If you want to see how the application behaves in a mobile platform, you should install an emulator or deploy your app to a physical device, connecting an Android device to your Windows machine or installing and running the app on an iOS device connected through a Mac OS machine.

The official documentation provides the procedure for setting up your Android device and iOS device.

Suppose you have an Android device. To run the application on your mobile device, you should connect the device to the USB port, enable the USB debugging feature on the device, and install the Windows device driver.

If you have done everything right, you should see your Android device listed in the Project Manager window, under the Target folder of the Android target platform.

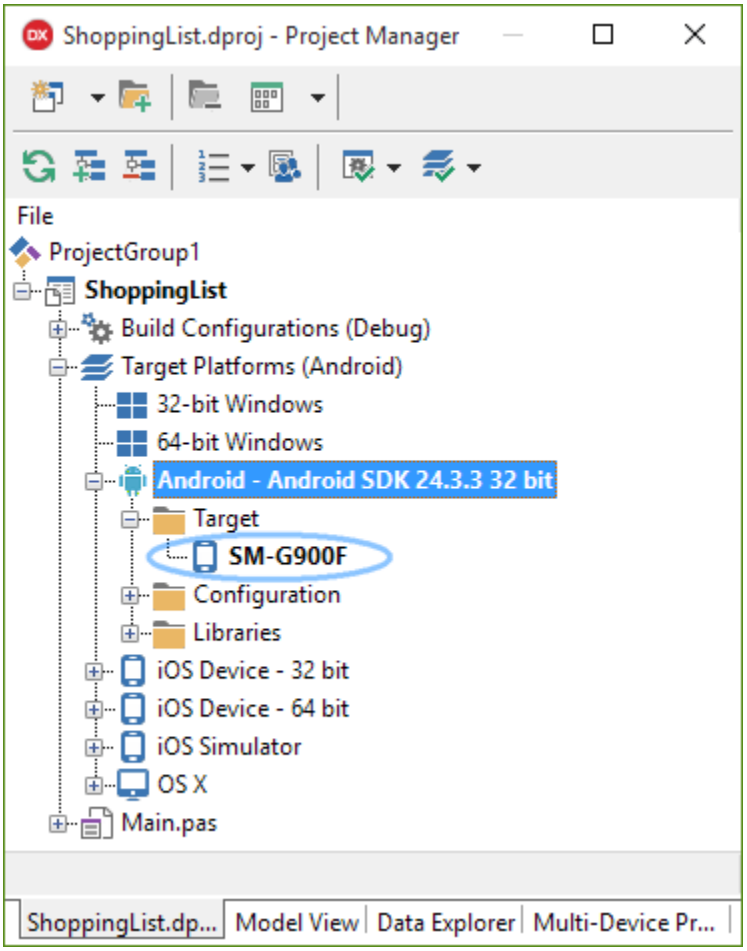

Figure 37: Android Target Platform

With the Android device selected, select **[Run|Run Without Debugging]** to start the build process. Deploy the app on your Android device, installing and running it. Select **[Run|Run]** to run the application while attaching the debugger to the process and perform a controlled execution.

Summary

In this final chapter, we have seen how you can build cross-platform application from a single source code using Delphi.

Unified desktop and mobile development is possible thanks to FireMonkey, a new generation graphic library that makes possible to build complex user interfaces while it takes care of the differences in terms of styles and behavior.

LiveBindings technology comes to help connecting controls to other components, and vice versa, and is the core foundation of data binding in FireMonkey where it let you create bindings to connect data from a data source to your visual controls, without writing any code.

The task of accessing data is delegated to FireDAC, a high performance data access library that provides many components to connect to a wide range of databases.

This chapter ends our journey inside the marvelous world of Delphi and Object Pascal development, but keep in mind that there are lots of additional features that we did not cover that still add a priceless value to the product. These include Windows 10 support, Gesture Management, Beacon and Bluetooth support, App Tethering, App Analytics, Enterprise Mobility Services (EMS), and many other technologies and libraries.