Introduction to Programming with Xojo

A TEXTBOOK

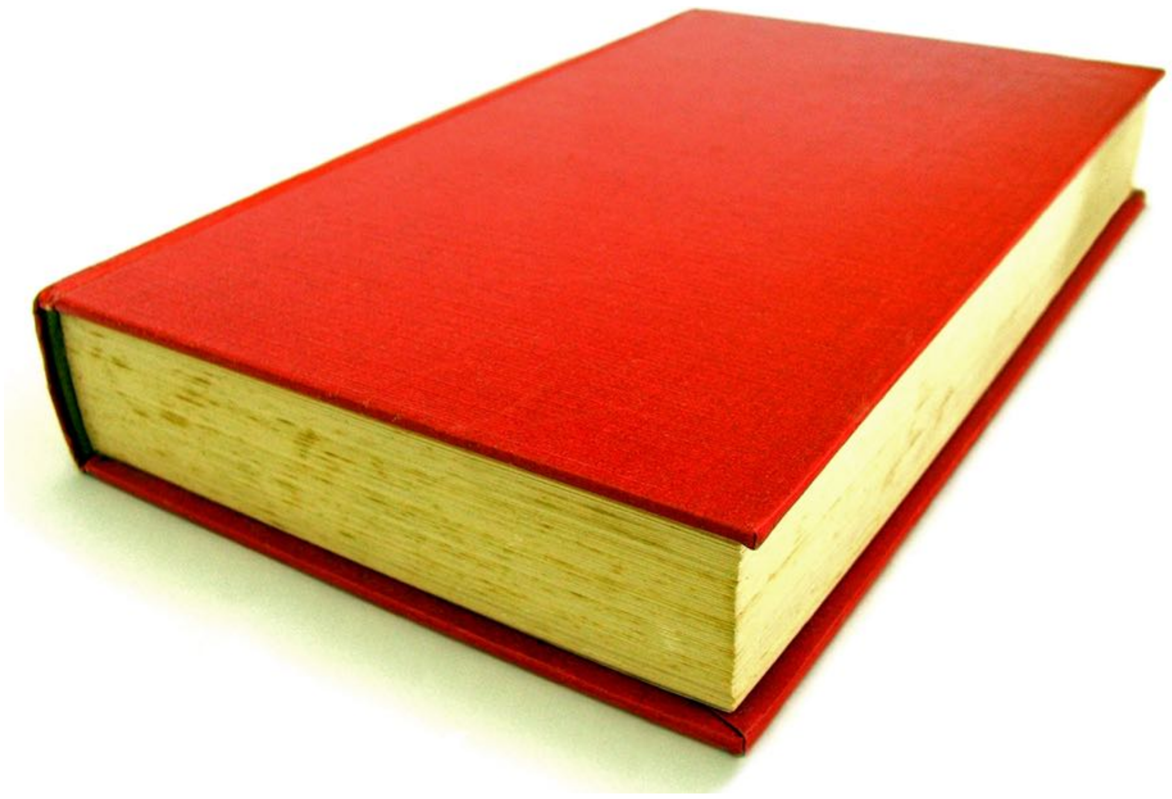
RHINE, LEFEBVRE

XOJO®

Introduction

CONTENTS

Foreword



When you finish this book, you won't be an expert developer, but you should have a solid grasp on the basic building blocks of writing your own apps. Our hope is that reading *Introduction to Programming with Xojo*, will motivate you to learn more about Xojo or any other programming language.

The hardest programming language to learn is the first one. This book focuses on Xojo - because it's easier to learn than many other languages. Once you've learned one language, the others become easier, because you've already learned the basic concepts involved. For example, once you know to write code in Xojo, learning Java becomes much easier, not only because the languages are similar and you already know about arrays, loops, variables, classes, debugging, and more. After all, a loop is a loop in any language.

So while this book does focus on Xojo, the concepts that are introduced are applicable to many different programming languages. Where possible, some commonalities and differences are pointed out in notes.

Before you get started, you'll need to download and install Xojo to your computer. To do so, visit http://www.xojo.com and click on the download link. Xojo works on Windows, macOS and Linux. It is free to download, develop and test - you only need to buy a license if you want to build standalone apps.

Acknowledgements

Special thanks go out to Brad Rhine who wrote the original versions of this book with help from Geoff Perlman (CEO of Xojo, Inc).

We'd also like to acknowledge the Xojo community. We often hear your stories about how Xojo made it easy for you to learn programming or build an app or start your own business. You have made Xojo possible for over 20 years.

Conventions

Because Xojo can run on different operating systems and build apps for different operating systems, some of the screenshots in this book were taken on Windows and some were taken on macOS . One of the sample apps is web-based, so you'll see that its screenshots were taken in a web browser.

As you read this book, you also will notice different formats of text.

One format is code examples. Anything in a code example is meant to be typed into Xojo exactly as it appears on the page. A code example looks like this:

```
Var x As Integer
Var y As Integer
Var z As Integer
x = 23
y = 45
z = y * x
MessageDialog.Show(z.ToString)
```

Another format you will see is steps. A step looks like this:

1) **This is something you're supposed to do. You might be asked to set "Some Text" as something's caption. If that happens, type what's inside the quotation marks, but not the quotation marks themselves.**

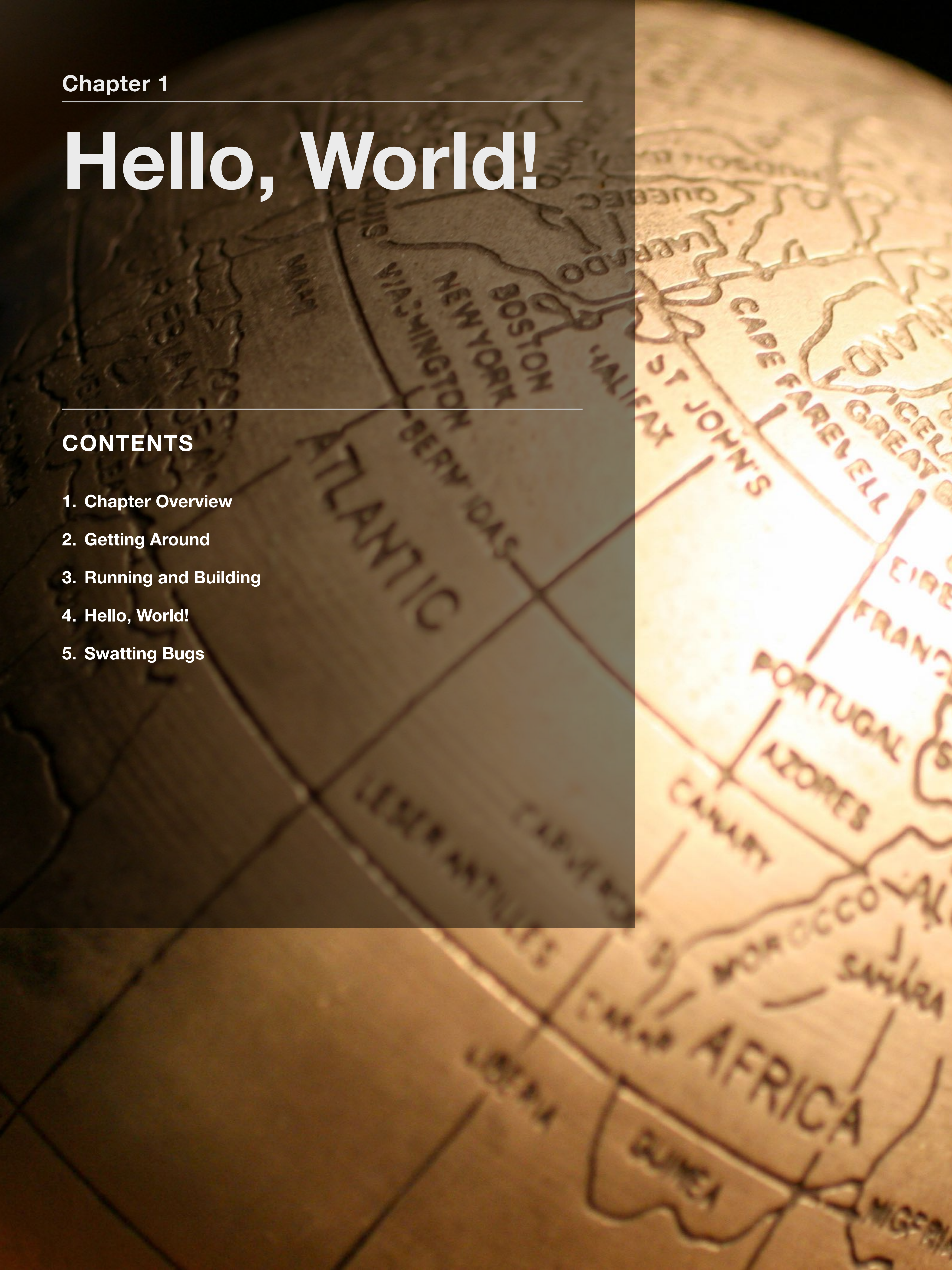
   This is a more detailed explanation of the step above. It will probably provide more details about the task you're working on.

Most of the code examples in this book are accompanied by a series of steps explaining how things work.

Finally, you may see a note. A note looks like this:

This is a note. The text in the note isn't absolutely essential, but it might provide some background information on the current topic.

Copyright & License



Chapter 1

Hello, World!

CONTENTS

1.1 Chapter Overview

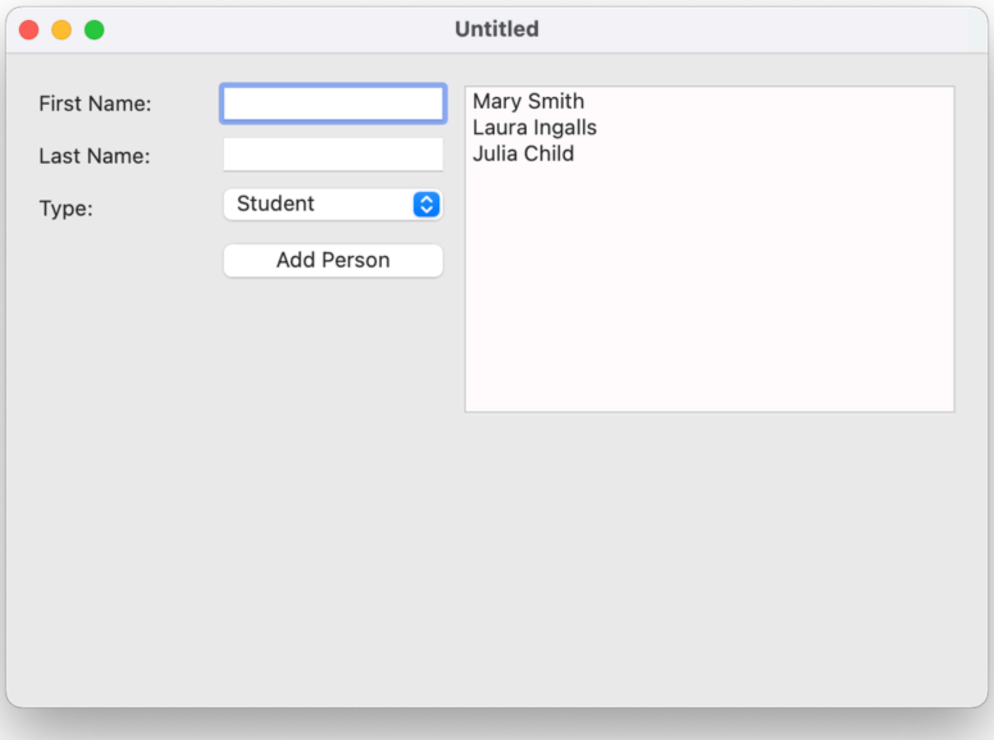
Welcome to Introduction to Programming with Xojo.

Xojo is an integrated development environment, or IDE, used to design and build other software applications. It uses a programming language which is also called Xojo. In very general terms, you as the programmer or developer enter your Xojo code into the Xojo IDE, which then compiles your code into a native app that can be run, independent of Xojo, on your computer or on someone else's.

This chapter introduces the IDE. You will learn how to navigate the IDE, how to customize it, how to organize your projects, and how to run and build your own applications. Some of the concepts introduced in this chapter may not make much sense at the moment, but they will be explained in more detail in later chapters.

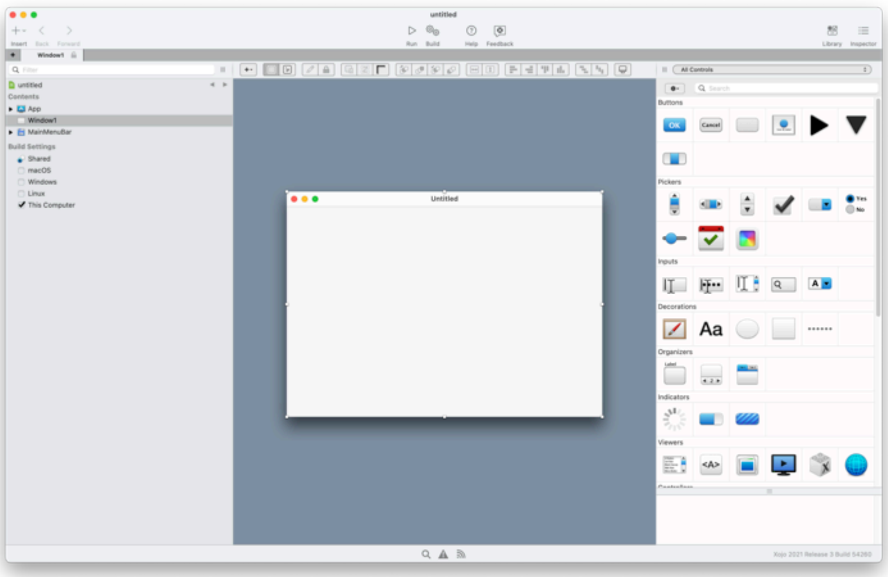
1.2 Getting Around

Begin by opening the Xojo application. Xojo launches with a Getting Started window showing some of the resources available to you as you learn Xojo. When you close this window, you will come to the Project Chooser window. This will prompt you to choose the type of new project. Select “Desktop” and press the OK button.



Xojo can build many different types of apps, including web apps, console (or command line) apps, iOS apps and apps for Raspberry Pi.

After you choose a project type, Xojo will create an empty project based on that template. The project is the file that stores all of the source code, user interface designs, and information about the app you are developing. The default empty desktop project looks like this:



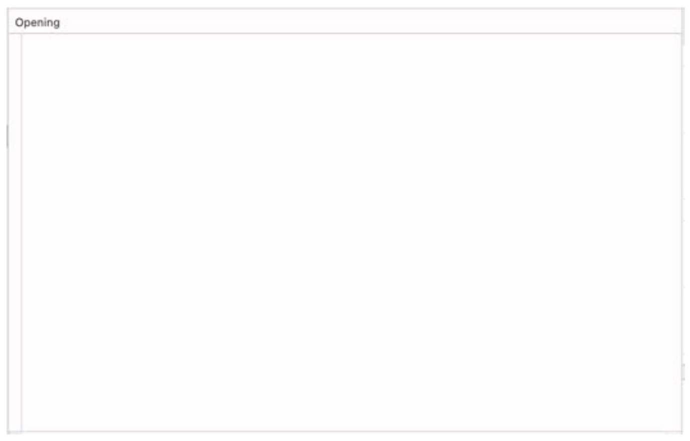
This main window is referred to as the workspace window. By default, it has three distinct areas. The larger empty area, also known as the pasteboard because you can paste things into it, holds an empty window. The window is where you can begin to design the user interface (UI) for your application. The left pane, known as the Navigator, holds a list of items in your project. The right pane is the Library, which is a list of controls that you can add to your interface (by dragging and dropping). By default, you won't see anything other than App, Window1, and MainMenuBar. Beneath the Contents you should see some options for Build Settings. Build Settings allow you to change the settings of the application you will be creating, such as its name, version number, and icon.

Immediately above the window and pasteboard is the command bar. Here is where you'll find buttons for common commands related to what is being shown. In the case of a window, you'll see an add button, alignment buttons and others.

Above the command bar at the top of the workspace window is the toolbar. The two rightmost buttons toggle the visibility of the Library and the Inspector. Remember the Library is a list of controls and the Inspector allows you to modify the properties of whatever you have selected in the pasteboard. For example, if you drag and drop a button from the Library to your interface, you can select that button and use the Inspector to change its caption or physical size.

All together, this view is called Layout View. In Layout View, you visually design the look and feel of your application. Another important view is Code View. This is where you will enter the Xojo source code that controls the behavior and functionality of your app. The easiest way to get started in Code View is to use the Insert Menu to insert an Event Handler. Events will be discussed in much greater detail in a later chapter, but for now, you need to know that events allow your application to react to actions taken by your application's end users as well as actions that the computer or operating system may cause while your application is running. Select Event Handler from the Insert Menu and choose Opening in the list of events that appears and click the OK button.

Notice how the interface changes. The pasteboard disappears and is replaced by the code editor, while the Navigator and Inspector or Library remain visible. The name of the event that you're editing will be visible at the top of the code editor.



As you add components to your projects, you will see them in the Navigator, whether you are in Layout View or Code View. You may double click on an item in this list to open it for editing. You may also drag and drop these items to arrange them in the order you desire. The order in which you arrange these items will have no bearing on the performance or functionality of your built application; it is up to you to organize your project in a way that makes sense to you. You may even add folders and subfolders if you wish to organize your project in such a way. To add a folder, go to the Insert Menu and choose Folder. You can then drag and drop other project items into the folder.

The Insert Menu is one that you will be using often as you build more and more complex applications. In addition to folders, you will use it to add classes, windows, and other components to your projects. Once again, these concepts will be explained in later chapters.

1.3 Running and Building

On the toolbar, one of the buttons that appears is the Run button. Clicking this button will tell Xojo to build a temporary copy of your project and execute it. You may also run your project by choosing Run from the Project menu. Although you have yet to add any code to your project, go ahead and run it now.

You will be presented with a blank window. While this may not seem impressive, quite a lot has already been accomplished. First, your project has been converted from a Xojo project file into an app that can be run on your computer. In addition, your app can respond to menu commands and keyboard shortcuts and react accordingly. For example, if you press Command-Q on Macintosh or Alt-F4 on Windows, your blank app will quit and you will be returned to Xojo.

Running your app in this way allows you to access the Xojo debugger, which will be covered later in this chapter. What it does not give you, however, is an app that you can share with other people. The app produced by running is temporary and is only intended to be used for testing and debugging.

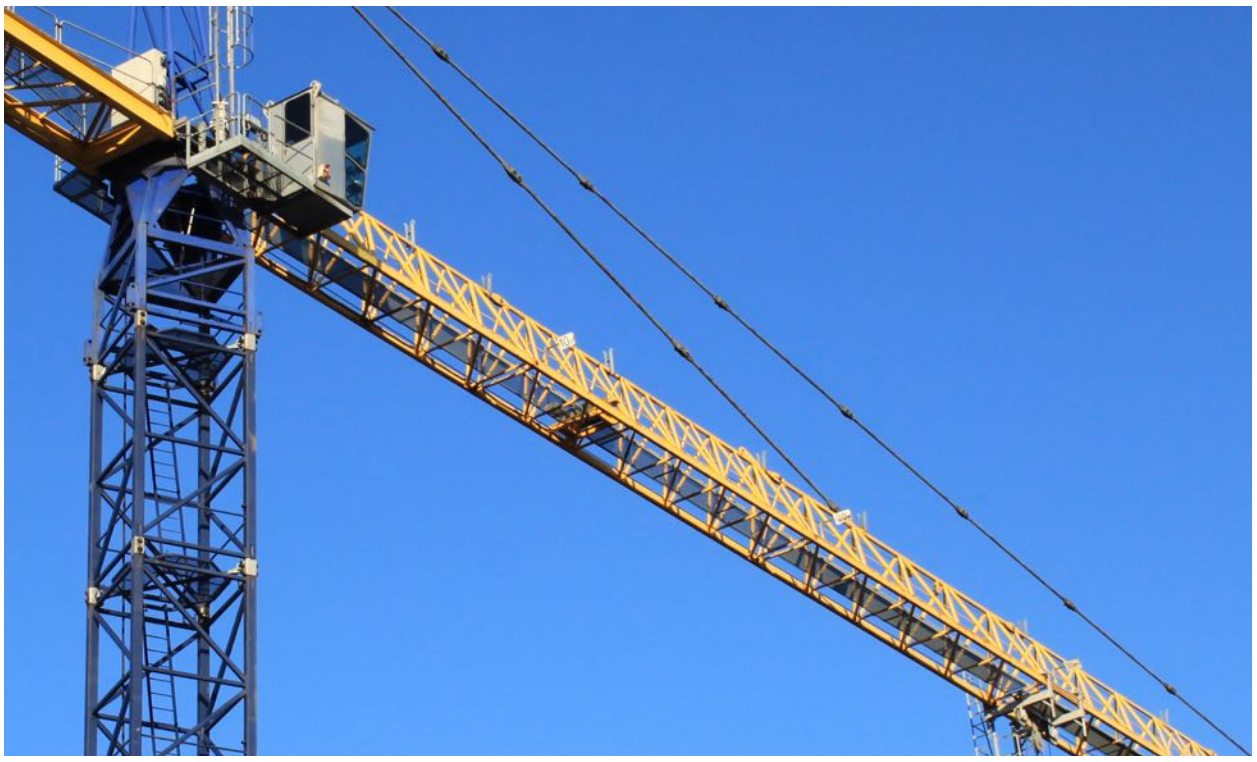
To create an app that can be shared, you need to build your application. The Build button is found directly to the right of the Run button on the toolbar, or you may choose Build Application from the Project menu.

Remember, you can run your projects with the free version of Xojo, but you must purchase a Xojo license in order to build your apps.

If you build your project now, you will have an app that can be run, but it will have a rather generic name and icon and will not do much of interest at this point.

To change that, select the App item in the Navigator and select it. This item, called a class, allows you to set properties that will apply to your app as a whole. With the App class selected, you will see the right panel change to show the Inspector, if not, click on the Inspector button in the toolbar.

The Inspector is grouped into several sets of properties. Not all of these sets will be covered at this time. Notice that there is a separate group of build settings for each platform for which your project can be built: Windows, Linux, and macOS. Find the setting appropriate for your platform and change the App Name property.



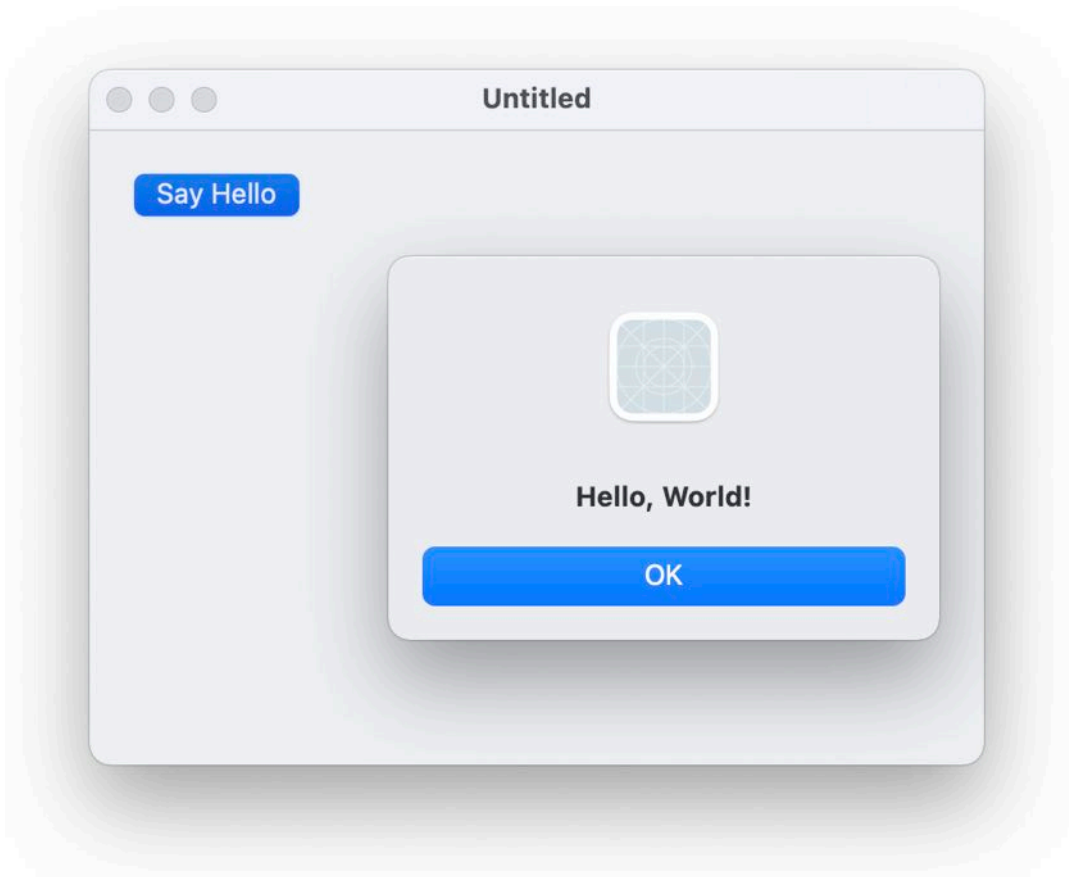
Click on Shared Settings in the Build Settings area of the Navigator. You will see more groups of settings, including one called Version. For a blank app such as this one, this is not important, but as you build more complex apps and upgrade those apps, you will need to keep those settings up to date. In general, you should increase MajorVersion when you have added major features or functionality to your application. MinorVersion should be increased for smaller features, and

BugVersion should be increased for new builds of your application that specifically fix bugs. If you were to enter 3 for the MajorVersion, 1 for the MinorVersion, and 4 for the BugVersion, your application would report itself to the operating system as version 3.1.4.

Below BugVersion is StageCode, which offers four options: Development, Alpha, Beta, and Final. As you work on adding code to your project, the StageCode should be set to Development. Once your application is feature complete, you should change the StageCode to Alpha. StageCode should be set to Beta once your application is nearly complete and most of your internal testing is complete. Before you release a build of your application to the world at large, you should set the StageCode to Final.

1.4 Hello, World!

While computer programming is far from an ancient art, it does have some traditions. One such tradition is the Hello World app. Whenever a developer is learning a new language, it is traditional to start with a very simple app that announces its existence by declaring, “Hello, World!”. You will not be breaking with said tradition, so this section will help you build your own Hello World app. Here is a screenshot of the final app.



In the Navigator, find the item called Window1. Window1 represents the default view that your app displays to the end user (this can be changed, as will be seen in later chapters). Click on Window1 to open it in the Layout Editor. This is the view that you will use to design the user interface for your projects.

As noted above, there are three main areas that you will see in Layout View. The largest area, found in the center, is called the pasteboard. Within the pasteboard, you will see an empty window. On the left side of the window is the Navigator, which contains the items in your project. On the right is either the Inspector or the Library. The Library contains interface elements that you may add to your window (or position elsewhere on the pasteboard).

1) **Find the Default Button in the controls list and drag it onto the window in the pasteboard.**

   First, turn your attention to the Library. As you scan up and down the list, you will see various interface elements, some of which should look familiar, such as the Check Box, various Buttons, and the Scroll Bar. The usefulness of other controls, such as the Canvas, the Page Panel, and the Timer, may not be immediately apparent. These controls and others will be covered in Chapter Six. For the Hello World application, the only control required is the Default Button. Hover over the various controls and you will see more details displayed in the bottom section of the Inspector.

Find the Default Button in the controls list and drag it onto the window in the Layout Editor. Dragging controls onto windows is a task that you will perform repeatedly as you build your app. Once you drop the control onto the window, you may drag it to change its position, and you may also use the Inspector properties to modify it.

2) **With the newly added Default Button selected, click the Inspector Button in the toolbar.**

There are only a few properties that you will change for your Hello World application, and in reality, they are all optional. However, you want to begin even now to develop habits that will lead to better productivity down the road. With that in mind, the first property you will edit is the first property in the list: Name.

3) **Change your PushButton's Name to "HelloButton".**

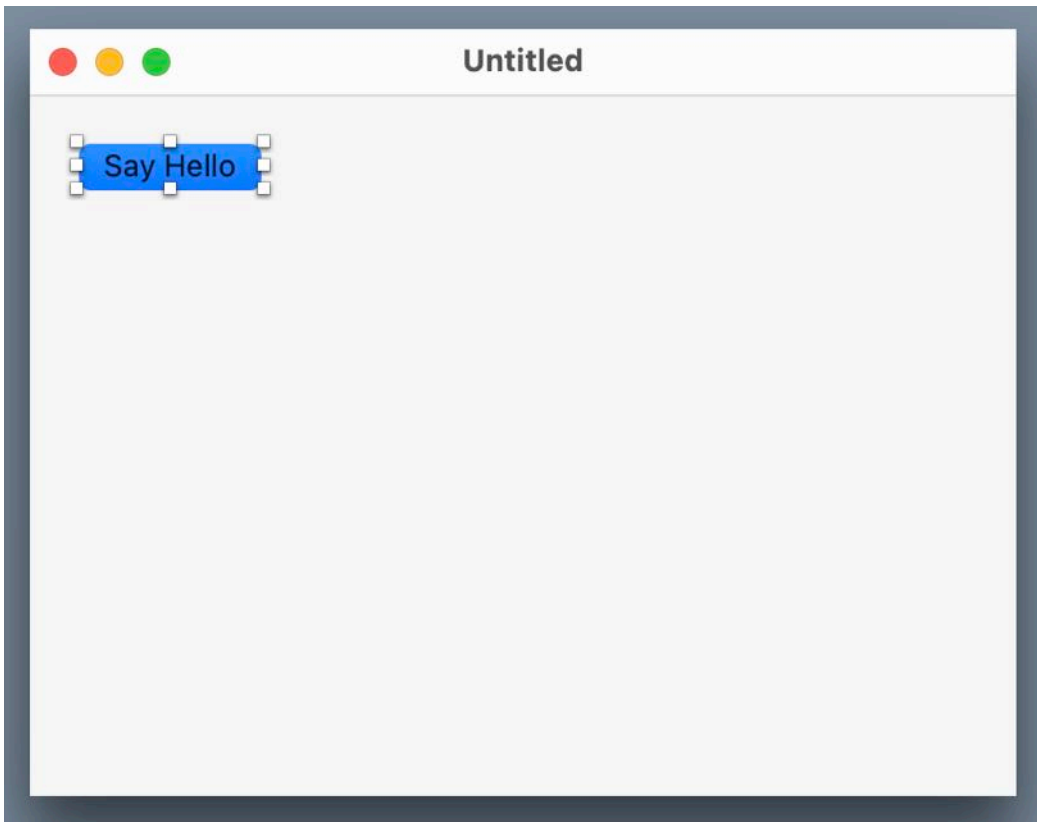
A control's name is the way you will refer to it in your code. A control's name may not include spaces or punctuation, aside from the underscore character. It is highly advisable to give each control a name that will remind you of its purpose. In this case, the name HelloButton will remind you that this button will say, "Hello." The Xojo language is not case sensitive, so if you prefer to use all lower case letters for the name, Xojo will not object.

4) **Change HelloButton's caption to "Say Hello" in the Inspector.**

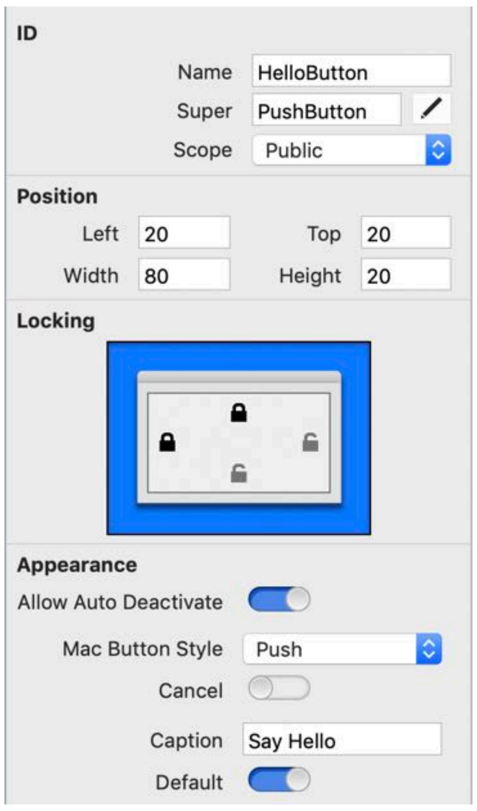
The second property you will modify is the PushButton's Caption. While the control's name is how you as the developer will refer to the control, the caption is how the end user of your application will see it. In the case of a Button, the caption will be the visible text on the button itself. Hover over the Button to find and select the Caption icon. Change HelloButton's caption to "Say Hello" in the Appearance section of the Inspector. The caption can include spaces and punctation.

5) **Change the Button's location on the window by modifying its left and top properties in the Position section of the Inspector.**

If you prefer to modify your Button's position visually, you may drag it into position manually and use the guide lines that appear to help it "snap" into position. Don't worry too much about the specific location of the button; just use your own judgement to design the window. Your window may look something like this:



And here is a closer (partial) view of the Inspector:


ID
Name: HelloButton
Super: PushButton
Scope: Public

Position
Left: 20
Top: 20
Width: 80
Height: 20

Locking

Appearance
Allow Auto Deactivate: On
Mac Button Style: Push
Cancel: Off
Caption: Say Hello
Default: On

6) **Run your project.**

Your window should appear, complete with your "Say Hello" button. Clicking the button, however, does not bring up a "Hello, World!" message. To accomplish that, you need to add some code to the Button.

7) **Quit your running app to return to Xojo.**

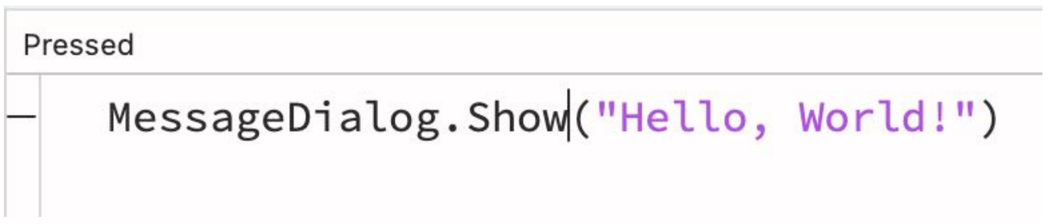
Make sure to save all projects you build throughout the book. You will be returning to and further developing various projects as you learn.

When you see Layout Editor again, double-click on HelloButton. Xojo will present you with a list of events to which you can add code. Select the Pressed event (the code in the Pressed event is run whenever the Button is pressed). You will see other events listed as well, such as KeyDown, LostFocus, Opening, and others, but for now you will only need the Pressed event (events will be explained in greater detail when controls are discussed in Chapter Six).

8) **With the Pressed event highlighted, click OK to display the Code Editor. This is the code you'll use:**

```
MessageDialog.Show("Hello, World!")
```

Your Code Editor should now look like this:


Pressed

```
MessageDialog.Show("Hello, World!")
```

As you enter code into the Code Editor, you may notice that Xojo helpfully autocompletes as you type. Xojo's autocomplete is a great way to learn more about the language, since you can begin typing a few letters to see what suggestions appear. In addition, help appears at the bottom of the window, offering information about the method or function to which the mouse is currently pointing. This is another great way to learn more about the Xojo language.

As for what you actually typed in, you entered two things: a method and a parameter. These terms will be explained in detail in Chapter Three, but for now, just know that a method in Xojo is simply a way of telling the computer to do something. A parameter for that method gives the computer additional details about what you want. In essence, the method is what you want to do, and the parameters are how to do it.

The method you're running is called Show and it belongs to the MessageDialog class. It takes a piece of text and displays it to the end user of your app. The piece of text in question in this example is "Hello, World!".

9) Run your project again.

Once again, your window should appear, complete with your "Say Hello" button. This time, clicking the button does indeed display a "Hello, World!" message. Congratulations! You've built and run your first Xojo app.

10) Quit your application.

If you are familiar with other programming languages, you may notice some differences between them and Xojo. First of all, many languages require a semicolon at the end of a line, rather than relying on white space and/or line breaks to indicate the end of a line. Xojo code should never end with a semicolon - Xojo will actually use the Syntax Help area at the bottom of the window to warn you if you type a semicolon out of habit.

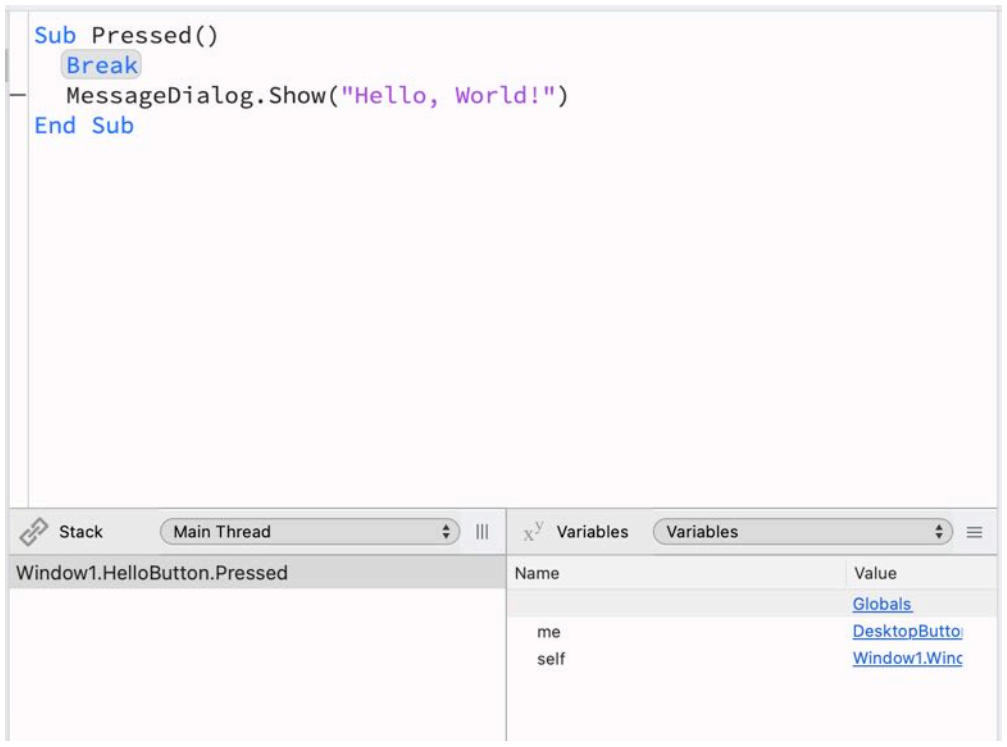
1.5 Swatting Bugs

Now that your first application is successfully running, this section will give you a very brief introduction to the Xojo debugger. Note that while the debugger is a critical part of Xojo, it will not have an entire chapter dedicated to it. Instead, as other concepts are introduced, you will gradually learn more about the debugger. For now, you will see two ways to access the debugger: one intentional way and one accidental way.

First, the intentional way: back in the Code Editor, find the place you entered the MessageDialog.Show line earlier (if your Hello World application is still running, you will need to quit or exit from it). Change the code in HelloButton's Pressed event to look like this:

```
Break
MessageDialog.Show("Hello, World!")
```

The Break keyword causes your app to pause, but not stop, and displays Xojo's debugger. With the Break keyword in place, run your project. When you click HelloButton, you will see a screen similar to the one below:



```
Sub Pressed()
  Break
  MessageDialog.Show("Hello, World!")
End Sub
```

Window1.HelloButton.Pressed

Name	Value
	Globals
me	DesktopButto
self	Window1.Winc

This is Xojo's debugger. With your current project, there is not much to see, but as your projects become more involved, the debugger can provide you with a wealth of information about your application while it is running. In the screenshot above, note that the currently executing line is highlighted (the Break keyword). The pane in the lower right portion of the window provides you with a hierarchical view of your app's variables and properties.

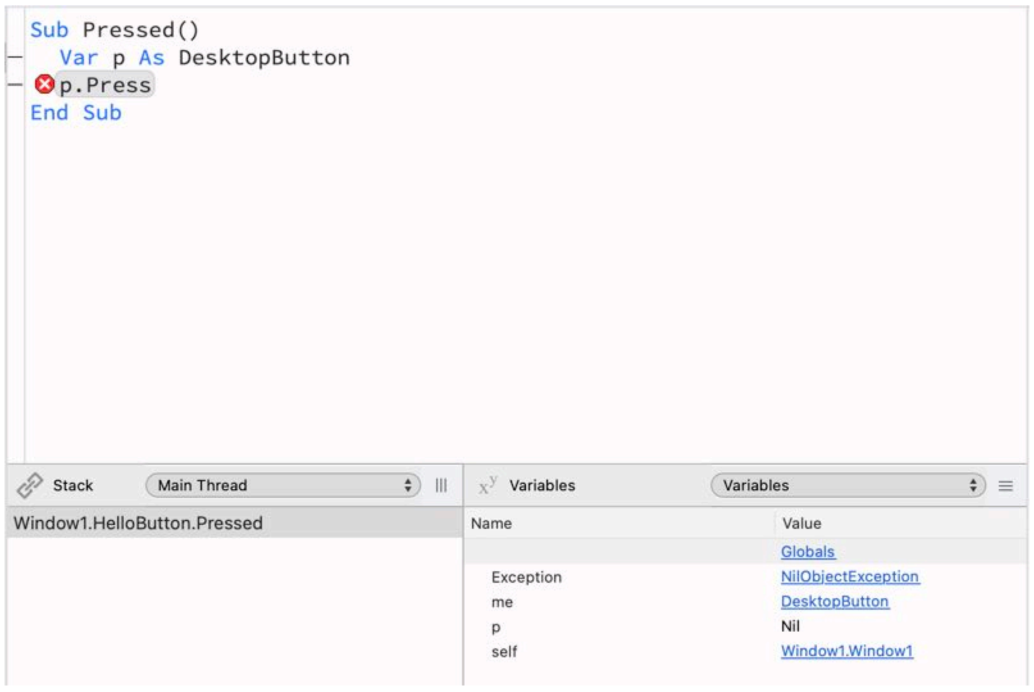
Press the Stop button on the Editor Toolbar to stop execution of your app and return to editing your project.

Now that you have seen one way to access the debugger intentionally, let's look at an accidental way. This example is, admittedly, contrived, but it should show you an important aspect of using the debugger.

Change the code in HelloButton's Pressed event to the two lines below:

```
    Var p As DesktopButton
    p.Press
```

For now, you need not worry about what the above code is even attempting to do. Simply run your project again and press HelloButton. The debugger should appear again, but with a slightly different look:



```
Sub Pressed()
  Var p As DesktopButton
  p.Press
End Sub
```

Window1.HelloButton.Pressed

Name	Value
	Globals
Exception	NilObjectException
me	DesktopButton
p	Nil
self	Window1.Window1

The red icon indicates where the error has occurred. A glance at the variables pane shows that P, the DesktopButton, is Nil. The meaning of this will become clear in later chapters, but in essence, you have attempted to access something that simply does not exist yet.

In programming terms, a non-existent object is called Nil. When Xojo encounters a Nil object, or one of many other types of exceptions, the debugger will, by default, be displayed.

This is what is called a "bug" in the code.

Bugs? You may wonder why programming errors are called bugs. It is often recounted how in the early days of computing, computers like the Mark II relied heavily on vacuum tubes which generated both heat and light, and they would attract moths, which would interfere with the computer's operation. But the notes of Thomas Edison as far back as the 1870's, long before the Mark II, used the word bug to refer to the malfunctions in machines.

In general terms, an exception occurs whenever something happens in a running app that your code does not expect. This could be a number falling outside of an expected range or, as in the example above, an attempt to access an object that does not exist. Writing defensive code to prevent exceptions is a major part of software development.

Chapter 2

Introduce Yourself

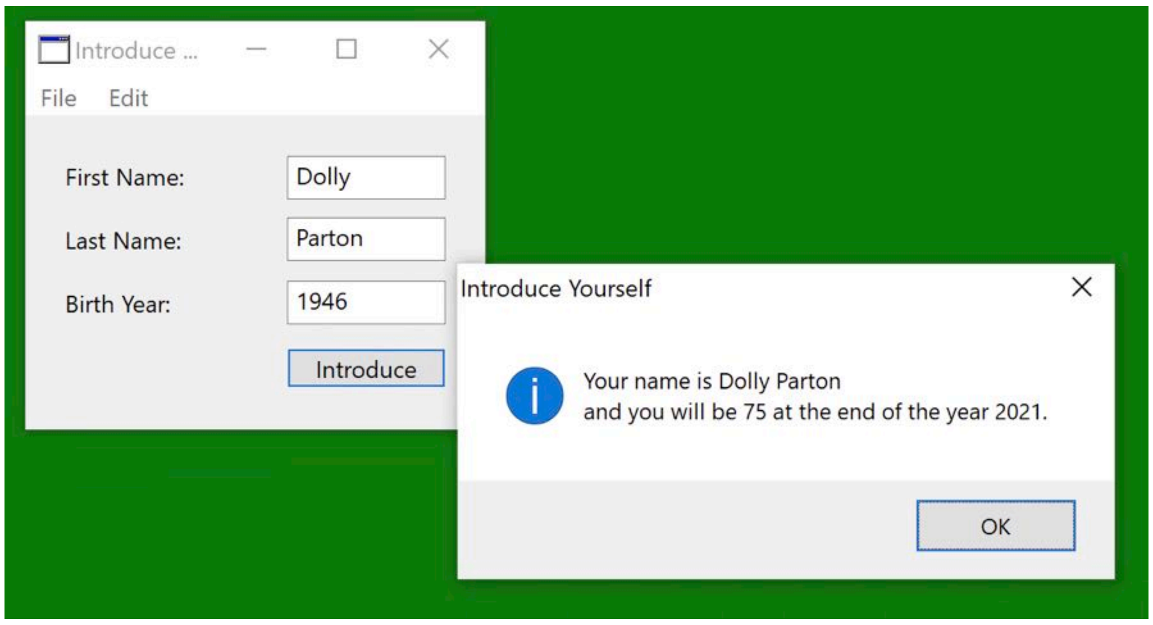
CONTENTS

2.1 Chapter Overview

This chapter will introduce the concept of variables. A variable is a name given to a location in your computer's memory that holds a value. As far as you as the developer are concerned, a variable has three properties: a name, a value, and the type of data it contains.

For example, you may have a variable named MyAge. Its type could be integer, or a whole number, and its value might be 16.

A variable's data type could be almost anything. This chapter will discuss some of the more common data types and how to use them in your code. It will also discuss some best practices for naming your variables, as well as how to assign values to them. Finally, you will build a small application called "Introduce Yourself." This application will ask the end user a few questions and provide some information back to the user.



First Name: Dolly

Last Name: Parton

Birth Year: 1946

Your name is Dolly Parton
and you will be 75 at the end of the year 2021.

2.2 A Place For Your Stuff

A variable is a way for you to refer to a location in the computer's memory that holds a value you may need to access, either to read it or change it. In order to refer to this memory location, you will begin by choosing a name for your variable. Establishing good variable naming conventions early in your programming experience will help you tremendously down the road. Naming variables can be difficult, but by keeping a few guidelines in mind, you will be able to come up with a convention of your own.

Bear in mind that these are simply conventions. Each person needs to develop a naming convention that best fits their coding style.

First, pick variable names that are specific and descriptive. For example, the sample project you will build later in this chapter will have a variable that needs to store the end user's first and last name, so that variable is called fullName. If you needed to differentiate between different people's names, you may have variables called employeeFullName and supervisorFullName. Don't worry about using long names for your variables, since Xojo's autocomplete can help with the typing later on. Avoid generic names. Working under a deadline, you may be tempted to use a single letter for a variable name. While this may be appropriate for certain counting and looping functions (as we'll see in later chapters), in general, specific names are more practical. A variable name should indicate its purpose to you at a glance.

Second, when it comes to capitalization, no single way is the right way. Some developers prefer camelCase (also known as medial capitals), some prefer names_with_underscores, and some use lowercasewithnopunctuation. Whatever you decide to use, be logical and, most importantly, be consistent. In this book, camelCase variable names will be used.

Third, some developers prefer to use the variable name to indicate its type, although this is admittedly far from universal. Examples would include nameString, birthDate, and favoriteColor. This is not part of the convention we will be following in this book, but you are free to follow it in your own code and projects.

Finally, note that as you expand your knowledge of Xojo, you will be able to apply these same guidelines and conventions to other aspects of your code, such as function names and custom classes.

Now that you have some ideas on variable naming, it's time to take a look at how you can tell Xojo about these names. To do that, you will use the Var keyword.

To use an example from above, you may have a variable called fullName that you will use to store someone's name. Its data type will be string (which will be explained in Section 2.3). To create this variable, enter this line into Xojo:

```
    Var fullName As String
```

This simple line accomplishes quite a lot. As was explained above, it reserves space in the computer's memory for the information you want to store. It also sets aside a specific, memorable name that by which you can refer to it. Finally, it tells Xojo what type of data you'll be storing (in the above example, a string, or text data).

"Var", as you may have already guessed, is short for "Variable."

Telling Xojo which data type you will be using is critical. Xojo is a *strongly typed language*. This means that every variable has a certain type, and that you as the developer are expected to treat each variable as its type warrants. For example, you may perform mathematical operations on numerical data, but not on text. Xojo will offer suggestions and warnings if you attempt to use a variable that is not supported by its data type.

Most data types need to be instantiated. While using the Var keyword sets aside space for your variable, the New keyword instantiates, or creates an instance of, your variable. Consider the following lines:

```
    Var holiday As DateTime
```

With that line, you have set aside space in memory for your variable, you have named it (holiday), and you have told Xojo what kind of data you will be working with (a date and time value). What you have not done, however, is create the DateTime object itself. So if you attempt to access one of holiday's properties, such as its Year property, before continuing, you will encounter a NilObjectException such as the one demonstrated at the end of Chapter One.

To instantiate your variable, use the New keyword:

```
    holiday = New DateTime(2020, 5, 25)
```

Once your variable is instantiated, you may freely access or modify its data and properties. A more complete example of creating a variable and accessing its properties follows:

```
Var holiday As DateTime
holiday = New DateTime(2020, 5, 25)
MessageDialog.Show(holiday.Year.ToString)
```

In this brief example, you have created your variable, you have instantiated it, and you have shown the end user a message box containing the current year.

2.3 Common Data Types

You may have noticed "data types" being mentioned quite a bit in the past few pages, but what is a data type? Simply put, a data type is a form of information that behaves in a certain way and has certain characteristics. In this section, you will learn about some very common data types. Most programming languages use the same basic set of data types.

STRINGS

One of the most common data types you will encounter is the string. A string is simply a piece of text. It can be of any length (the maximum size of a string is limited only by the computer's memory) and can contain any data that can be represented by letters (of any language), numbers, and punctuation. To create a string in Xojo, simply use the Var keyword:

```
    Var myName As String
```

Strings are one of a small set of data types that do not need to be instantiated. As soon as the line of code containing the Var keyword is executed, the string is created in memory, although it is empty.

Whenever you enter string data into Xojo, it must be surrounded by double quotes:

```
    myName = "Elvis Presley"
```

If you have a string that already contains double quotes, you must “escape” the quotes by doubling them:

```
    myName = "Elvis ""The King"" Presley"
```

A doubled double quote may look odd, but the extra double quote tells Xojo that it is part of your string data and does not denote the end of your string.

Strings can sometimes be a source of confusion for new developers. Because they may contain any textual data, their data can sometimes look like data of another type. Consider the following:

```
    Var myAge As String
    myAge = "18"
```

If you look at the value of the myAge string, it appears to us as humans to be numerical data: 18. However, to the computer (and the Xojo compiler), it is not. It is simply a series of bytes that represent a 1 followed by an 8. Before using this data, which appears to be numerical, as part of a mathematical function, you need to perform an intermediate step. To retrieve its value as a number that the computer will recognize as such, use the ToInteger function:

```
    Var myAge As String
    Var myNumericAge As Integer
    myAge = "18"
    myNumericAge = myAge.ToInteger
```

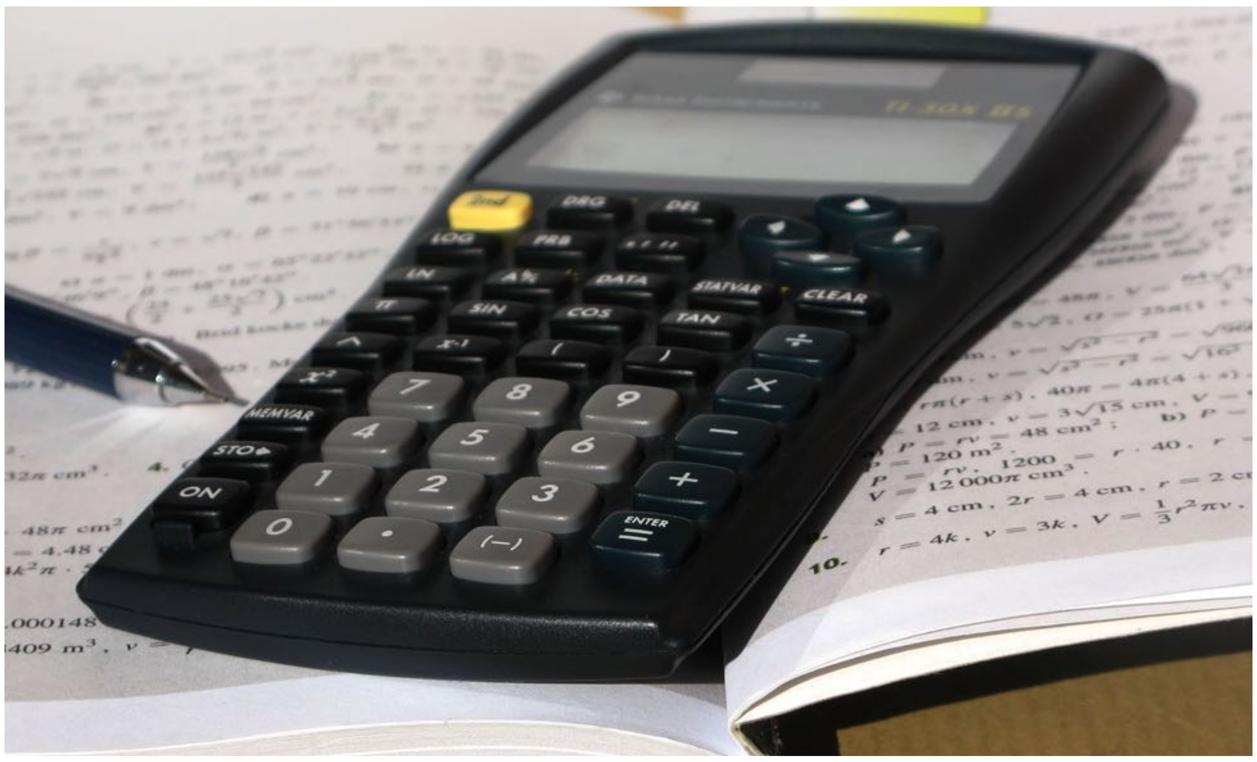
ToInteger is method built-in to strings that converts the string value to an integer and returns it. What is an integer? An Integer is a whole number.

INTEGERS AND DOUBLES

An integer is a positive or negative whole number, so it contains no precision beyond its decimal point. Examples of integers are -3, 0, 42 or 999.

The maximum and minimum values of an integer depend on the computer. On a 32-bit system, an integer can be as low as -2,147,483,648 and as high as 2,147,483,647. That’s a range of 4,294,967,295 possible values.

Most modern computers are 64-bit. A 64-bit computer can hold an integer as low as -2^63 and as high as one below 2^63.



There are several ways to declare an integer variable. The first and most simple way:

```
    Var myNumber As Integer
```

If you are on a 32-bit computer, that will give you a 32-bit integer. Appropriately enough, if you are on a 64-bit computer, that will give you a 64-bit integer.

If you wish to be more specific about the type of integer you need, you may do so:

```
    Var myHugeNumber As Int32
    Var myReallyHugeNumber As Int64
```

The two lines above will give you one 32-bit integer and one 64-bit integer. Unless you have a specific need to do otherwise, the simplest course of action is to declare your variables as Integer.

As noted above, an integer may not be as precise as you need it to be. Since integers are limited to whole numbers, you may not use them to store more precise numbers, such as 3.14. Trying to store 3.14 as an integer will result in a value of 3.

To store a more precise number, you may use the Double data type. A Double is a floating point number, also known as a double precision number. A Double may have a decimal point, and may have any number of significant digits beyond the decimal point.

The range of a double precision number is quite large. Its range is so large that you will probably never need to worry about exceeding it.

Since both integers and doubles are numeric data types, you may use them in mathematical operations without issue:

```
    Var oneNumber As Integer
    Var anotherNumber As Integer
    Var theResult As Integer
    oneNumber = 5
    anotherNumber = 10
    theResult = oneNumber + anotherNumber
```

In the code above, theResult has a value of 15.

The same can be done with doubles, of course:

```
    Var oneNumber As Double
    Var anotherNumber As Double
    Var theResult As Double
    oneNumber = 5.3
    anotherNumber = 10.2
    theResult = oneNumber + anotherNumber
```

In this code, theResult would have a value of 15.5.

In both examples above, you actually did some unnecessary typing. Notice how the three variables were declared on three separate lines. This is perfectly acceptable, but there is a shorter way:

```
    Var oneNumber, anotherNumber, theResult As Double
```

Which method you should use depends on your own preferences and on which way seems more readable to you.

Integers and doubles, like strings, do not need to be instantiated. When you declare them, they immediately exist with a value of zero.

BOOLEANS

A boolean is a data type that is used to store a truth value. Its value may be true or false. To create a boolean, use the Var keyword:

```
    Var thisIsAwesome As Boolean
```

As with strings, integers, and doubles, booleans do not need to be instantiated. Once declared, by default, a boolean will be false until set otherwise. So in the code above, you have a boolean called thisIsAwesome, whose value is false.

However, this is not accurate. Because programming is, in fact, awesome, that value needs to be changed:

```
    Var thisIsAwesome As Boolean
    thisIsAwesome = True
```

If, later on, you need to change a boolean back to false, you may do so:

```
    thisIsAwesome = False
```

DATES AND TIMES

A DateTime object can store all relevant details about a particular date and time. To create a DateTime object, use the Var keyword and instantiate the variable (unlike the data types that have been covered so far, a date does need to be instantiated, either with the New keyword or by using the Now function).

However, when you instantiate your DateTime, you must tell it what date and time to start off with. The easiest way to do this is with the Now function that is part of DateTime:

```
     Var today As DateTime = DateTime.Now
```

Each “element” of the current date and time can be accessed using its properties. However, most cannot be modified. This is due to the fact that you could easily create a date that does not exist by setting the Month to 13 for example. Instead, when you create a DateTime, you pass it the year, month and day to which you wish the date to be set:

```
    Var someDay As New DateTime(1944, 6, 6)
```

You can pass in a specific time as well. In this example, we've added 10AM:

```
    Var dDay As New DateTime(1944, 6, 6, 10, 0, 0)
```

These values are then stored in various properties that you can retrieve:

```
    Var today As DateTime = DateTime.Now
    Var thisMonth As Integer
    thisMonth = today.Month
```

With this code, you now have an integer called thisMonth, which contains a numeric representation of the current month (January = 1, February = 2, etc.).

A DateTime object has other properties that may be read but not modified (in programming terms, you may "get" them but not "set" them). DayOfWeek is an integer that represents the day's position in the week (1 = Sunday, 7 = Saturday). DayOfYear is, similarly, an integer that represents the day's position within the year (a date object representing February 1 would have a DayOfYear value of 32). There is also a WeekOfYear property.

To view the DateTime as a string, use the DateTime's ToString function which returns the DateTime in string form. You can even choose various formats in which to display the date and time. How the DateTime appears as a string will vary based on the settings and locale of the end user's computer.

COLORS

It may be odd to think of a color as a variable, but Xojo does. To create a color variable:

```
    Var myColor As Color
```

As with strings, integers, and doubles, colors do not need to be instantiated. When created, a color defaults to black.

You may change a color in your code or by asking the user to select a color. You will see both methods.

To change your color in code, you must understand a few basics about how Xojo stores colors. A color has three properties that will be discussed here: Red, Green, and Blue. Each of these is an integer that may be anywhere from zero to 255. The higher the value, the more that shade is present in the color. For reference, the color black would have zero for each property, while white would have 255 for each property. Pure red would have 255 for Red, zero for Green, and zero for Blue. Purple would have 255 for Red, zero for Green, and 255 for Blue.

To set these properties in code, you may simply enter a value for each, using the RGB method:

```
Var thisIsWhite As Color
thisIsWhite = RGB(255, 255, 255)
Var thisIsPurple As Color
thisIsPurple = RGB(255, 0, 255)
```

Or you may want to ask your end user to choose a color, using the Color.SelectedFromDialog function:

```
Var myFavoriteColor As Color
If Color.SelectedFromDialog(myFavoriteColor, "Pick A Color:") Then
  // We have the user’s color now
End If
```

It can be helpful to add notes to your code to remind you of how something works or why you chose to code it the way you did. This is called "commenting." To comment your code, start a line with either // or a ' character followed by your comments. This tells Xojo that it can ignore this and not attempt to treat it as code.

The Color.SelectedFromDialog function takes two parameters. The first is a color variable, and the second is a string that will be presented to the user (in the code, "Pick A Color:"). The function returns a boolean (true or false) depending on whether the user picked a color or somehow cancelled the selection. That is why the second line of the code above starts with "If" - the outcome is based on the end user's actions, which you cannot control or know in advance. All of the code that happens between the "If" line and the "End If" line will be executed if the user picked a color. If the user cancelled, nothing will happen. For now, you can leave the middle section of that code blank, since you'll be learning about conditionals and logic in Chapter Three.

If you are familiar with HTML or CSS, or if you have experience working with hexadecimal colors (where red, green, and blue run from 00 to FF), you may also set a color using "&c" (known as a literal):

```
Var ThisIsBlue As Color
ThisIsBlue = &c0000FF
```

If you are not familiar with hexadecimal math, you may want to stick with the RGB method to get started.

2.4 Putting Variables To Use

Now that you know the basics of creating variables, it's time to learn about getting, setting, and comparing their values.

For the most part, to set a variable's value, use the equal sign. Depending on the data type, you may or may not need to use quotation marks. They are required for string data, but must be omitted for numeric data types and booleans.

```
    Var meaningOfLife As Integer
    Var chapterTitle As String
    Var thisIsEasy As Boolean
    meaningOfLife = 42
    chapterTitle = "Introduce Yourself"
    thisIsEasy = True
```

As mentioned above, datetime and colors are a different story. A date's value cannot be set directly (unless you have another datetime object already in existence, in which case you still have the issue of the original datetime needing a value to begin with). To set a datetime's value, you use something called a *constructor*):

A constructor is a special method that runs when a new object is instantiated. If you know the date values, you assign them when you instantiate the date with this syntax:

```
    Var nineEleven As DateTime(2001, 9, 11)
```

The date's constructor can take up to seven parameters. In the above example, you only used three: year, month, and day. The year must be specified, but the month and day may be left out, in which case they are assumed to be 1; everything else that is left unspecified is assumed to be zero. The parameters are, in order, year as integer, month as integer, day as integer, hour as integer, minute as integer, second as integer, nanosecond as integer and timeZone as timezone (the timezone can safely ignored for now).

Colors may be set by assigning each of the color's RGB properties:

```
    Var logoBackgroundColor As Color
    logoBackgroundColor = Color.RGB(255, 0, 255)
```

Of course, programming is typically more complicated than simply assigning values as you have seen in these examples so far. Quite often, performing a calculation is involved. For numeric data, Xojo supports the common mathematical operations that you would expect. Addition is performed with the + (plus sign) operator, subtraction is performed with the - (minus sign) operator, and multiplication is performed with the * (asterisk) operator. If you have done any mathematical work on a computer, these should be familiar to you. Here are some code examples:

```
    Var unitCost, quantity, totalCost As Double
    unitCost = 25
    quantity = 4
    totalCost = unitCost * quantity
    // totalCost is now 100
```

A piece of code that executes a formula or function as seen above is called an expression. Expressions may also be algebraic and use the same variable multiple times:

```
    Var theAnswer As Integer
    theAnswer = 25 + 35
    // theAnswer is now 60
    theAnswer = theAnswer + 10
    // theAnswer is now 70
```

Division is slightly more complicated. There are three operators related to division: / (forward slash), \ (backslash), and Mod. The most commonly used is the forward slash, which is used for what is known as floating point division:

```
    Var exactAnswer As Double
    exactAnswer = 5 / 2
    // exactAnswer is now 2.5
```

With the backslash, Xojo performs integer division, which does not account for fractional values:

```
    Var roundedAnswer As Integer
    roundedAnswer = 5 \ 2
    // roundedAnswer is now 2
```

Finally, Mod is used to calculate the remainder of a division operation:

```
    Var leftoverValue As Double
    leftoverValue = 5 Mod 2
    // leftoverValue is now 1,
    // since 5 divided by 2 is 2 with a remainder of 1
```

Mod can also be useful for determining whether an integer is even or odd. If your integer Mod 2 is equal to 1, the number is odd. If the answer is zero, the number is even.

The plus operator may also be used on strings to concatenate (combine) them:

```
    Var presidentName As String
    presidentName = "Abraham" + " " + "Lincoln"
    // presidentName is now "Abraham Lincoln"
```

In the example above, note the space that is added between the first and last name. Omitting this space is a common error in string concatenation.

As with mathematical operations, the plus operator can use expressions that refer to the same variable multiple times:

```
    Var presidentName As String
    presidentName = "Abraham" + " " + "Lincoln"
    // presidentName is now "Abraham Lincoln"
    presidentName = "President" + " " + presidentName
    // presidentName is now "President Abraham Lincoln"
```

Getting a variable's value is typically an equally simple matter. When debugging, an easy way to display a variable's value is to use the MessageDialog.Show method that you used in Chapter One. Since the MessageDialog.Show takes a string as its only required parameter, displaying the value of a string is trivially easy:

```
    Var theGreeting As String
    theGreeting = "Hello!"
    MessageDialog.Show(theGreeting)
```

Numeric variables are slightly trickier, since the MessageDialog.Show method cannot take an integer or double directly. Xojo provides several ways to convert numeric data to strings but the easiest way is with the ToString function.

To use it, call it from your integer or double variable, and it displays the string value of that number:

```
    Var myAgeAsANumber As Integer
    Var myAgeAsAString As String
    myAgeAsANumber = 16
    myAgeAsAString = myAgeAsANumber.ToString
    MessageDialag.Show(myAgeAsAString)
```

The code above stores the value of the integer in a string and then displays that string to the user in a message dialog.

Unless you specify otherwise, your numeric value will be unformatted. For example, a large number may be displayed in scientific notation, or a decimal value might display more places than you'd like. In such cases, a format can be passed to the ToString function.

A format specification is a string that describes how the number should be displayed. For example, if you had the value .25 and wanted to display it as a percentage, you would use “#%” as the format specification:

```
    Var myPercent As Double
    myPercent = 0.25
    MessageDialog.Show(myPercent.ToString("#%"))
```

At first glance, that may look like nonsense, but there are just a few simple rules for the format specification. First, the pound (or hash) sign (#) represents the number you wish to format. In the example above, the percentage sign (%) tells Xojo to display the number as a percentage, so that the number is multiplied by 100 and displayed with the percentage sign following it.
The chart below lists some of the possibilities for the format specification.

Character	Description
#	Placeholder that displays the digit from the value if it is present. If fewer placeholder characters are used than in the passed number, then the result is rounded.
0	Placeholder that displays the digit from the value if it is present. If no digit is present, 0 (zero) is displayed in its place.
.	Placeholder for the position of the decimal point.
,	Placeholder that indicates that the number should be formatted with thousands separators.
%	Displays the number multiplied by 100 followed by the % character.
+	Displays the plus sign to the left of the number if the number is positive or a minus sign if the number is negative.
-	Displays a minus sign to the left of the number if the number is negative. There is no effect for positive numbers.
E or e	Displays the number in scientific notation.

The next chart contains some examples of those format specifications in practice. One quick note: a format specification can be made up of one or three parts, separated by semicolons. The first part is the format specification for positive numbers, the second part is for negative numbers, and the third part is for zero. If only one format is specified, it will be used for all numbers.

Here is a code example:

```
    Var stickerPrice As Double
    stickerPrice = 24995.9
    MessageDialog.Show(stickerPrice.ToString("###,###.00"))
```

This would display “24,995.90” to the end user. Other sample formats are shown in the table.

Format	Number	Formatted String
#.##	1.786	1.79
#.0000	1.3	1.3000
0000	5	0005
#%	0.25	25%
###,###.##	145678.5	145,678.5
#.##e	145678.5	1.46e+05
-#.##	-3.7	-3.7
+#.##	3.7	+3.7

So far you have seen how to convert numbers into strings for display, but you can also convert strings into numbers to use in calculations. As mentioned earlier in the chapter, the best way to accomplish this is by using the ToInteger (or ToDouble) function:

```
Var myAge As String
Var myNumericAge As Integer
Var myAgeInTenYears As Integer
myAge = "18"
myNumericAge = myAge.ToInteger
myAgeInTenYears = myNumericAge + 10
// myAgeInTenYears is now 28
```

Now that you have learned some things about variables, as well as getting and setting their values, it’s time to build this chapter’s sample project, “Introduce Yourself.”

2.5 Hands On With Variables

If you have not already done so, launch Xojo and create a new desktop project.

1) **Click on Window1 in the Navigator.**

The Layout View for Window1 should appear.

In Chapter One, you used the Inspector to set the properties of a button on a window. You may also use the Inspector to set the properties of the window itself. With no objects on the window selected, the Inspector will modify the window's properties. You are going to set three of the window's properties: Title, Width, and Height.

2) **Set Window1's Title to "Introduce Yourself".**

Note that the window's Title and Name properties are different. The title is what appears to your end user at the top of the window in your running application. The name is how you as the developer refer to it in your code. In a more complex application, it is always recommended to give your windows meaningful titles, but for your purposes in the project, setting just the title will be sufficient.

3) **From the Library, drag a Label onto Window1.**

4) **Switch to the Inspector and set the Label's Text property to "First Name:" and its name to "FirstNameLabel".**

5) **Switch to the Library and drag a TextField onto Window1.**

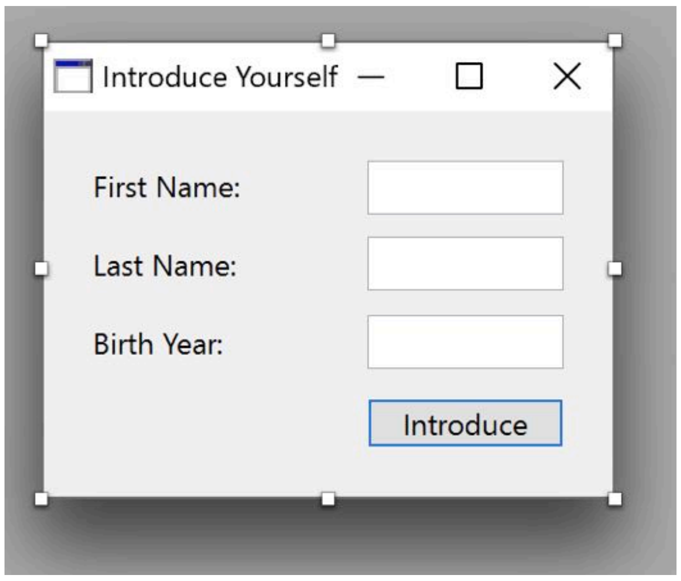
6) **Switch to the Inspector and set the TextField's Name property to "FirstNameField".**

7) **Add two more Labels, two more TextFields, and one Button to the window, using the properties in the table below.**

Control	Name	Text	Caption
Label	LastNameLabel	Last Name:	N/A
Text Field	LastNameField	N/A	N/A
Label	BirthYearLabel	Birth Year:	N/A
Text Field	BirthYearField	N/A	N/A
Default Button	IntroduceButton	N/A	Introduce

8) **Position and size your controls using your own creativity.**

When all of your controls have been added to Window1, your window may be similar to this screenshot:



Now that your interface is complete, you will need to write some code to make your application do something. To add code, double click on IntroduceButton and choose the Pressed event to display the Code Editor.

At the beginning of the chapter, you learned what this application will do. As a reminder, it will calculate the end user's full name and their age at the end of the current year and then display this information to the user. You will need four variables to make this happen.

1) In the Code Editor, add the following variables:

```
Var fullName As String
Var currentAge As Integer
Var today As DateTime = DateTime.Now
Var theMessage As String
```

FullName is the string you will use to store the user's concatenated first and last names. CurrentAge will store the user's age in years. Today will hold the current date (note that the code above has already instantiated it, so it already contains data about the current date). Finally, theMessage is a string that will put everything together to display to the end user.

2) Enter this line of code:

```
fullName = FirstNameField.Text + " " + LastNameField.Text
```

This code concatenates your user's first and last names. To do this, you will access the Value properties of your TextFields. The Value property holds whatever is visible in the TextField. It is accessed using what is known as dot notation, which you have already used when getting and settings properties for dates and colors. Dot notation is a way of accessing properties by using the object's name, followed by a dot, followed in turn by the name of the property, such as FirstNameField.Value. This should be familiar to you from looking at string concatenation earlier in this chapter.

3) To determine the user's current age in years, enter this code:

```
currentAge = Today.Year - BirthYearField.Text.ToInteger
```

Note: for the purposes of this project, you will ignore the month of birth and simply calculate the age in years at the end of the current year; taking months into consideration is not difficult, but it involves some skills that have not yet been covered. You will retrieve the data entered by the user into BirthYearField. Remember, however, that BirthYearField.Value will give you a string, which you will need to convert into numeric data, using ToInteger. You will subtract that number from the current year to determine the user's current age in years.

4) Add this line of code:

```
theMessage =  "Your name is " + fullName + EndOfLine
```

You now have all of the data that you need to display to the end user. The next step is to assemble it into a message to display to the end user. You will again do this through string concatenation. To make your message more pleasing to the eye, you will also add a line break after the user's full name. This is accomplished using the EndOfLine class.

For historical reasons, macOS and UNIX use a different line ending than Windows. To account for this, Xojo's EndOfLine class will automatically use the correct line ending for the current platform, although a platform-specific line ending can also be used when necessary. For the most part, simply use EndOfLine and you should be fine.

5) Enter this code:

```
theMessage = theMessage + "and you will be "
theMessage = theMessage + currentAge.ToString
theMessage = theMessage + " at the end of the year "
theMessage = theMessage + today.Year.ToString + "."
```

At this point, theMessage now contains "Your name is ", followed by the user's full name, which you have already calculated, followed by a line break. These lines of code will build the rest of theMessage, including the user's age.

6) Display theMessage to the end user using this line:

```
MessageBox(theMessage)
```

Altogether, your code should look like this:

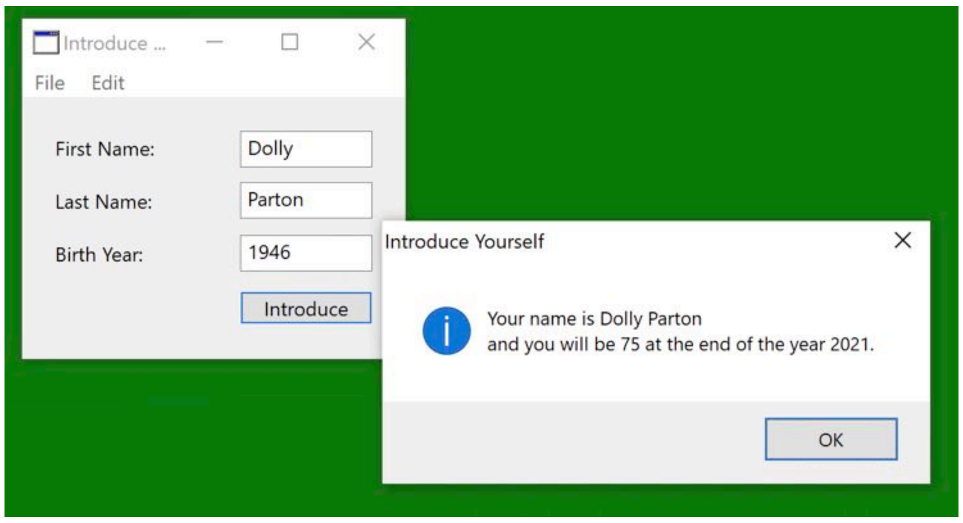
```
Var fullName As String
Var currentAge As Integer
Var today As DateTime = DateTime.Now
Var theMessage As String
fullName = FirstNameField.Text + " " + LastNameField.Text
currentAge = today.Year - BirthYearField.Text.ToInteger
```

```
    theMessage =  "Your name is " + fullName + EndOfLine
    theMessage = theMessage + "and you will be "
    theMessage = theMessage + currentAge.ToString
    theMessage = theMessage + " at the end of the year "
    theMessage = theMessage + today.Year.ToString + "."
    MessageBox(theMessage)
```

7) **Run your project.**

8) **Fill in the form and press IntroduceButton.**

You should see something like this:



First Name: Dolly
Last Name: Parton
Birth Year: 1946

Your name is Dolly Parton
and you will be 75 at the end of the year 2021.

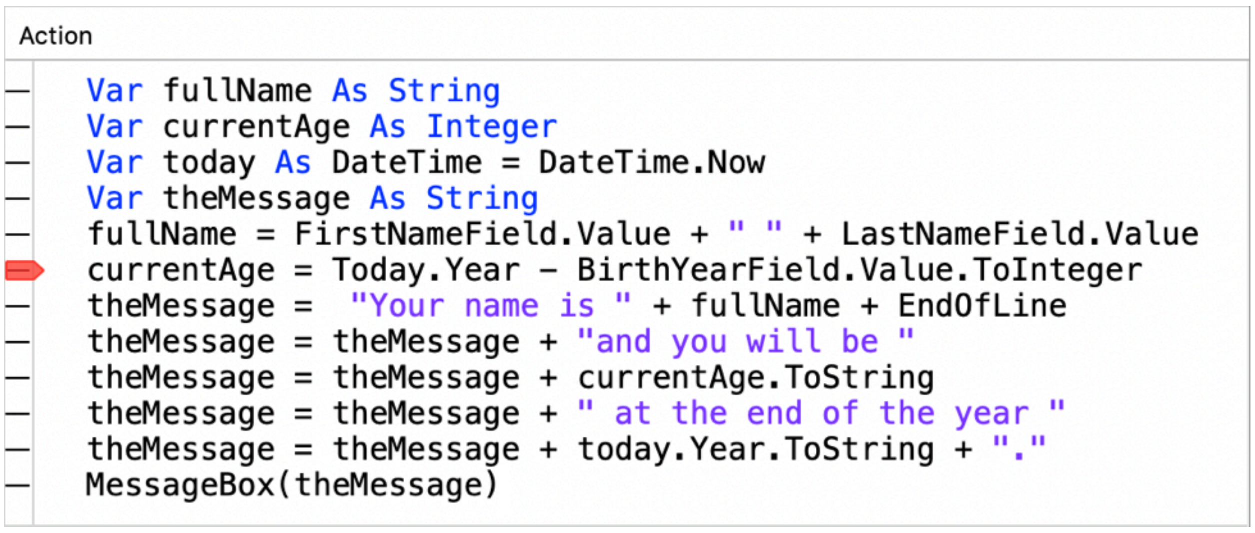
9) **Quit your application.**

2.6 Advanced Bug Swatting

In Chapter One, you got a brief look at Xojo's debugger, using the Break keyword. With the current "Introduce Yourself" project, this is a good time to take a slightly deeper look into breakpoints.

A breakpoint is a place in your code where you want Xojo to pause executing your code and display debugger.

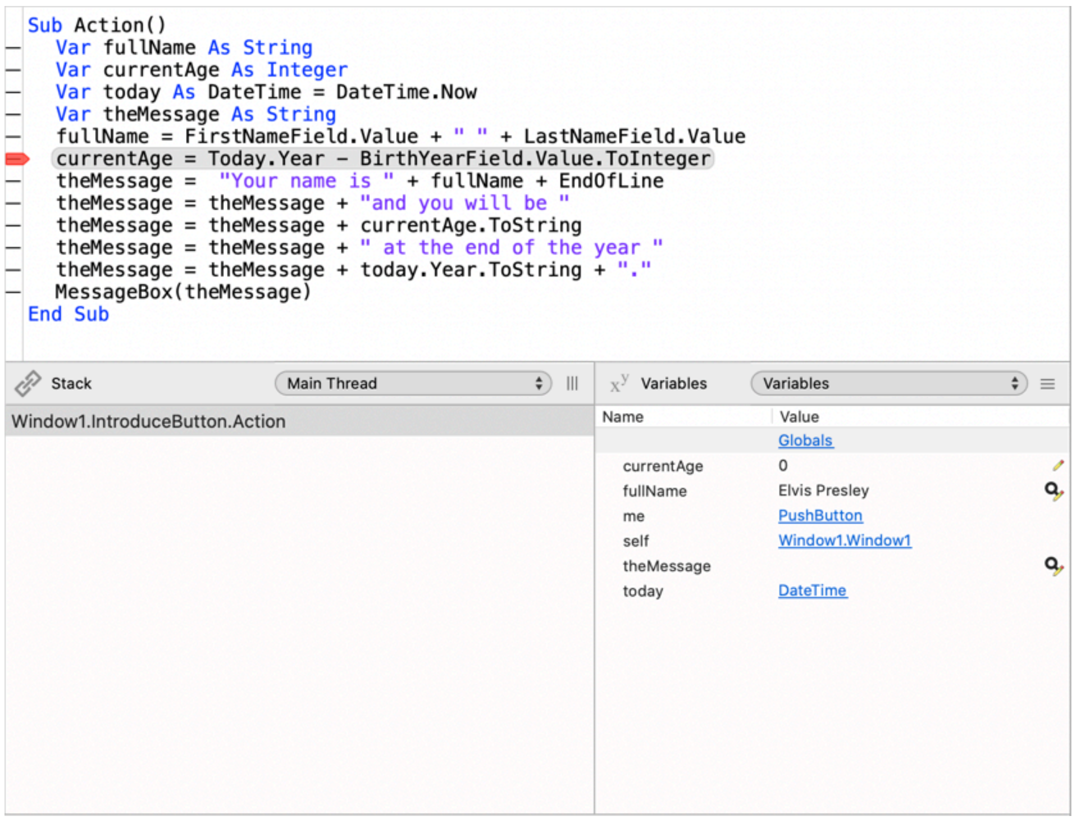
To add a breakpoint, locate the gray hyphen in the left margin next to a line of code and click on it. It will change to a red dot. Add a breakpoint to the CurrentAge line in your current code:



Action

```
Var fullName As String
Var currentAge As Integer
Var today As DateTime = DateTime.Now
Var theMessage As String
fullName = FirstNameField.Value + " " + LastNameField.Value
currentAge = Today.Year - BirthYearField.Value.ToInteger
theMessage =  "Your name is " + fullName + EndOfLine
theMessage = theMessage + "and you will be "
theMessage = theMessage + currentAge.ToString
theMessage = theMessage + " at the end of the year "
theMessage = theMessage + today.Year.ToString + "."
MessageBox(theMessage)
```

This tells Xojo to pause running your application before the CurrentAge line is executed. To demonstrate, run your project now and fill in your name and birth year. When you press IntroduceButton, you will see the debugger, paused at the line of code with the breakpoint:

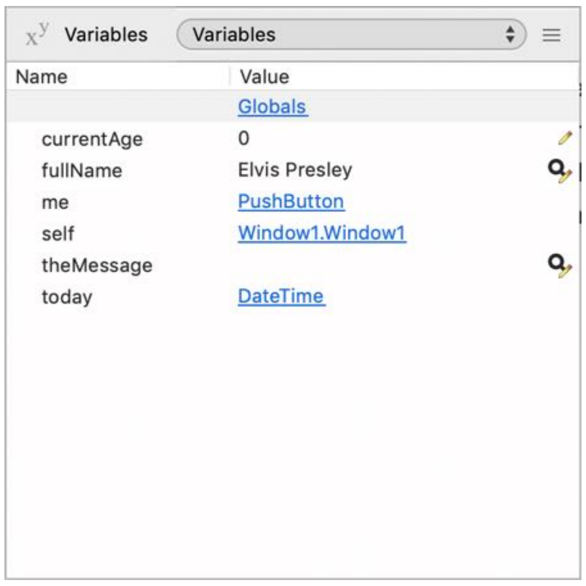


```
Sub Action()
  Var fullName As String
  Var currentAge As Integer
  Var today As DateTime = DateTime.Now
  Var theMessage As String
  fullName = FirstNameField.Value + " " + LastNameField.Value
  currentAge = Today.Year - BirthYearField.Value.ToInteger
  theMessage =  "Your name is " + fullName + EndOfLine
  theMessage = theMessage + "and you will be "
  theMessage = theMessage + currentAge.ToString
  theMessage = theMessage + " at the end of the year "
  theMessage = theMessage + today.Year.ToString + "."
  MessageBox(theMessage)
End Sub
```

Window1.IntroduceButton.Action

Name	Value
	Globals
currentAge	0
fullName	Elvis Presley
me	PushButton
self	Window1.Window1
theMessage	
today	DateTime

As you saw in Chapter One, the current line of code is highlighted in gray and the bottom right pane shows a list of variables. Here's a closer view of the variables:



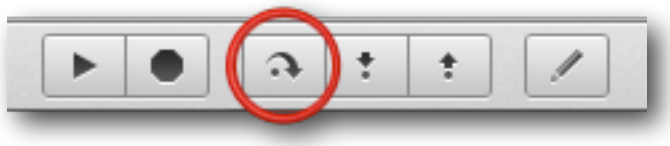
Name	Value
	Globals
currentAge	0
fullName	Elvis Presley
me	PushButton
self	Window1.Window1
theMessage	
today	DateTime

The string called FullName is already assembled, but the integer CurrentAge is zero. That's because this line of code:

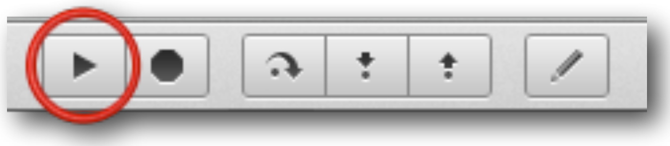
```
    currentAge = today.YearBirthYearField.Value.ToInteger
```

has not yet been executed. When you set breakpoints, be careful about the location, or you may see unexpected results.

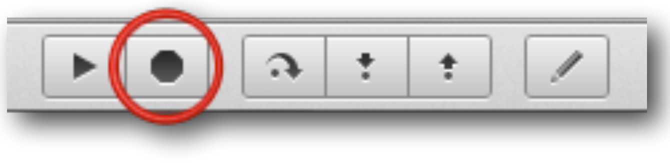
In the debugger toolbar, locate the Step button.



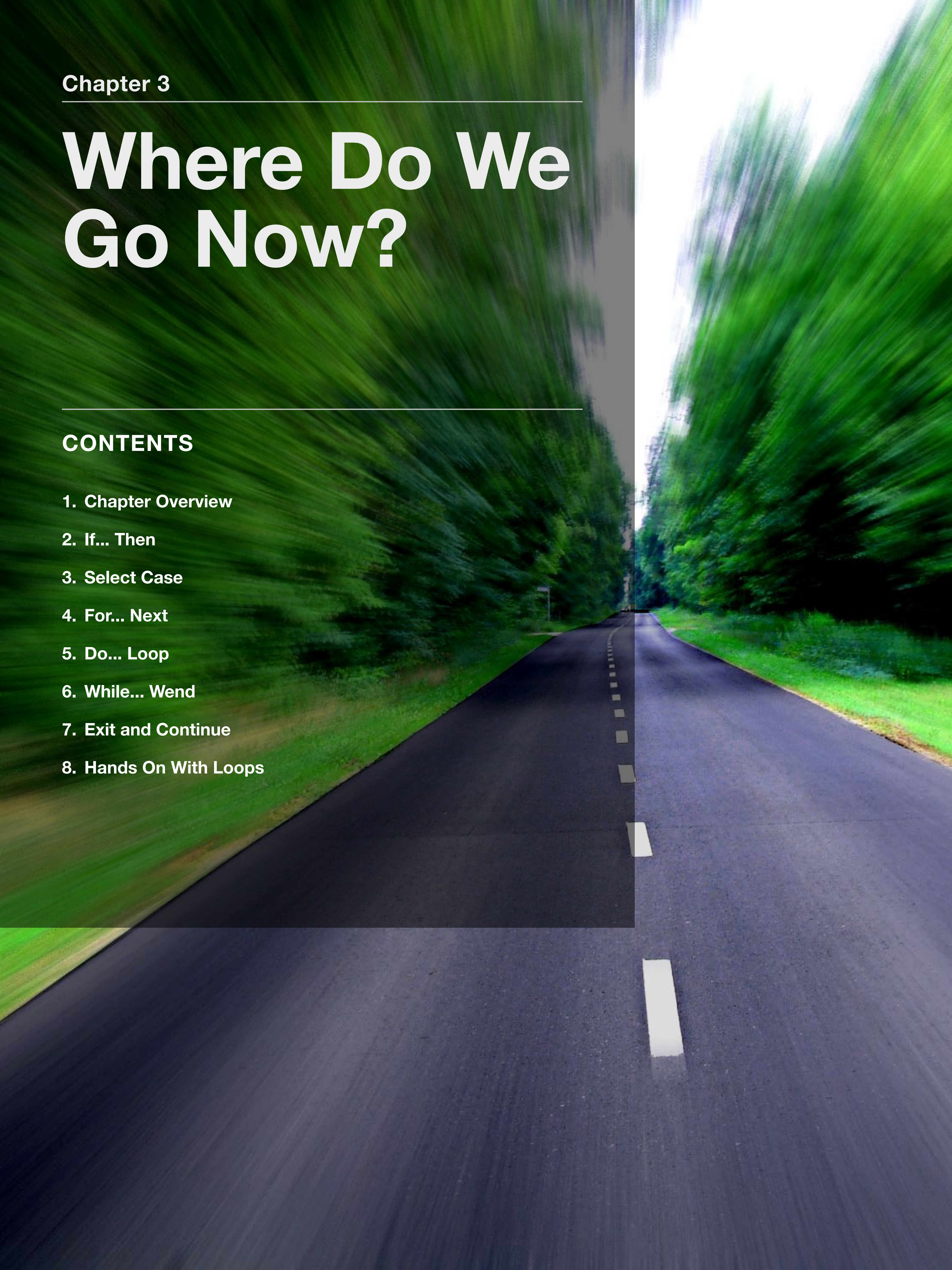
Click on that button to step to the next line of code. The next line will now be highlighted in gray and the variables pane will change to show the calculated value of CurrentAge. You may continue to press the Step button to walk through your code. If you are ready to return to your running application, press the Resume button in the debugger toolbar.



If something has gone wrong enough that you need to stop debugging (and thus the app) to return to your code, press the Stop button.



When developing a complex application, it is not uncommon to find yourself with dozens of breakpoints scattered throughout your code. If you wish to remove them all at once, go to the Project menu and choose Breakpoint->Clear All.



Chapter 3

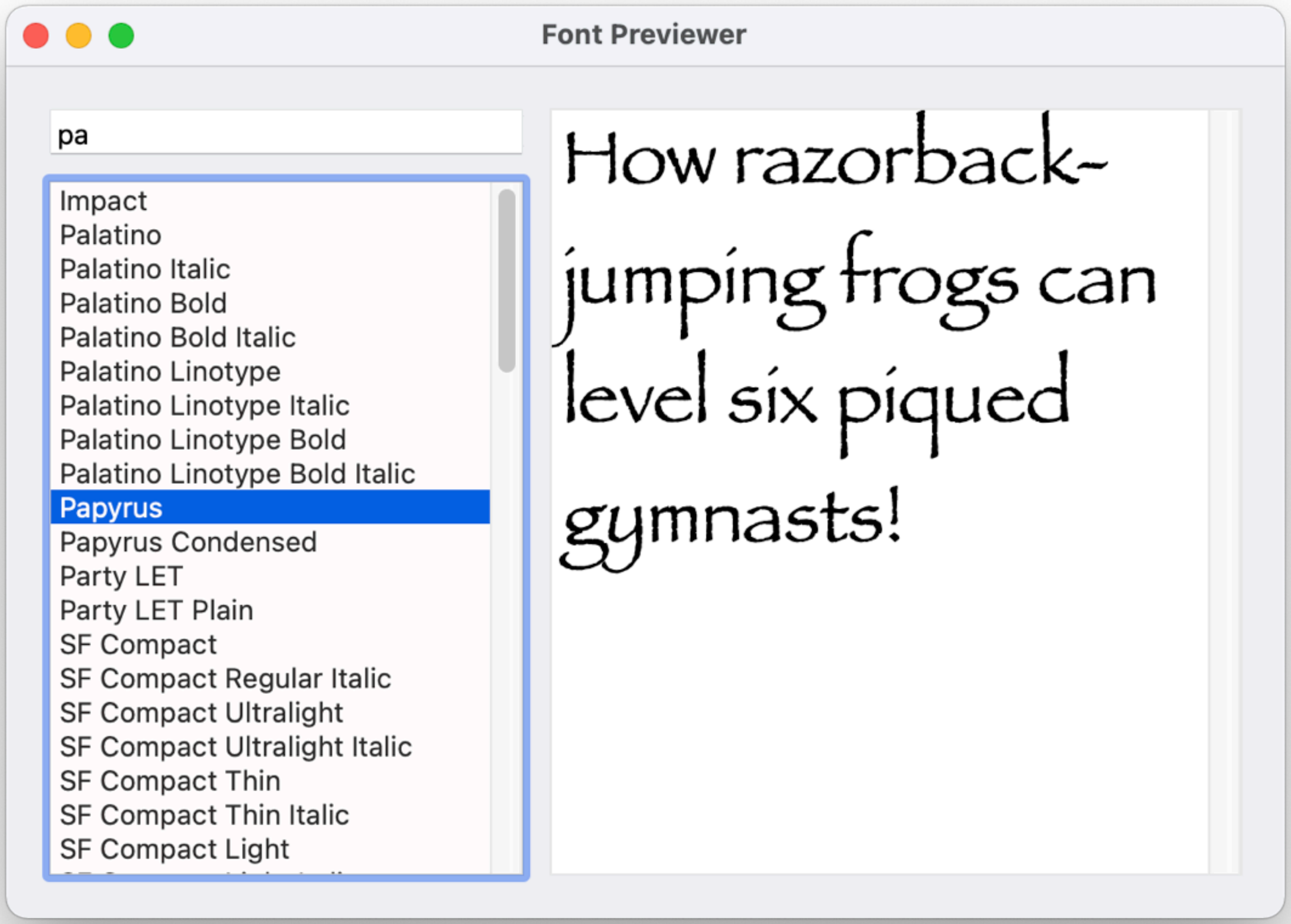
Where Do We Go Now?

CONTENTS

3.1 Chapter Overview

So far in this book, you have seen straightforward situations of getting and setting values. In any reasonably complex application, you will need to exercise quite a bit of logic. In this chapter, you will learn about checking and comparing variables, responding appropriately in your code, and controlling the flow of your applications.

For this chapter's project, you will build a custom font previewer. By default, it will list all fonts installed on the end user's computer. When a font is selected, your application will show a preview of that font. Your application will also allow us to search for specific fonts as well. A screenshot is provided below.



How razorback-jumping frogs can level six piqued gymnasts!

3.2 If...Then

The most common logic problem you will encounter as a developer is checking whether one value matches another value. For example, you may need to check if a boolean is true or false, or you may need to check if an integer is greater than or equal to 100. You may even need to check whether a string contains another string. This type of logic is performed using If.

Most programming languages, such as C#, Swift, C, C++, PHP, Java, feature the If keyword. Some languages use a slightly different syntax, but the end result is the same. Once you learn how to use If...Then in Xojo, you will be on your way to using it in any other language.

To use an example from above, imagine that you needed to check whether a boolean was true or false:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn = True Then
  // React here to the fact that the lights are on
End If
```

What does theLightsAreOn mean? theLightsAreOn is a boolean variable, which you learned about in the previous chapter. Its value may come from somewhere else in your code, or it may come from the user checking a checkbox or clicking on a radio button.

Note that the If statement follows a certain structure. The line always begins with If, followed by the expression to evaluate, followed by Then. *The next section of code is always automatically indented by Xojo*, and it will execute only if the expression evaluates to true. Following that code is the final line: End If.

Note that if you are checking the value of a boolean, you can omit the " = True" or " = False" part of the expression:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn Then
  // React here to the fact that the lights are on
End If
```

If you have worked with any C-based languages in the past, you may notice something interesting about Xojo. In most C-based languages, you must use a double equal sign to compare data (If MyVar1 == MyVar2), whereas here you are using only one equal sign. Xojo overloads its equal sign operator so that it can be used for comparison or for setting values. In fact, if you enter "==" into Xojo, it will issue a warning.

In the above example, you only tell Xojo what to do if theLightsAreOn is true. There are many times when you need to react accordingly if a value is false. To do so, use the Else keyword:

```
Var theLightsAreOn As Boolean
theLightsAreOn = True
If theLightsAreOn Then
  // React here to the fact that the lights are ON
Else
  // React here to the fact that the lights are OFF
End If
```

You are, of course, not limited to checking only boolean values. If you needed to check an integer, you may certainly do so:

```
Var myAge As Integer
myAge = 16
If myAge > 17 Then
  MessageBox("You can vote.")
Else
  MessageBox("You cannot vote yet.")
End If
```

In the above example, you are using the greater than symbol instead of the equal sign. When using greater than or less than, remember that they are not inclusive of the number you are comparing against. In other words, in this example, if myAge were equal to 17, the message box would read, "You cannot vote yet."

If you want your comparisons to be inclusive, you may use operators for "greater than or equal to" and "less than or equal to" in your expressions:

```
If myAge >= 17 Then
  MessageBox("You can vote.")
Else
  MessageBox("You cannot vote yet.")
End If
```

You may use any data type with If, but it must always result in a boolean expression. Here is an example using strings:

```
Var myString As String
myString = "Hello"
If myString = "Hello" Then
  MessageBox("We have a match!")
Else
  MessageBox("We do not have a match!")
End If
```

Xojo’s string comparisons are not case-sensitive. Because of this, the above example could be rewritten as below with identical functionality:

```
Var myString As String
myString = "Hello"
If myString = "hello" Then
  MessageBox("We have a match!")
Else
  MessageBox("We do not have a match!")
End If
```

In a case-insensitive system, a lower case “a” will match an upper case “A” - the characters are considered to be equal. If you need to perform a case-sensitive string comparison, you may use the String.Compare function.

Often, you’ll need to determine whether one string contains another string. For example, you may need to check if some data entered by the user contains a particular keyword. This can be accomplished with the IndexOf function which (like Compare) are methods of String themselves.

IndexOf takes one to four parameters and returns an integer. The first parameter is an integer, and it is optional. It indicates the position within the string to be searched that Xojo should begin searching. It defaults to zero, and will be zero if unspecified. The second parameter is the string to search for. The value returned by IndexOf is an integer that indicates where in the source string the found string occurs. If it is not found within the source string, IndexOf will return zero. Any non-zero result indicates that the string has been found.

```
Var alphabet, part As String
Var location As Integer
alphabet = "abcdefghijklmnopqrstuvwxyz"
part = "def"
location = alphabet.IndexOf(part)
// The part was found and location = 4
part = "foo"
location = alphabet.IndexOf(part)
// The part was not found and location = 0
```

For example, assume that you have asked the end user for a short paragraph about his or her background and that you need to issue an error message if the user includes the word “ninja” anywhere in his or her bio. Your application’s interface would likely contain a TextField called BioField, where the user would be expected to enter his or her biographical data.

```
If BioField.Value.IndexOf("ninja") > 0 Then
  // Issue a ninja alert!
Else
  // This is a ninja-free zone
End If
```

If you remember back to the Introduce Yourself project that you built in Chapter Two, you may recall that you asked your end user to enter his or her birth year into a TextField. You then took that value and converted it to an integer. What you did not do, however, is verify that the data could, in fact, be numeric. Xojo has a function called IsNumeric that will check to see if a string’s value can be converted to numeric data. Its usage is fairly simple:

```
Var numberText As String
numberText = "123"
If IsNumeric(numberText) Then
  MessageBox(numberText + " can be converted")
  // This is the message box that will appear
Else
  MessageBox(numberText + " cannot be converted")
End If
numberText = "Marbles"
If IsNumeric(numberText) Then
```

```
      MessageBox(numberText + " can be converted")
    Else
      MessageBox(numberText + " cannot be converted")
      // This is the message box that will appear
    End If
```

IsNumeric is not limited to simple integers. It will correctly interpret floating point numbers and even scientific notation.

If you need to compare DateTime values, the best way is to compare their SecondsFrom1970 properties:

```
    If today.SecondsFrom1970 > yesterday.SecondsFrom1970 Then
      // Time is moving forward
    End If
```

You may use any other date properties for comparison as well, such as year, month, day, etc. Or if you need to check a date to see if it is part of a given year:

```
    If today.Year = 2012 Then
      // The world may have ended.
      // Be cautious.
    End If
```

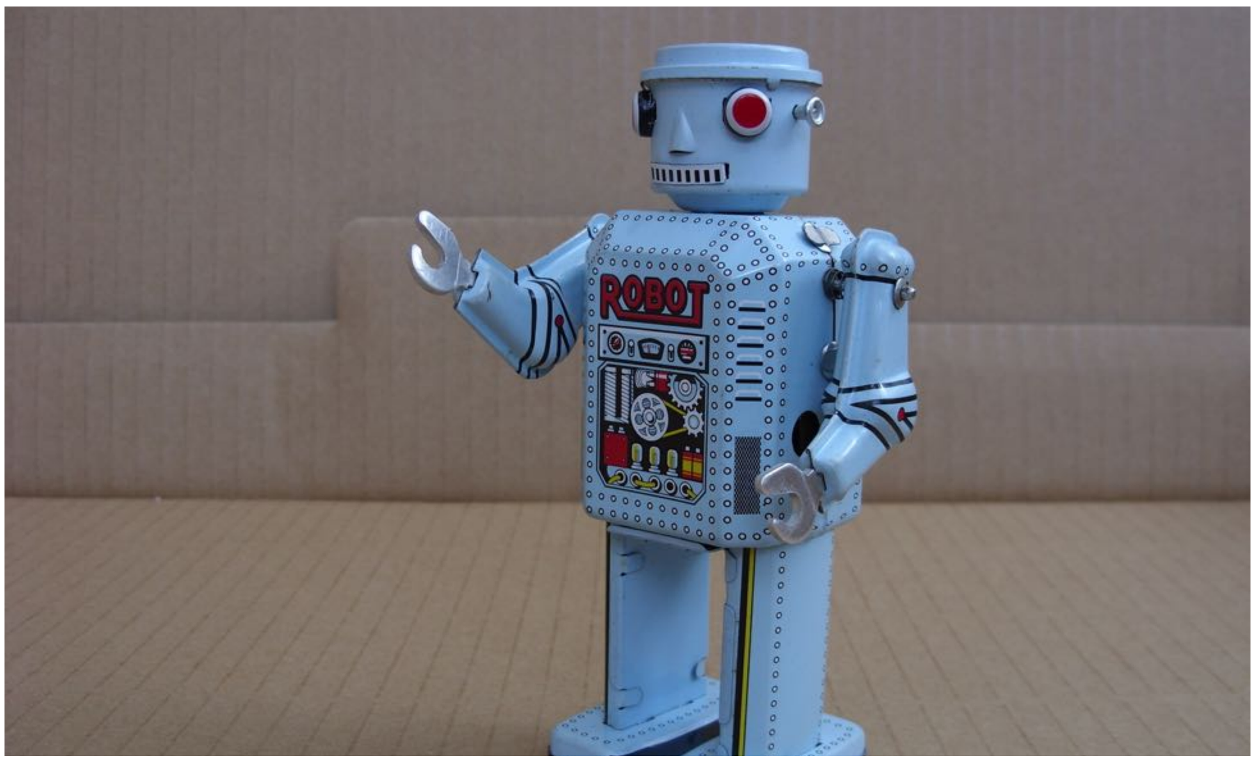
What's all this about the end of the world? Some people believed that an ancient Mayan prophecy predicted that the world would end in December of 2012. Obviously, it didn't.

You may also check for more than one possibility:

```
    If today.Year = 9999 Or today.Year = 10000 Then
      // Be aware of potential Y10K problems.
    End If
```

The code above uses "Or" to separate your conditions, so the expression evaluates to true if either condition is true. You may also use "And" to separate conditions, in which case both conditions will need to be true; if only one is true, the statement evaluates to false.

```
    If today.Year = 2012 And today.Month = 12 Then
      // The world may be ending really soon. Be
      // cautious.
    End If
```



There will be times when you will need to check for multiple conditions and react accordingly. For example, a few paragraphs above, you checked for ninjas. What if your code also needed to be aware of pirates, zombies, and robots? In between the If and the End If, you may add an ElseIf. Or you may add several ElseIf statements:

```
If BioField.Value.IndexOf("ninja") > 0 Then
  // Issue a ninja alert!
ElseIf BioField.Value.IndexOf("robot") > 0 Then
  // Issue a robot alert!
ElseIf BioField.Value.IndexOf("pirate") > 0 Then
  // Issue a pirate alert!
ElseIf BioField.Value.IndexOf("zombie") > 0 Then
  // Issue a zombie alert!
Else
  // We are safe from perceived threats
End If
```

Note that Xojo will jump to the end of the If statement as soon as one of the conditions is met. In other words, if the code above encounters a robot, the user's bio will not be checked for references to pirates and zombies. If you need to have a separate check for each, you must use a separate If statement for each.

The ElseIf statement adds quite a bit of flexibility to Xojo's logic. But once you have added more than a few ElseIfs, your code can become unwieldy fairly quickly. Fortunately, there is an easier way.

3.3 Select Case

Xojo’s Select Case statement provides you as the developer with a cleaner way to check a variable or expression for multiple values. It allows you to specify each condition you wish to check, using the Case statement, as well as the code that should execute when a matching condition is found. Xojo stores the current month as a number between 1 and 12, so the following code can tell you the name of the current month:

```
Var today As DateTime = DateTime.Now
Select Case today.Month
Case 1
  MessageBox("It's January")
Case 2
  MessageBox("It's February")
Case 3
  MessageBox("It's March")
Case 4
  MessageBox("It's April")
Case 5
  MessageBox("It's May")
Case 6
  MessageBox("It's June")
Case 7
  MessageBox("It's July")
Case 8
  MessageBox("It's August")
Case 9
  MessageBox("It's September")
Case 10
  MessageBox("It's October")
Case 11
  MessageBox("It's November")
Case 12
  MessageBox("It's December")
Else
  MessageBox("Unable to determine current month")
End Select
```

You may also specify a default action by using Else, as seen in the above example. Note that in a well written application, whatever action is specified under Else should never happen, but your code should be prepared to handle it either way.

Each Case statement may have a range of values as well. You may specify a range by providing a comma separate list or using the To keyword. Here is an example using both methods:

```
Var today As DateTime = DateTime.Now
Select Case today.Month
Case 1, 2, 3
  MessageBox("It's Q1")
Case 4, 5, 6
  MessageBox("It's Q2")
Case 7 To 9
  MessageBox("It's Q3")
Case 10 To 12
  MessageBox("It's Q4")
Else
  MessageBox("Unable to determine current quarter")
End Select
```

Some other common programming languages use "switch" instead of "select case" for such comparisons. Aside from that minor word change, the functionality is similar.

3.4 For...Next

Any time that you need to perform a similar operation on a series of variables, a For...Next loop is an easy way to do it. In its simplest form, a For...Next loop simply steps through a list and executes whatever code you specify. Every For...Next loop needs a variable to use as a counter.

Create a new desktop project in Xojo. In the Layout View of Window1, double click on the window to open its Code Editor. Locate Window's1 Opening event and enter this code:

```
Var counter As Integer
For counter = 1 To 10
  MessageBox(counter.ToString)
Next
```

Next, run your project. You should see 10 message boxes, for number 1 through number 10. After dismissing all 10 message boxes, quit the application.

You are not limited to using the counter alone. Change your code to the following:

```
Var counter As Integer
For counter = 1 To 10
  MessageBox(Str(counter * 2))
Next
```

Run your project again to see the difference.

Your counter can also count down instead of up:

```
Var counter As Integer
For counter = 10 DownTo 1
  MessageBox(counter.ToString)
Next
```

Your counter may also skip numbers, using the Step keyword:

```
Var counter As Integer
For counter = 1 To 10 Step 2
  MessageBox(counter.ToString)
Next
```

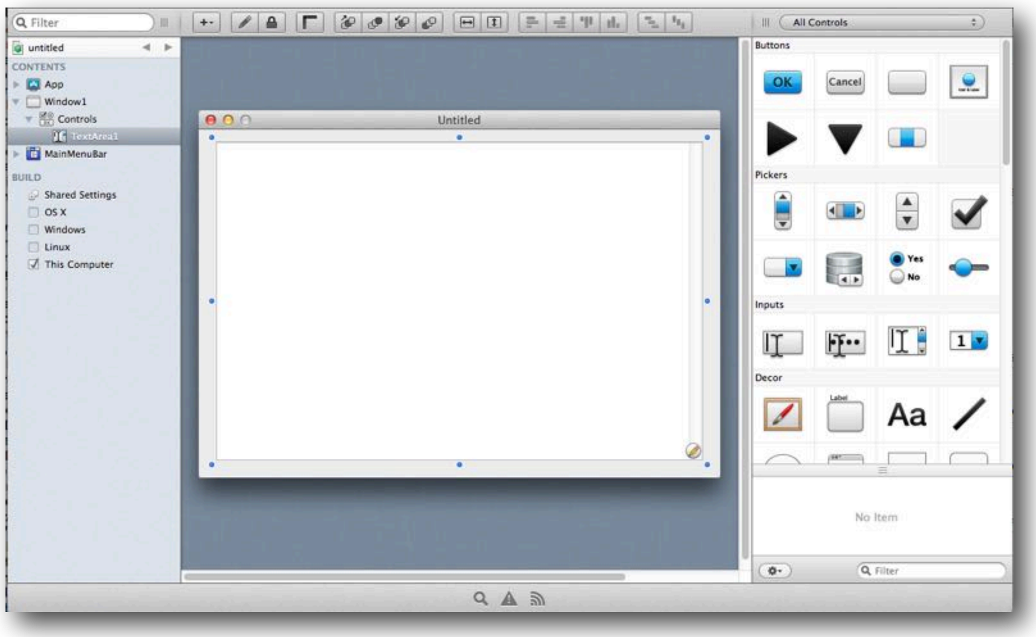
Nearly every time that you use a For...Next loop, your code will do something much more interesting than simply pop up a message box. There are many times when you will have a group of similar objects. You will often store these in an array. You will learn about arrays in depth in Chapter

Five, but for now, just think of an array as a numbered list of similar objects or variables. This section and the next few sections will cover how to step through groups of objects, but rather than get into the details of how arrays work at this point, you will use an easy to access array that is built into your computer: your fonts. Xojo has a built in function called FontCount, which returns the number of fonts you have installed. You will be using that function to gather your group of objects, which in this case will be a list of fonts.

1) Create a new desktop project in Xojo.

2) Open up Layout View for Window1 and add a Text Area control to the window.

Size it so it takes up almost all of the window:



3) Once your Text Area is in place, double click on it to open its Code Editor and locate its Opening event.

In this event, you are going to need two variables:

```
Var counter As Integer
Var myFontList As String
```

4) Set up the For...Next loop.

```
For counter = 0 To System.FontCount - 1
  myFontList = myFontList + System.FontAt(counter) + EndOfLine
Next
```

This loop will continue to add font names to MyFontList. When the list is complete, you will display it in the TextArea.

Don't worry about how System.FontAt(Counter) works. That will become clear in Chapter Five. For now, just know that it gives you the name of the next font. The font name is followed by a line break.

5) **After the loop, display the list in the Text Area:**

```
Me.Value = myFontList
```

All together, the code in your TextArea's Opening event should look like this:

```
Var counter As Integer
Var myFontList As String

For counter = 0 To System.FontCount - 1
  myFontList = myFontList + System.FontAt(counter) + EndOfLine
Next

Me.Text = myFontList
```

6) **Run your project.**

After a few seconds, you should see a list of your computer's fonts displayed in the TextArea.

7) **Quit your application.**

Now let's take a look at speeding up your code.

Your loop begins with this line:

```
For counter = 0 To FontCount - 1
```

Remember that FontCount (which is part of the System module - hence, System.FontCount) is a function built into Xojo. One thing to keep in mind is that every time this loop runs, that function will also be run. If you have a fast computer or few fonts, you may not notice the speed hit, but there is a way to optimize this code. Add a new variable:

```
Var maxFontIndex As Integer
```

Use maxFontIndex to remember the result of the FontCount function:

```
maxFontIndex = System.FontCount - 1
```

You may wonder why you are subtracting one from FontCount. This is because your computer's fonts are stored in an array, and arrays in Xojo are zero-based, as in most programming languages. This means the maximum is one less than the count. That explanation may not help much now, but it will become clearer in Chapter Five.

8) Change the first line of your loop:

```
For Counter = 0 To maxFontIndex
```

Now your loop will refer to the local value instead of running the FontCount function every time it runs. Your new code should look like this:

```
Var counter As Integer
Var myFontList As String
Var maxFontIndex As Integer

maxFontIndex = System.FontCount - 1

For counter = 0 To maxFontIndex
  myFontList = myFontList + System.FontAt(counter) + EndOfLine
Next

Me.Value = myFontList
```

9) Run your project, and you should see the same results.

You will likely not notice a speed difference in this example, but as your projects grow more complex, this type of optimization is good to know about.

10) Quit your application.

3.5 Do...Loop

Another kind of loop that is available in Xojo is the Do...Loop. A Do...Loop is useful when you need to check for a certain condition each time the loop runs. The Do...Loop has two forms, one of which does not guarantee that the loop will be run at least once, and one that does make such a guarantee.

Here is an example of a Do...Loop (as with all of the exercises in this book, feel free to create a new desktop project in Xojo and enter this code in Window1's Open event to try it out):

```
Var x As Integer
x = 1
Do Until x > 100
  x = x * 2
  MessageBox(x.ToString)
Loop
```

If you run this project, you will see a series of message boxes, each a number twice as large as the preceding number: 2, 4, 8, 16, 32, 64, 128. After you reach 128, the loop stops, because you told the loop to end once x is greater than 100. Now let's try a slightly different version of this loop. Set x to 101:

```
Var x As Integer
x = 101
Do Until x > 100
  x = x * 2
  MessageBox(x.ToString)
Loop
```

Run the project again, and no message boxes will appear. This is because the condition (x > 100) has already been met, so the loop exits. However, if you change the loop again, you should see a different result:

```
Var x As Integer
x = 101
Do
  x = x * 2
  MessageBox(x.ToString)
Loop Until x > 100
```

This time, one message box will appear containing the number 202. When the Until keyword is at the end of the loop, your loop is guaranteed to run at least once. If the Until keyword is at the beginning of the loop, the loop may not run at all, depending on whether the condition is already met.

3.6 While...Wend

A third type of loop that you will learn about in this chapter is the While...Wend loop (Wend is shorthand for While End). This is similar to the Do...Loop. Here is a code example:

```
Var x As Integer
While x < 100
  x = x + 1
Wend
MessageBox(x.ToString)
```

When run, this code will display a message box containing the number 100. The While...Wend loop is particularly useful when dealing with databases, as you will see in Chapter Twelve.

3.7 Exit and Continue

There may be situations in which your loop is searching for a matching value and can stop searching when the first match is found. For this, the Exit statement is used. To use a modified version of your font example from above, assume that you want to exit a For...Next loop once you have located the font Courier New. Add these three lines of code, a simple If...Then statement:

```
    If System.FontAt(counter) = "Courier New" Then
      Exit
    End If
```

You will add this code just inside the For...Next loop, so that your completed code looks like this:

```
    Var counter As Integer
    Var myFontList As String
    Var myFontCount As Integer

    myFontCount = System.FontCount - 1

    For counter = 0 To myFontCount
      If System.FontAt(counter) = "Courier New" Then
         Exit
      End If
      myFontList = myFontList + System.FontAt(counter) + EndOfLine
    Next
    Me.Value = myFontList
```

If you run this project, the TextArea that used to contain your full font list will now only list fonts up to (and excluding) Courier New. If you want Courier New to be included in the list, you would move your If...Then statement to after the line that builds your list of fonts:

```
    myFontList = myFontList + System.FontAt(counter) + EndOfLine
    If System.FontAt(counter) = "Courier New" Then
      Exit
    End If
```

Using the Exit statement to break out of a loop is another good way to optimize your code so that your end user spends less time waiting.

Another way to modify the behavior of your loops is with the Continue statement. Suppose that you had a loop such as this (this is pseudo-code and is not meant to be run):

```
    Var x As Integer
    For x = 1 To 100
    DoMyFunction(x)
    DoAnotherFunction(x)
    DoAThirdFunction(x)
  Next
```

Assume that DoMyFunction, DoAnotherFunction, and DoAThirdFunction are functions that will do something interesting with your integer x. But suppose that for the number 72, you only wanted to run the first function. This code would accomplish that:

```
  Var x As Integer
  For x = 1 To 100
    DoMyFunction(x)
    If x = 72 Then
      Continue
    End If
    DoAnotherFunction(x)
    DoAThirdFunction(x)
  Next
```

The Continue statement tells Xojo to skip that iteration of the loop and go back to the start. Here is a way to run only the first function for even numbers, while odd numbers would have all three functions executed:

```
  Var x As Integer
  For x = 1 To 100
    DoMyFunction(x)
    If x Mod 2 = 0 Then
      Continue
    End If
    DoAnotherFunction(x)
    DoAThirdFunction(x)
  Next
```

3.8 Hands On With Loops

For this chapter's project, you will be building the font previewer that you saw at the beginning of the chapter.

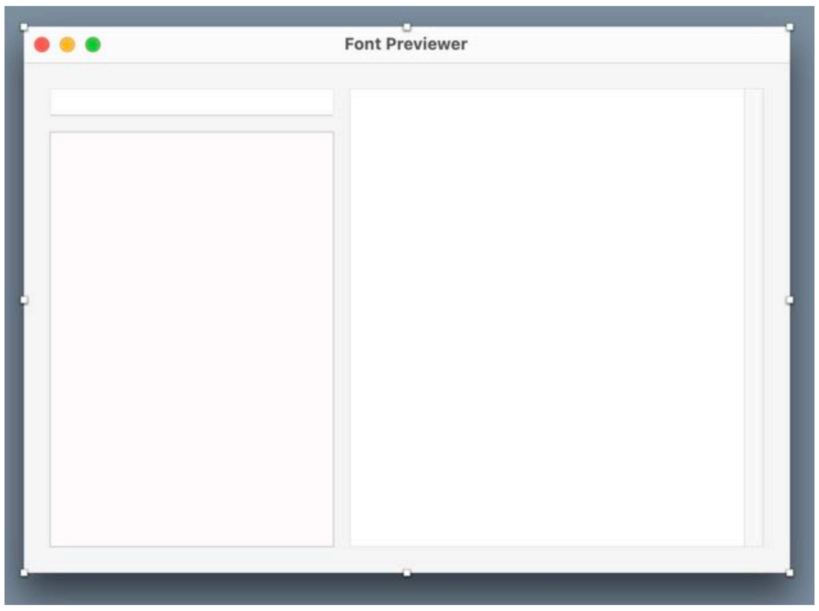
1) **Create a new desktop project in Xojo and save it as "FontPreviewer".**

2) **Set Window1's title to "Font Previewer".**

3) **Window1 will have three controls: a Text Field, a List Box, and a Text Area. Here are the names you should use:**

Control	Name
Text Field	SearchFontField
Text Area	PreviewField
List Box	FontListBox

Design your interface as you see fit, using your own creativity. You may want to use this layout as a guide:



4) **Set PreviewField's Text property to: "How razorback-jumping frogs can level six piqued gymnasts!"**

Since your application will be previewing fonts, your end user will need some sample text to look at. This is a fairly short sentence that still displays all the letters of the english alphabet. Also, set its FontSize property to 36. You'll find that on the Advanced tab (the gear icon) of the Inspector. You can get there by clicking the gear icon.

Your interface is now complete. This project will have much more code than the previous chapter's project, but it will also do a lot more. The List Box, which was named FontListBox, will display a list of all of the fonts installed on the computer. Clicking on the name of one of those fonts will update PreviewField and change the font to the selected one. The small Text Field above FontListBox, which was named SearchFontField, will allow the end user to search for fonts on the computer. The user will be able to enter all or part of a font's name, at which point FontListBox will be updated to show only those fonts that match.

With that in mind, the first thing you need to do is build your font list. You will be dealing with more controls and events than you have learned about so far, but don't be concerned with the technical details at this point. These issues will be covered in depth in Chapter Six. To build your font list, double click on FontListBox to open the Add Event Handler window. Choose the Opening event and click OK.

5) In the FontListBox's Opening event add the following code:

```
Var counter, myFontCount As Integer
myFontCount = System.FontCount

For counter = 0 To myFontCount - 1
  Me.AddRow(System.FontAt(counter))
Next
```

Most of that code should look familiar. You are using the FontCount function to determine the number of fonts on the computer and looping through them with a For...Next loop. On each iteration of the loop, you add a new row to FontListBox. Note that in any of a control's events, using "Me" refers to the control. So this line:

```
Me.AddRow(System.FontAt(counter))
```

is equivalent to this line:

```
FontListBox.AddRow(System.FontAt(counter))
```

Using Me instead of the control's name is simply shorthand, although it can come in very handy if you ever change the control's name later on.

6) Add the SelectionChanged event handler to FontListBox (the "+" button in the toolbar). Then add this code:

```
If Me.SelectedRowIndex <> -1 Then
  PreviewField.FontName = Me.SelectedRowValue
End If
```

The SelectionChanged event occurs when the user selects or deselects a row in the ListBox. Because the user could be deselecting, you need to check whether or not a row is selected. Again, don't worry too much about how the code works. It will become clear in later chapters.

7) Run the project and try selecting different fonts.

Each selection should change the font displayed in PreviewField. The Search Field doesn't work yet, but you will tackle that next.

8) Quit your application.

9) Double click on SearchFontField and add the TextChanged event handler with this code:

```
Var searchString, theFontName As String
Var counter, myFontCount As Integer
```

This event occurs whenever the text inside the TextField is modified, whether by typing, deleting, or pasting. The code in this event will be similar to the code in FontListBox's Opening event, but you will need to do some checking. This code starts by declaring your variables.

You have your SearchString, theFontName plus two integers: Counter and MyFontCount, which should be familiar from the above examples.

10) **Before continuing, save your FontCount:**

```
myFontCount = System.FontCount
```

This optimization is even more important here, since this code will run every time a letter is typed or deleted in the Search Field. Because it will run so often, you should make it as fast as possible.

11) **Add this code:**

```
If Me.Text <> "" Then
  // We'll do more stuff here
End If
```

This checks if your Search Field contains any text. If it does, you'll loop through your fonts and display any matching items. If not, you'll display the full font list.

There are two paths that this code can take. First, if SearchFontField is not empty, you will loop through your fonts and display only those whose names contain the SearchString. The following code will accomplish that:

```
searchString = Me.Text
For counter = 0 To myFontCount - 1
  theFontName = System.FontAt(counter)
  If theFontName.IndexOf(searchString) >= 0 Then
     FontListBox.AddRow(theFontName)
  End If
Next
```

On the other hand, if SearchFontField is empty, you should assume that the end user wants to see all of the fonts installed on the computer. Here is the code for that (note that it is very similar to the way you originally filled up FontListBox):

```
For counter = 0 To myFontCount - 1
  FontListBox.AddRow(System.FontAt(counter))
Next
```

12) **Add one line of code before the If statement to clear the list before anything is added to it:**

```
FontListBox.RemoveAllRows
```

You will learn about this step in greater detail in Chapter Six.

When all is assembled SearchFontField's TextChanged event should look like this:

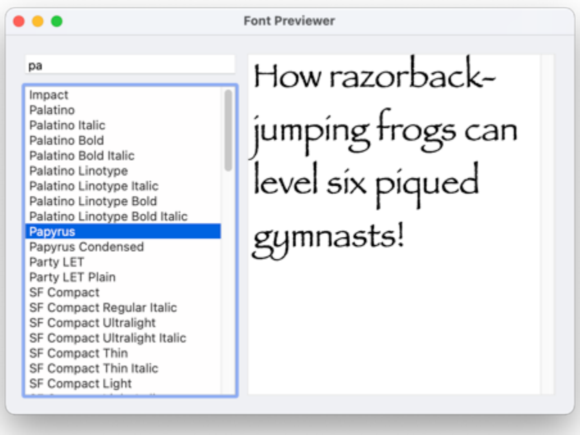
```
Var searchString, theFontName As String
Var counter, myFontCount As Integer

myFontCount = System.FontCount

FontListBox.RemoveAllRows
If Me.Value <> "" Then
  searchString = Me.Text
  For counter = 0 To myFontCount - 1
    theFontName = System.FontAt(counter)
    If theFontName.IndexOf(searchString) >= 0 Then
      FontListBox.AddRow(theFontName)
    End If
  Next
Else
  For counter = 0 To myFontCount - 1
     FontListBox.AddRow(System.FontAt(counter))
  Next
End If
```

13) **Run your project.**

You should see an application like this:



Impact
Palatino
Palatino Italic
Palatino Bold
Palatino Bold Italic
Palatino Linotype
Palatino Linotype Italic
Palatino Linotype Bold
Palatino Linotype Bold Italic
Papyrus
Papyrus Condensed
Party LET
Party LET Plain
SF Compact
SF Compact Regular Italic
SF Compact Ultralight
SF Compact Ultralight Italic
SF Compact Thin
SF Compact Thin Italic
SF Compact Light

How razorback-jumping frogs can level six piqued gymnasts!

Choosing any font from the list should cause PreviewField to be updated with that font. Entering some text into the Search Field should cause FontListBox to display only those fonts that match. As a bonus, you can change the text used for the preview to anything you like.

14) **Quit your application.**

In this chapter, you learned several different ways to react to different conditions in your code. Since it's nearly impossible to predict every condition your application will face, these skills are very valuable in everyday coding situations.

Chapter 4

Getting Things Done

CONTENTS

4.1 Chapter Overview

So far, you have entered code into specific events to make sure certain things happen at certain times. This is a common way to code, but it can lead to duplication. For example, in your Font Previewer, you had a few lines of code whose job it was to fill up the FontListBox with all of the fonts on the computer. This is some of the code in FontListBox's Open event:

```
For counter = 0 To myFontCount - 1
  Me.AddRow(System.FontAt(counter))
Next
```

You had some very similar code in the SearchFontField's TextChanged event:

```
For counter = 0 To myFontCount - 1
  FontListBox.AddRow(System.FontAt(counter))
Next
```

The code isn't quite identical, but it's very close. In fact, if you replaced Me.AddRow with FontListBox.AddRow inside your If statement, the code would be completely identical and would still function exactly the same.

This violates a fundamental rule of programming: *Don't Repeat Yourself*.

To illustrate, let's look at a real world example. Imagine that I asked you to make me some spaghetti. If you didn't know how to make it, I could tell you (in very general terms):

1) Boil some water

2) Cook the spaghetti noodles

3) Heat up some spaghetti sauce

4) Drain the noodles

5) Add the sauce to the noodles

6) Top with mozzarella cheese (optional)

If I asked you for spaghetti again next week, you would know how to make it, whether from memory or from writing down the steps. In other words, you would have a *method* for preparing spaghetti.

In programming, you will often write methods for accomplishing certain tasks, especially those tasks that may need to be performed multiple times. If you defined a method for cooking spaghetti, that would save us time later on; instead of listing each of the six steps, you could simply type this:

```
    CookSpaghetti
```

In this chapter, you will learn about methods and functions, and you will make some changes to your Font Previewer project to streamline your code using methods and functions.

4.2 Simple Methods

As stated above, a method is a set of steps for accomplishing a task. Open your Font Previewer project in Xojo and open Window1. From the Insert menu, choose Method. Xojo will show you four fields in the Inspector for you to fill out.

First is the Method Name. Name your method FillFontListBox. For now, Parameters and Return Type should be left blank and Scope should be Public. In the Code Editor in the center of the Workspace window, enter the following code:

```
Var counter, myFontCount As Integer
myFontCount = System.FontCount

FontListBox.RemoveAllRows
For counter = 0 To MyFontCount - 1
  FontListBox.AddRow(System.FontAt(counter))
Next
```

This code should look familiar; it’s the code you used to list your fonts in FontListBox. Now, navigate to FontListBox’s Opening event. Delete all of the code there and enter this:

```
FillFontListBox
```

Notice that Xojo’s autocomplete knows about your FillFontListBox method and will offer to autocomplete it for you.

Run your project. Its behavior should be identical, because when the computer reaches the line of code that says “FillFontListBox”, it refers back to your method and runs each line of it.

Quit your application and navigate to SearchFontField’s TextChanged event. You can also shorten that code by using your method. Change your If statement to match this:

```
If Me.Text <> "" Then
  searchString = Me.Text
  For counter = 0 To myFontCount - 1
    theFontName = System.FontAt(counter)
    If theFontName.IndexOf(searchString) >= 0 Then
      FontListBox.AddRow(theFontName)
    End If
  Next
Else
  FillFontListBox
End If
```

Run your project again, and again, it should behave identically. Quit your application.

If you look at the code in SearchFontField's TextChanged event, you still have some similarities there. The code to fill FontListBox with only matching fonts certainly isn't identical to your other code, but it is definitely similar.

4.3 Parameters

We’ve talked about spaghetti already. Some people, but not all, prefer meatballs with their spaghetti. Your theoretical CookSpaghetti method can handle the spaghetti, but how do you tell it to add meatballs, and only some of the time at that?

Methods can take parameters. A parameter is data that you give to a method; the method can either do something directly to the data or use it to determine how to function. Your CookSpaghetti method might take a boolean called AddMeatballs as a parameter. If AddMeatballs is true, CookSpaghetti would mix together some meat, spices, and bread crumbs, and then add those to the meal.

For a more concrete example, go back to your Font Previewer project.

As mentioned above, in SearchFontField’s TextChanged event, you have some code that isn’t a duplicate of other code, but it’s close. If you don’t have a searchString, the code does this:

```
For counter = 0 To myFontCount - 1
  FontListBox.AddRow(System.FontAt(counter))
Next
```

If you do need to match a searchString, you do this:

```
FontListBox.RemoveAllRows
For counter = 0 To myFontCount - 1
  theFontName = System.FontAt(counter)
  If theFontName.IndexOf(searchString) >= 0 Then
     FontListBox.AddRow(theFontName)
```

```
    End If
  Next
```

There are only two lines of code that are different: the If statement. You're going to add that code to the FillFontListBox method by giving it a parameter.

1) **Navigate to the FillFontListBox method and enter this into the Parameters field:**

```
searchString As String
```

Every parameter needs to have a name and a data type, just like a variable. In fact, you can think of a parameter as a variable that can be used inside the method. If necessary, toggle the disclosure triangle next to the line that says, "Sub FillFontListBox" at the top.

2) **Run your project.**

This time, it won't run, because you now have a programming error. You've told the computer that FillFontListBox has to be given a string when it runs, but you haven't given it a string.

Giving a method its parameter when you run it is called "passing" the parameter. You need to pass a string to FillFontListBox in two places: in FontListBox's Opening event and in SearchFontField's TextChanged event.

3) **In FontListBox's Opening event, you can pass an empty string. To pass a parameter, include it immediately after the method name, wrapped in parentheses, like this:**

```
FillFontListBox("")
```

You can do this because you have no font names to search for or match. In SearchFontField's TextChanged event, things will be a bit more complicated, because you're going to move most of the logic into the FillFontListBox method. SearchFontField's TextChanged will now look like this:

```
FillFontListBox(Me.Text)
```

You no longer need to check for a blank string or modify your code's behavior or even declare any variables, because all of that will now happen in the FillFontListBox method which you will now need to expand:

```
Var theFontName As String
Var counter, myFontCount As Integer
myFontCount = System.FontCount

FontListBox.RemoveAllRows

If searchString <> "" Then
  For counter = 0 To MyFontCount - 1
    theFontName = System.FontAt(counter)
    If theFontName.IndexOf(searchString) >= 0 Then
      FontListBox.AddRow(theFontName)
    End If
  Next
Else
  For counter = 0 To MyFontCount - 1
    FontListBox.AddRow(System.FontAt(counter))
  Next
End If
```

All of the logic that was previously found in SearchFontField's TextChanged event is now contained in the FillFontListBox method. In changing the project this way, you have also eliminated a lot of duplicate and near-duplicate code.

Returning to the spaghetti dinner, it is well known that many people enjoy garlic bread with their spaghetti, whether or not they have meatballs. The imaginary CookSpaghetti method will not accept multiple parameters, but this is no problem. You can give it addMeatballs As Boolean and includeGarlicBread As Boolean. To add multiple parameters to your method declaration in Xojo, enter them into the Parameters field and separate them with commas:

```
addMeatballs As Boolean, includeGarlicBread As Boolean
```

The order of parameters is critical. Imagine if you went to a restaurant and order spaghetti with no meatballs, but with garlic bread, and your meal was brought to you with meatballs and with no garlic bread. Every time you run a method, you need to be certain that your parameters are in the correct order.

4.4 Default Values

Let’s expand your virtual Italian restaurant and imagine a new method called CookLasagna. It will be similar to the CookSpaghetti method, but meatballs won’t be an option - only garlic bread. So your Parameters field would look like this:

```
    includeGarlicBread As Boolean
```

Suppose now that the chef insisted that every patron should receive garlic bread unless he or she specifically asks for it not to be included. You could change the Parameters field to this:

```
    includeGarlicBread As Boolean = True
```

What this line of code does is not only describe the parameter, but gives it a default value. Of course, this is not limited to boolean values. You could give a string parameter a default like this:

```
    myLastName As String = "Smith"
```

Setting a default value for a parameter enables us to omit that parameter when you run the method. Return to your Font Previewer project. Recall that in FontListBox’s Opening event, you run the FillFontListBox method with an empty string:

```
    FillFontListBox("")
```

Navigate to the FillFontListBox method and change its Parameters field to this:

```
    searchString As String = ""
```

That’s just two double quotes, also known as an empty string. That tells the method that if you do not specify what the searchString should be, it should give it a default of the empty string. Now you can change the Opening event of FontListBox to this:

```
    FillFontListBox
```

Run your project. Once again, its functionality is identical to what it was before, but you’ve streamlined and simplified your code.

4.5 Comments

You may have noticed that in a few of the examples used here, there have been lines of code that start with two slashes followed by a note. These are comments. Comments are another way to simplify your code. In general, your code should be “self-documenting”; that is, your method names and variable names should make it clear to someone reading your code exactly what is happening. With code that you have seen so far, that has been relatively easy to accomplish. As your code grows more complex, however, comments can become extremely valuable.

A comment is a special line of code that is simply for the developer’s reference. You may need to document what a certain variable is for or how a specific method works, or you may simply feel the need to make yourself a note for the future.

A comment can be added in two ways: the double slash and the single quote. They are interchangeable and can be added anywhere in your code. You may choose to enter a full line of comments:

```
' This method will unleash killer bees. Do not use.
```

Or you may enter your comment part of the way through an existing line of code:

```
MyLastName = NameField.Text // NOTE: use 2 fields
```

Note that anything entered on a line after the comment marker will be considered part of the comment.

There are times when it is helpful to turn lines of code into comments temporarily. This is especially true when trying to track an error. You may do this manually, by typing the double slash or the single quote in front of each line. Xojo also has a feature that will comment/uncomment several lines of code at one time. Highlight the code you wish to comment out, and then choose Comment from the Edit menu. If the code is already commented with single quotes, this command will uncomment it.

4.6 Functions and Return Values

Some methods can report back to you. This result is called a return value. It can be data of any type: a boolean to tell you whether or not your method was successful, a numeric result of a mathematical calculation, or some text that has been concatenated. A method that returns a value is sometimes known as a function.

Return to your Font Previewer project.

1) **Add a new method called "GetSelectedFont" to Window1.**

   You can add a new method by choosing Method from the Insert menu. It won't take any parameters, but its Return Type should be String, because this method will return some text.

   This method will find out if a font has been selected in FontListBox and if so, return its name to us.

```
Var currentFont As String
currentFont = FontListBox.SelectedRowValue
Return currentFont
```

   Now the interface for your Font Previewer needs to be modified.

2) **Raise the bottom edge of FontListBox and add a Button.**

3) **Name the Button "FontButton".**

   Again, don't worry too much about the exact positions of your controls; use your creativity. Give it "Which Font?" for a caption.

4) **Add this code to FontButton's Pressed event:**

```
Var fontName As String
fontName = GetSelectedFont
MessageBox("You have selected " + fontName)
```

5) **Run your project and select a font.**

6) **Click on FontButton and your message box should appear.**

7) **Quit your application.**

Now it's time to simplify the code in FontButton's Pressed event. You have accessed the return value of the GetSelectedFont method by using this line of code:

```
fontName = GetSelectedFont
```

While this is perfectly fine, it’s actually an unnecessary step. As soon as your method runs, you have the return value and you can use it in your code. You can almost treat the method like a variable. Remove all of the code from FontButton’s Pressed event and enter this instead:

```
    MessageBox("You have selected " + GetSelectedFont)
```

Run your project again, and you should see identical behavior, but once again, with simpler code.

4.7 Scope

With all of these methods, you have variables all over the place in your code. And that’s fine. But it is important to remember that these variables are not accessible everywhere. This is because of something called scope.

A variable’s scope determines where that variable can be used. For example, the FillFontListBox method has three variables that you declared: theFontName, counter and myFontCount. These variables can only be used within the FillFontListBox method. Code in other methods and in your controls’ events has no knowledge of these variables, and trying to access them will result in an error. There are ways to make variables accessible on a larger scale, but for now, just remember that when you declare variables in a method or event, those variables may only be used in that method or event.

Chapter 5

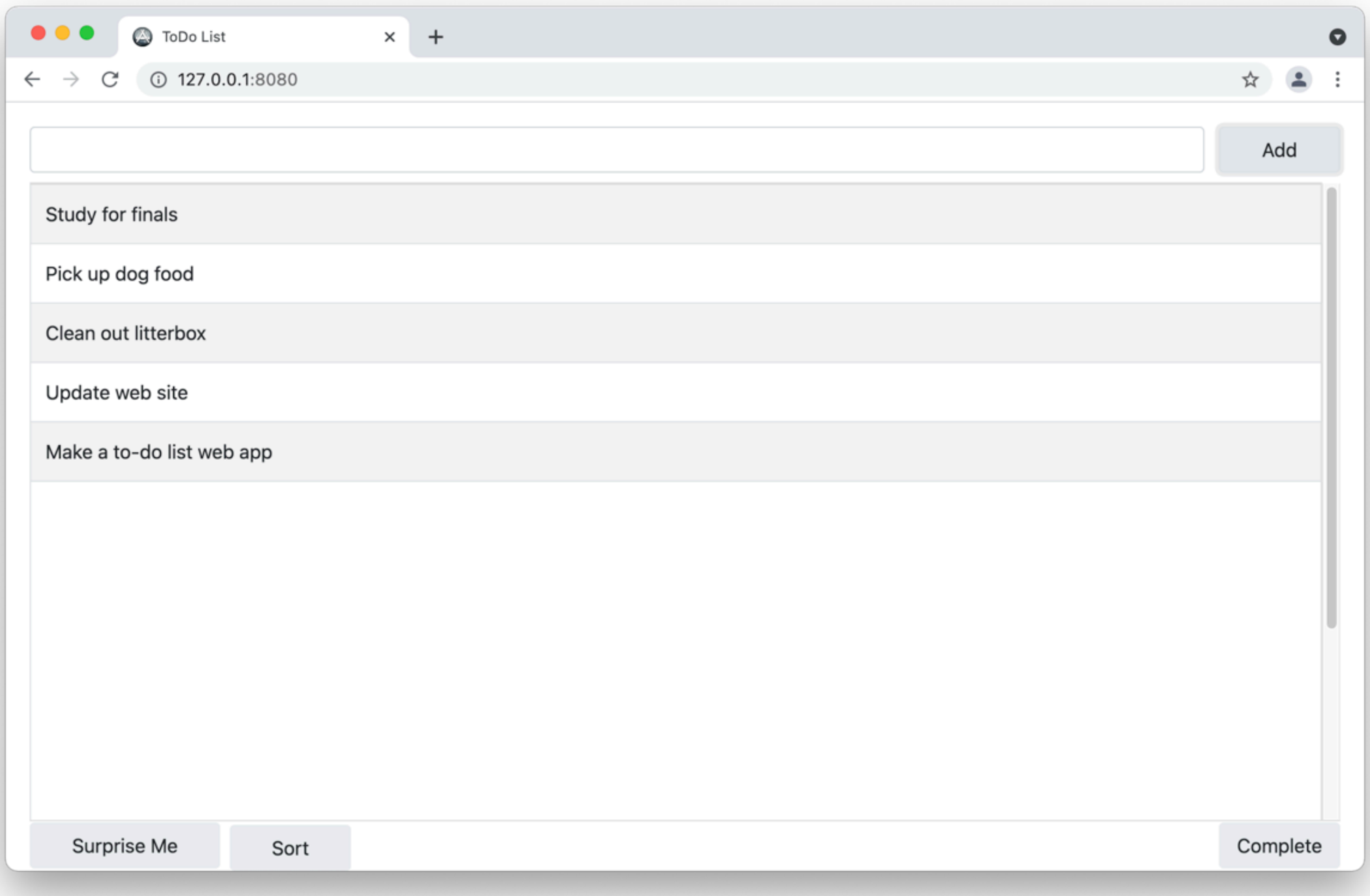
Making A List

CONTENTS

5.1 Chapter Overview

One issue that you will often face in programming is maintaining a list of items. These items could be text, numbers, colors, dates, or any type of data that you can imagine. In Xojo, and in most other programming languages, this is done using an array. An array is simply a numbered list of similar items.

In this chapter, you will use arrays and build on your knowledge from previous chapters to build a to do list manager. One big difference is that this app will be a web app. Here is what you'll be making (this copy of the app is running in Firefox on a Mac, but it can run in any modern browser on any operating system):



Study for finals
Pick up dog food
Clean out litterbox
Update web site
Make a to-do list web app

5.2 Adding To The List

Creating an array is very similar to creating any other type of variable. There's just one thing that you must add when you declare your variable. Remember that you can create a string using this syntax (and remember from Chapter 2 that a string is a piece of text):

```
Var myString As String
```

If you wanted an array of strings, you would simply add some parentheses:

```
Var myStringList() As String
```

myStringList is now a list of strings. In this example, your list contains no items. You can still add items to it, but if you wanted an array with a specific number of items, you may specify that number in the parentheses:

```
Var myStringList(10) As String
```

Many times, you will not know how many items are in an array. You can ask Xojo how many items are in an array by using the array's Count function:

```
myArrayCount = myStringList.Count
```

As mentioned earlier, myStringList itself is not a string; it's an array of strings. Because it's not a string, you can't use it in the same way you've been using strings up until now. For example, you can display a string to the end user by using a message box:

```
MessageBox(myString)
```

You can't display an array to the end user in the same way. However, you can display one of the string items. If you wanted to display the first item from myStringList, you could use this code:

```
    MessageBox(myStringList(0))
```

Note that the first item in an array is item number zero. There are historical reasons why this is the case, but it's worth keeping in mind, because one of the most common errors made when working with arrays is starting at number one instead of number zero. This also means that the last (highest numbered) item in the array is one less than its count. If an array has ten items, the highest numbered item is nine because they are numbered 0 to 9. Fortunately, arrays also have a LastIndex function that will return this value.

So you can treat myStringList(0) in the same way that you treat any other string. If you want to assign a value to it, you can do so:

```
    myStringList(0) = "Just your average string here"
```

And you can do the same with myStringList(1), myStringList(2), and so on. Each item in the array is just a normal variable. So in the case of an array of strings, each item is just a regular string. If you had an array of integers, each item would be a regular integer. To declare an array of integers, use a syntax similar to what you used above:

```
    Var myNumberList() As Integer
```

Again, this would give us an empty array of integers. If you wanted to specify the number of items in the array, do so as you did above:

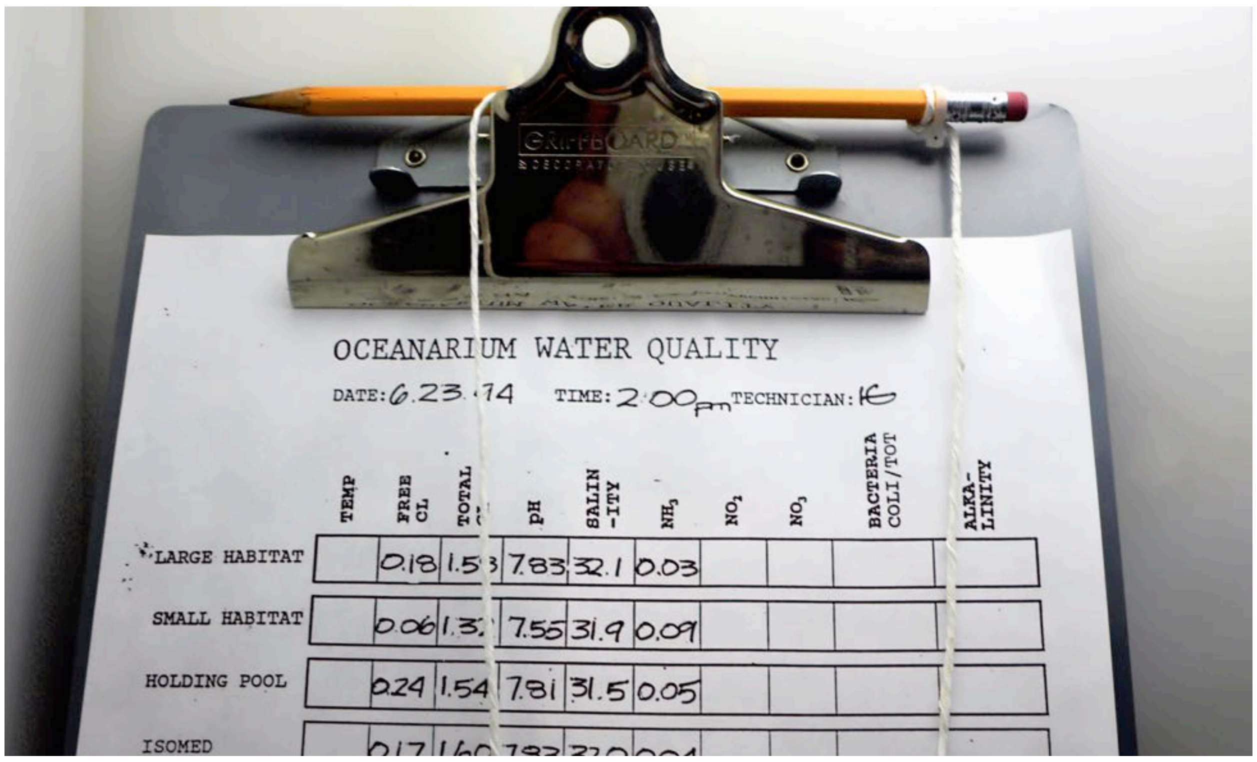
```
    Var myNumberList(5) As Integer
```

That would give us an array with six elements. As you can see, you create and treat arrays the same regardless of what type of data they hold. In a similar vein, you treat each item in an array as a normal variable of its data type.

Creating an array is one thing, but an array isn't valuable until it contains some items. To add an item to an array, you may use the array's Add or AddAt methods.

When you add an item to an array, the item is added to the end of the array, and the number of items in the array (as well as its LastIndex) is increased by one:

```
    Var myStringList(10) As String
    // myStringList.LastIndex is equal to 10, items are numbered 0 through 10.
    myStringList.Add("Hey there")
    // myStringList.LastIndex is now equal to 11
```



OCEANARIUM WATER QUALITY

DATE: 6.23.14 TIME: 2:00pm TECHNICIAN: KG

	TEMP	FREE CL	TOTAL	pH	SALIN -ITY	NH_3	NO_2	NO_3	BACTERIA COLI/TOT	ALKA- LINITY
LARGE HABITAT		0.18	1.53	7.83	32.1	0.03				
SMALL HABITAT		0.06	1.32	7.55	31.9	0.09				
HOLDING POOL		0.24	1.54	7.81	31.5	0.05				
ISOMED		0.17	1.60	[illegible]	[illegible]	[illegible]				

It may not always be the case that you want to add an item to the end of the array, though. If you need to add an item at a specific location, use the AddAt method. Where the Add method only takes one parameter (the value you wish to add to the array), the AddAt method takes two: first, an integer that specifies the items's location in the array, and then the value itself. An item's location in the array is known as its Index. Using the AddAt method will increase the number of items in the array by one, and will also increase the index of each successive item by one. This is best illustrated with a code example:

```
    Var stooges() As String
    // stooges.LastIndex is now equal to -1
    stooges.Add("Larry")
    stooges.Add("Curly")
    stooges.Add("Moe")
    // stooges.LastIndex is now equal to 2
```

The stooges array now looks like this:

Index	Value
0	Larry
1	Curly
2	Moe

If you run the following line of code:

```
    stooges.AddAt(1, "Shemp")
```

stooges.LastIndex is now equal to three, and the array itself looks like this:

Index	Value
0	Larry
1	Shemp
2	Curly
3	Moe

5.3 Clearing Out

As you saw in the example above, the AddAt method will “shift” the other items in the array. Another method that will shift the items in your array is the RemoveAt method. This method, as implied by its name, will remove an item from the array, decrease the number of items in the array by one, and decrease the index of the remaining items. Continuing the example above:

```
stooges.RemoveAt(2)
```

The stooges.LastIndex value is now equal to two, and the array itself looks like this:

Index	Value
0	Larry
1	Shemp
2	Moe

Removing one item at a time is certainly useful, but there will be many occasions in which you need to remove all items from an array at once. While you could certainly use a For...Loop and remove each item manually, that would be time consuming and error prone. Fortunately, arrays in Xojo includes a RemoveAll method:

```
stooges.RemoveAll
```

The stooges array is now empty and contains no items. Its LastIndex is now -1.

5.4 Managing Order

Every time you use an array, the order of the items matters. The computer will remember which item is in which position and will preserve that information. Because of this, every time you work with the same array, you can know that whatever data you stored in the tenth position, for example, will remain in the tenth position until you specify otherwise.

But you will encounter situations in which you want to reorder the items in your array. There are certainly “brute force” methods of doing this, but Xojo includes two array functions that are very useful for reordering arrays.

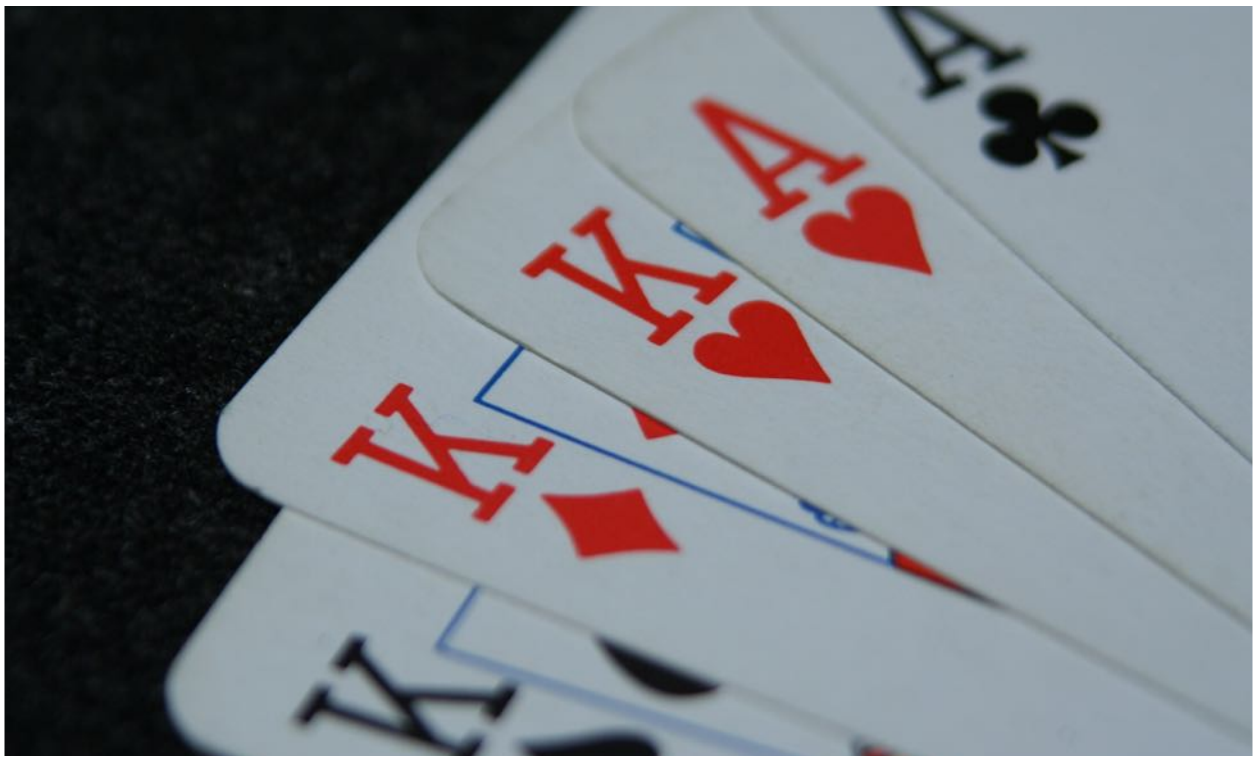
The most obvious way to reorder the items in an array is to sort them. To do so, use the Sort method:

```
Var contacts() As String
contacts.Add("Zeke")
contacts.Add("Abe")
contacts.Add("Mary")
contacts.Sort
```

The contacts array now looks like this:

Index	Value
0	Abe
1	Mary
2	Zeke

An array of strings will be sorted alphabetically in ascending order. If your array contains numeric data (in a numeric data type), it will be sorted numerically. When the array is sorted, each item's index, or position in the array, is adjusted accordingly.



Another way of reordering the items in an array is to shuffle them, or rearrange them into a random order. Imagine that you were building a game that included a standard deck of 52 cards. As with most card games, you would likely need to shuffle the deck at some point. Assume that in your imaginary card game, you have an array of strings called cards, which contains 52 items (two through ten, plus Jack, Queen, King, and Ace for each of the four suits). To shuffle the deck:

```
    cards.Shuffle
```

The cards will now be in random order. That's all there is to it. Shuffle works on any type of data, and it is completely random. If you shuffle the same array twice, you will most likely get different results each time (there is a statistically insignificant chance that the computer could produce the same exact set of random numbers twice in a row, but it's so small that it's barely worth mentioning).

Shuffle is useful outside of card games. Suppose you were creating an interactive quiz. You could use Shuffle to randomize the order of the questions, and the order of the multiple choice answers for each question, in order to minimize cheating.

5.5 Converting Text Into A List

So far, you have been adding data to your arrays using the Add method, and while this is perfectly fine, there will be times when you may need to create an array based on one existing string. Imagine that you were writing an application that allowed the end user to “tag” content by entering keywords separated by commas. You might use a Text Field called TagField for the user to enter this data, and you would likely end up with a string that looks similar to this:

Movies,Comedy,90s

Now imagine that you needed to turn this string data into an array. You could use the IndexOf function that you learned about in Chapter Three to find all of the commas and parse the data yourself. Or you could use the ToArray function built-in to strings. ToArray will take a string and separate it into an array, based on the delimiter you give it. Returning to the tagging example above:

```
Var tags() As String
tags = TagField.Text.ToArray(",")
```

The array tags would then contain one item for each part of the string that was separated by a comma. If the user entered “Movies,Comedy,90s” as listed above, the tags array would have three items:

```
// tags(0) = "Movies"
// tags(1) = "Comedy"
// tags(2) = "90s"
```

The String.FromArray function works in the opposite way. FromArray takes an array of string and a delimiter, and puts it all together in one long string:

```
Var tags() As String
Var combinedTags As String
tags.Add("Movies")
tags.Add("Comedy")
tags.Add("90s")
combinedTags = String.FromArray(tags, ",")
// combinedTags is now "Movies,Comedy,90s"
```

In this example, note that there are no spaces in combinedTags. If you wanted to have a comma and a space between each item, you would use that as your delimiter. The delimiter, in both ToArray and FromArray, can be any string you want to use.

5.6 Using Key/Value Pairs To Store Data

As discussed above, arrays are useful for storing lists of similar items. In addition, they maintain the order of elements until you modify it. But the only way you can look up a value is by using the index. Some languages have a feature called associative arrays that essentially allow you to use any data type as an “index” by which you can look up a value. So, while a traditional array might look like this:

Index	Value
0	Abe
1	Mary
2	Zeke

Each row in an associative array would contain a key instead of an index:

Key	Value
0	Abe
1	Mary
2	Zeke

Xojo does not have associative arrays, but it does have a data type called Dictionary that is similar to an associative array in many ways. It can be used to store lists of data, but the order of the data is not maintained. Also, instead of using a simple numeric index to track its position in the list, the Dictionary uses a Key/Value relationship, where both the Key and the Value can be of any data type. The Dictionary is technically a class, so it must be instantiated before it can be used (similar to the DateTime data type you learned about in Chapter Two).

To create a Dictionary, use the Var keyword, just as with other data types:

```
Var settings As New Dictionary
```

Notice that this example used the shorter method of instantiating your variable by including the New operator on the same line as the Var keyword.

To add data to an array, you used the AddRow and AddRowAt methods. For a Dictionary you instead set a Value and provide a Key:

```
Var settings As New Dictionary
settings.Value("Name") = "Anakin Skywalker"
settings.Value("Title") = "Sith Lord"
```

```
    settings.Value("Dark Side") = True
    settings.Value("Age") = 39
    settings.Value("Trainer") = "Kenobi"
```

If you take a look at the code above, you may notice that you have used strings, an integer, and a boolean as Values in your Dictionary. This is perfectly fine; the Dictionary can use any data type as a Value, or as a Key, unlike arrays, in which each item must be of the same data type.

Although it may not be obvious from the example, you have provided Keys as well. Take this line of code for example:

```
    settings.Value("Dark Side") = True
```

The Key in this line is “Dark Side” and the Value is a boolean, which happens to be true in this example. So in essence, you are storing data very much like an array, but you are manually assigning the Key rather than using a numeric index.

Why is this useful? Suppose you wanted to retrieve information from the Dictionary, such as the Value you stored with the “Title” Key. Continuing the code:

```
    Var theTitle As String
    theTitle = settings.Value("Title")
    // theTitle = "Sith Lord"
    MessageBox(theTitle)
```

Because you can use anything for the Key, a common error is a KeyNotFoundException, which occurs when you attempt to retrieve a Value for a Key that doesn’t exist. To avoid this error, name your Keys carefully and logically so you can easily remember them.

If you encounter a situation where you know what the Key should be, but are unsure if it exists or not, you may use the Lookup function. The Lookup function takes two parameters: the first is what you believe the Key to be and the second is a default value to use if the Key can not be found. In your Dictionary above, you haven't set a Value for a "Lightsaber Color" Key, but can try to access it anyway:

```
    MessageBox(settings.Lookup("Lightsaber Color”, "Red"))
```

Because the "Lightsaber Color" Key does not exist, the message box in this case will display "Red." If you use an existing key:

```
    MessageBox(settings.Lookup(“Title", "Pilot"))
```

... then you'll see the Value from the Dictionary as expected.

To check for the existence of a specific Key, use the HasKey function. HasKey takes the Key you're looking for as a parameter and returns a boolean: true if the Key exists and false if it does not:

```
    If settings.HasKey("Mentor") Then
      MessageBox("Mentor: " + settings.Value("Mentor"))
    Else
      MessageBox("No mentor was found")
    End If
```

To remove an entry from the Dictionary, use the Remove method, which takes the Key as its parameter:

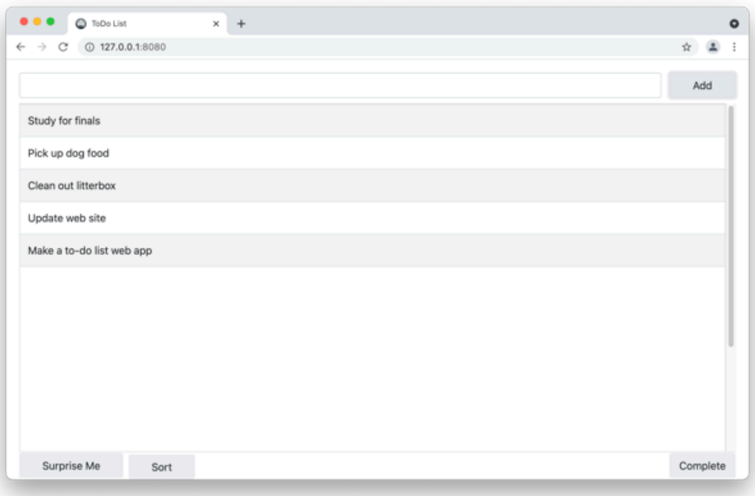
```
    settings.Remove("Dark Side")
```

To remove all entries from the Dictionary at once, use the RemoveAll method:

```
    settings.RemoveAll
```

5.7 Hands On With Arrays

As mentioned at the beginning of the chapter, your next sample project is a web-based To Do List Manager. The finished web app might look something like this:



1) **If you haven't already done so, launch Xojo and create a new Web Application. Save it as "ToDoList".**

To start, you'll build the interface, then add your code later. The following table lists the types and names of controls you will need. As before, don't worry too much about where each control should be positioned; use the screenshot at the beginning of the chapter as a guide, but feel free to use your creativity as well.

Your end user will enter To Do Items into ToDoField, then click AddButton to add them to ToDoListBox. The data for ToDoListBox will be stored in an array behind the scenes.

Since the ToDoListBox does not need a column header, you can turn off its column header by switching the HasHeading property in its Inspector to OFF.

2) **Lock the controls as listed below.**

If you have resized any of your running apps so far in this book, you probably noticed that the controls haven't

Control	Name	Caption
Text Field	ToDoField	N/A
Button	AddButton	Add
List Box	ToDoListBox	N/A
Button	ShuffleButton	Surprise Me
Button	SortButton	Sort
Button	CompleteButton	Complete

necessarily behaved as expected; instead of expanding to fill the window, they stay the same size and the same distance from the top left corner. This can be fixed by changing the control locking in the Inspector. You can lock any edge of a control; when that edge is locked, it will maintain its distance from that edge of the window. So a control locked to the left and top will stay in the same position, while a control locked to the right and top will always be in the upper right corner of the window. Your to-do app should fill the browser window, so lock the controls as listed in this table.

To lock a control, click the padlock under "Locking" in the Inspector. A locked padlock indicates that that edge of the control is locked, while an unlocked padlock, naturally, indicates the opposite.

3) **Add a property to WebPage1. Its type should be String and its Name should be "ToDoList()".**

In the last chapter, you learned a bit about scope, or where a variable can be accessed. Because every control on your window will need access to your array of To Do Items, you can't simply declare it in a method. You need to make it a property of the web page. Adding a property to a web page (or a window in a desktop app) means that it is accessible from any code on that web page, whether it's part of a method you create or in a control's event.

Control	Lock
ToDoField	Left, Top, Right
AddButton	Top, Right
ToDoListBox	Left, Top, Right, Bottom
ShuffleButton	Left, Bottom
SortButton	Left, Bottom
CompleteButton	Right, Bottom

4) **Add a method called "UpdateToDoListBox" to WebPage1.**

Whenever a change is made to your array, you'll need to update the data displayed in ToDoListBox. Because you'll need to do this from several places in your code, you'll create a method for it. Choose Method from the Insert menu. Name the method UpdateToDoListBox.

5) **Add this code to the UpdateToDoListBox method:**

```
ToDoListBox.RemoveAllRows
For Each row As String In ToDoList
  ToDoListBox.AddRow(row)
Next
```

UpdateToDoListBox will do the work of making sure that what's displayed in ToDoListBox matches what's in your toDoList. It will clear any existing data from ToDoListBox, then add a row for each row in your ToDoList array. In this method, we're using a different type of For loop called a For Each loop. You define a variable (*row* in this case) and the For Each loop loads each value of the array ToDoList into the variable row. As you can see, it greatly simplifies looping through an array. No need for a counter variable or use of an index.

6) **Add this code to AddButton's Pressed event:**

```
ToDoList.AddRow(ToDoField.Text)
UpdateToDoListBox
```

```
ToDoField.Text = ""
```

This code provides a way for your end user to get data into the array. The data will come from ToDoField, and ToDoButton will do the work of adding it to the array. It will then run the UpdateToDoListBox method. Finally, it will clear the contents of ToDoField so that your user doesn't accidentally add the same item multiple times.

As noted above, the first line adds your user's text to the array. The second line updates your display. And the third line clears out any existing text in ToDoField. If you run the project now, you should be able to add items to the list.

7) Add this code to CompleteButton's Pressed event:

```
If ToDoListBox.SelectedRowIndex <> -1 Then
  ToDoList.RemoveRowAt(ToDoListBox.SelectedRowIndex)
  UpdateToDoListBox
End If
```

Any good To Do List Manager should also allow you to mark items as complete. That will be the job of CompleteButton. Its job will be more complex than it may appear. First, it will need to determine if a To Do Item is selected. If one is selected, it will need to determine its position in the array and then remove it. Finally, it will update your display. You may recall from a previous chapter that you can find out which row in a ListBox is selected by checking its ListIndex property. If the ListIndex equals negative one, nothing is selected. Otherwise the ListIndex will be the row number that is selected (as with arrays, ListBoxes are zero-based, so the first row is number zero).

8) Add this code to SortButton's Pressed event:

```
ToDoList.Sort
UpdateToDoListBox
```

9) Add this code to ShuffleButton's Pressed event:

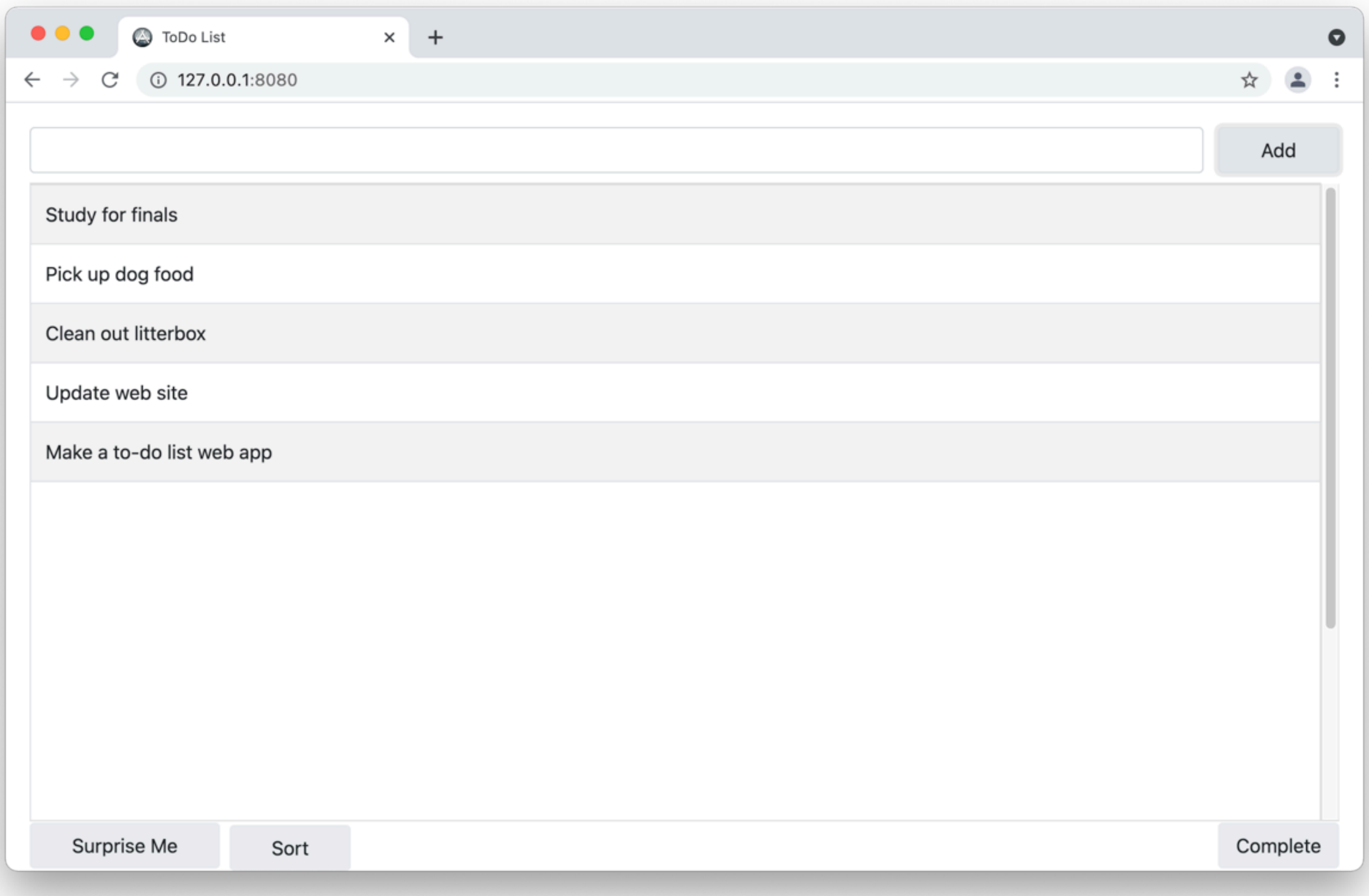
```
ToDoList.Shuffle
UpdateToDoListBox
```

Finally, you want to allow your end users to sort and shuffle their To Do Items. The previous two steps provide that functionality.

In reality, randomizing the order of your To Do Items probably has very little practical value, but this is just an example that can be applied to other apps and concepts.

10) Save and run your project.

Your app will open in your computer's default web browser. You should now be able to add and remove items, as well as shuffle and sort them.



Study for finals
Pick up dog food
Clean out litterbox
Update web site
Make a to-do list web app

11) Quit your application.

Chapter 6

May I Take Your Order?

CONTENTS

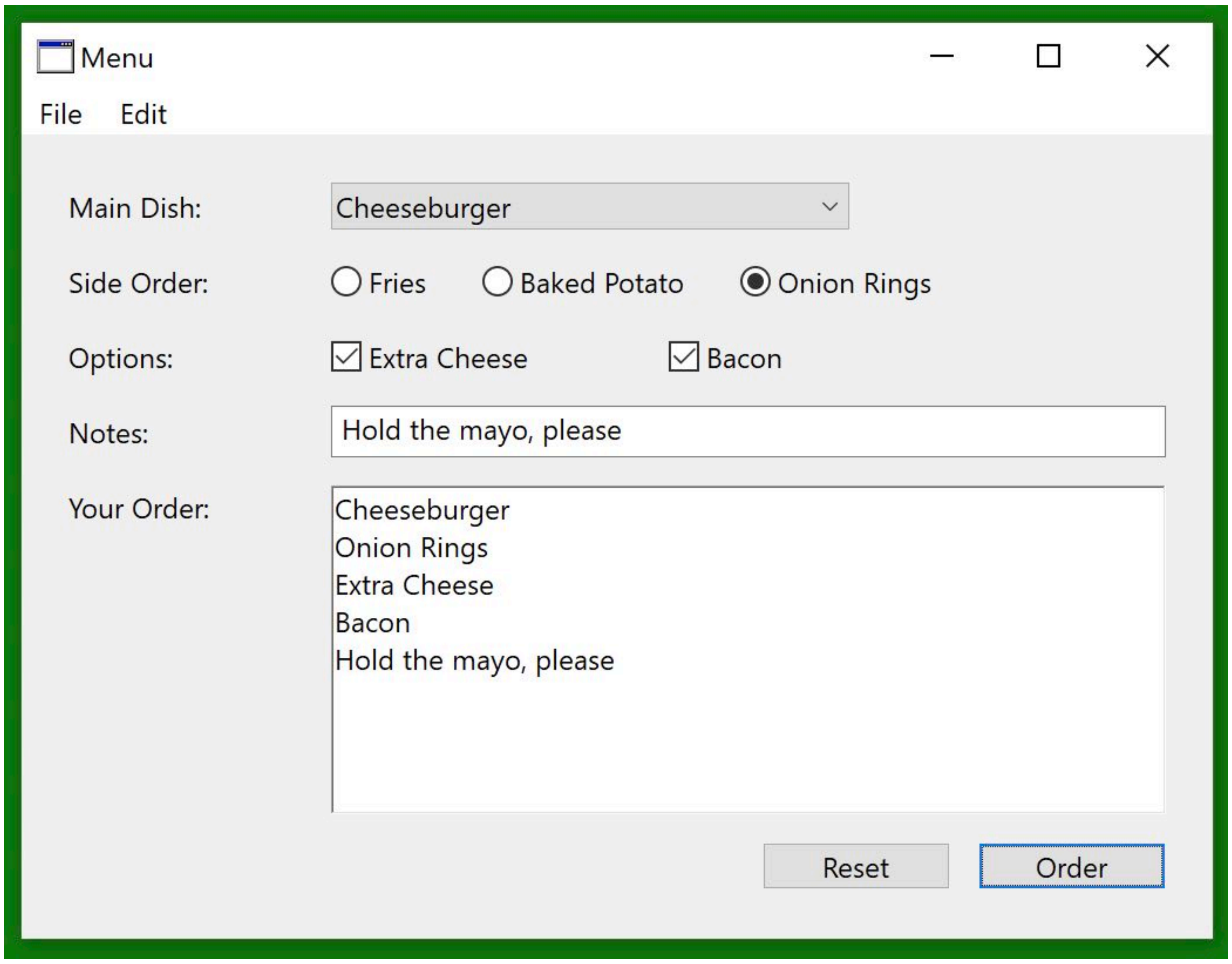
6.1 Chapter Overview

In this chapter and the next, you will learn about two critical aspects of Xojo: events and controls. Events and controls allow you to respond to your user's actions in a meaningful way.

Events are just that: things that happen. An event can be triggered, or called, by the user clicking on a button or typing into a field. Or it could be something initiated by the computer. In this chapter, you will learn about some events that are commonly used.

A control, as has been touched on in previous chapters, is an interface element, such as a button, a text field, or a popup menu. You will learn about many different controls in this chapter and the next, but some will be left for your further research (for example, controls that are specific to one platform, such as Windows or macOS).

Your sample project in this chapter will be an electronic meal menu.



Menu

File Edit

Main Dish: Cheeseburger

Side Order: ○ Fries ○ Baked Potato ● Onion Rings

Options: ☑ Extra Cheese ☑ Bacon

Notes: Hold the mayo, please

Your Order:

```
Cheeseburger
Onion Rings
Extra Cheese
Bacon
Hold the mayo, please
```

6.2 Introduction to Events

As mentioned above, an event is something that happens. Sometimes an event is triggered when the user does something, like clicking a button or choosing a menu item. Other times, an event is triggered by the computer or by the app, such as when an application launches or a window is opened.

Like methods and functions, events provide you with an opportunity to determine how your app will behave. Just as you entered code into your methods and functions, you can enter code into an event. The difference is that you have to tell the computer when to run a method or function; with events, you tell the computer to run certain code when a certain something happens.

For example, many controls have an event called Opening. This event is called when a control is first being created (or in technical terms, instantiated) on a window. You may have a popup menu that needs to have a specific set of items to choose. You can set up those items in the popup menu's Opening event.

Your popup menu also has a SelectionChanged event that is called when the user makes a selection. You can use the SelectionChanged event to respond to the user's choice.

Events are not limited to controls. Your windows (note: the interface element, not the operating system!) also have events. For that matter, your application itself has events!

6.3 Windows

Remember, this is not the operating system, it's the interface element on your screen that contains other controls.

1) **Launch Xojo and create a new desktop application. Save it as "Events".**

2) **Add a List Box to Window1.**

   Feel free to make it any size, but make sure it's wide enough to show several words and tall enough to show several lines at once.

3) **Add this code to Window1's Opening event:**

```
ListBox1.AddRow("Opening")
```

   Once your List Box is in place, click on the window background, then go to the Insert menu and choose Event Handler (it's important to make sure the window itself is selected, otherwise Xojo will add the event handler to whatever control is selected). A list of events will appear. Select the Opening event and press the OK button. You will be taken to Window1's Opening event.

4) **Repeat Step 3 for Window1's Activated, Deactivated, Moved, and MouseDown events.**

   Don't add the word "Opening" to ListBox1, though; use each event's name.

5) **Run your project.**

6) **Click away from your application and back to it. Drag the window around. Click in the window outside the List Box.**

   You should see the events being populated in your List Box.

7) Quit your application.

This should give you a rough idea of how events happen. When you move the window, switch applications, or click in the window, the events you implemented are called. Actually, the events are called no matter what; it's simply a matter of how you choose to respond to them.

Another helpful event is the Closing event. As implied by its name, this event is called when the window is closing. This is a good event to use to store things like the window's position so that you can present it to your user in the same way the next time it is opened.

Windows have other events that won't be covered in this book. You are encouraged to explore these events on your own. The List Box technique above is an excellent way to learn what calls different events.

Xojo is not the only language that uses events; many other languages use them as well, such as Swift, JavaScript, Java and C++.

In addition to events, windows also have different properties that can be set. Many of these properties can be set using the Inspector. Two important window properties that are often confused are the Name and the Title. The window's Name is how you refer to the window in your code. A window's default name is Window1, but you may change it to anything you like; it's always a good idea to use a more meaningful name, one that reflects its purpose, like EditingWindow or PreferencesWindow. The window's Title, on the other hand, is the text that will appear at the top of the window in the running application. A window's Title can also be set in code if you need to change it while the application is running.

A window also has several properties related to its size. The most obvious are its width and height, which are expressed in points. It's important to remember when designing the layout of your application that screens come in many different sizes. If you have a very large screen, bear in mind that you need to design for a smaller screen. For example, your screen may have a resolution of 1920 by 1080 (a common resolution for 24 inch displays), but many users' screens have a lower resolution. In general, it's a good idea to assume that your application's windows will need to fit into a screen with a resolution of 1024 by 768.

That's not to say that users with larger displays won't be able to resize your window to fit. That's where two more window properties come into play: MaximumWidth and MaximumHeight. These numbers indicate how large the window is allowed to become. The default value for both properties is 32,000 points, which is so large that no user is likely to run into that limitation. If you need to set a smaller maximum size, use these properties to do so.

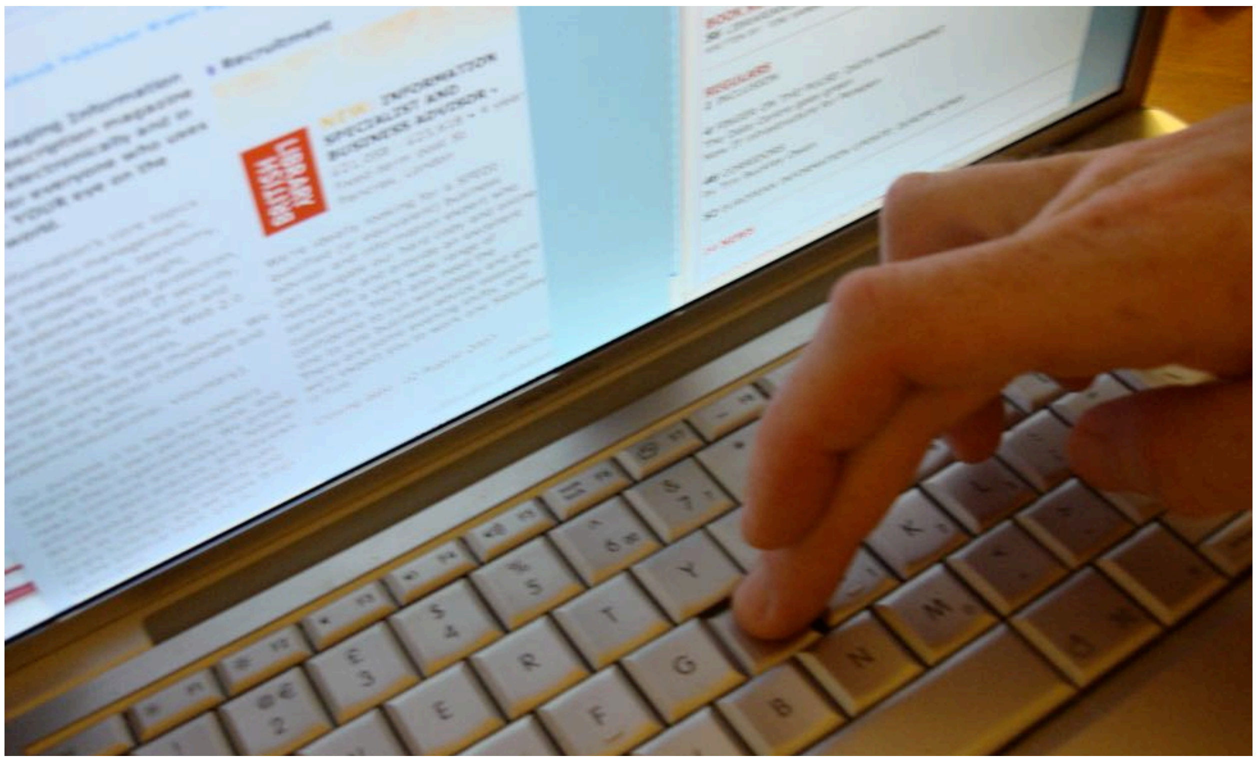
On the other hand, you may also need to define a minimum size for your window using the MinimumWidth and MinimumHeight properties. These properties determine how small a user can make the window. For example, you may have some interface elements that require a certain amount of space. Using MinimumWidth and MinimumHeight, you can guarantee that the user will not be able to resize your window to too small a size.

The maximum and minimum widths and heights assume that your window's size is adjustable at all. You can set this using the Resizable property. If Resizable is true, the user will be able to shrink or expand your window using the operating system's native abilities. If it is false, the user will be unable to do so (although you may still set the width and height in code when necessary).

On most modern operating systems, windows have three buttons built into the windows themselves: close, minimize, and maximize (sometimes called zoom). You can decide whether you want these buttons to be active or not by setting the HasCloseButton, HasMinimizeButton, and HasMaximizeButton properties. Depending on what operating system your application is running on, the buttons may still appear, but will be disabled.

6.4 Input

In earlier chapters, you used a Text Field to get information into your application. The Text Field is one of several input controls that you may use in your projects, the others being Text Area, Password Field, and Combo Box. In this section, you will learn about the events and properties of each of these controls.



But first, all of these controls have some properties and events in common. For example, all four input controls have properties to set their size and position: Left, Top, Width, and Height. These are all measured in pixels. Note that the Left and Top properties are relative to the window that's holding the control and not the screen itself.

Also, all four of these controls have a Name property. Like a window's Name property, this is the name by which you will refer to the control in your code. Again, it is good practice to give your controls meaningful names related to their purpose, such as FirstNameField or UsernameField.

These controls also have some common properties related to their contents. One such property is the Text property. The Text property contains whatever text value is contained in the control such as the text in a Text Field. So if you have a Text Field called FirstNameField, you can access its contents by using this code:

```
firstName = FirstNameField.Text
```

Sometimes your user will have some of the text selected. The Text Field, Text Area, and Password Field have a property called SelectedText, which will give you only the selected text. If you wanted to grab the selected text from a Text Field called FirstNameField, you could use the following code:

```
    currentSelection = FirstNameField.SelectedText
```

Other times, you may need information about the selected text, such as its length or position, without needing to know exactly what the selected text is. All four of the input controls have two properties called SelectionStart and SelectionLength. SelectionStart will give you the position of the first selected character and SelectionLength will give you the length of the selection text.

The Text Field itself is a simple text box that allows the user to enter plain text on a single line. It doesn't support inline styles like bold or italics or different fonts. However, you can set the entire Text Field to be bold or italic or a specific font. Whatever style, font, and text size you choose will be applied to the entire Text Field.

This differs from the Text Area, which has a Styled property. If the Styled property is set to true, your Text Area can support multiple fonts, text sizes, and styles. These styles can be a result of text that's pasted in from an outside source or they can be set by your code. A few paragraphs ago, you were introduced to the SelectedText, SelectionStart, and SelectionLength properties. The TextArea, since it supports styled text, builds on these by adding some style related selection properties: SelectionBold, SelectionItalic, SelectionFont, SelectionFontSize, and others. Suppose your application had a Text Area called BiographyField, where a user was to enter some background information on themself, and you wanted to allow the user to make certain words bold. You could add a Button to the window with this code in its Pressed event:

```
    BiographyField.SelectionBold = True
```

Setting SelectionBold back to False turns off the bold style. Most style buttons toggle a style on and off, which you could do with this code:

```
    BiographyField.SelectionBold =  Not BiographyField.SelectionBold
```

That code would toggle the bold style on or off without needing to know the current state. The Not keyword in the above example indicates opposite meaning, so that you can toggle the Bold property on or off without needing to know its current state.

The SelectionFontName property takes a string, the name of a font, rather than a boolean. The SelectionFontSize property takes an integer, which sets the text size in points.

The Password Field is very similar to the Text Field, with one glaring exception: the user can't see the text that they enter. The Password Field masks the characters entered by replacing them with bullets, in order to preserve password privacy. The contents are still accessible to your application; they're just hidden for the end user. In reality, the Password Field is a Text Field with its Password property set to True. It is provided as a separate control for your convenience.

The Combo Box allows the user to choose from several predetermined text values or to enter one of their own choosing. These predetermined options are set up using the Combo Box's AddRow method. For example, imagine you had a Combo Box for the user to select their grade level in school. You could use the Combo Box's Opening event to create a few common options:

```
    Me.AddRow("6")
    Me.AddRow("7")
    Me.AddRow("8")
    Me.AddRow("9")
    Me.AddRow("10")
    Me.AddRow("11")
    Me.AddRow("12")
```

The Combo Box allows the user to choose one of the text values from a list or to enter their own by treating it as a Text Field.

The AddRow method used in the last code example allows you to add choices to a Combo Box. You will see similar methods in other controls later in this chapter. AddRow takes a string and adds a new row to the end of the list of choices in a Combo Box. If you should need to add a new row in a certain location, you can use the AddRowAt method. AddRowAt takes an integer indicating where in the list you'd like to add the new row as its first parameter, followed by a string with the text you'd like to add (as with arrays, the rows in a Combo Box are zero-based).

If you should need to clear out all of the rows from a Combo Box, it also has a method called RemoveAllRows, which does exactly what its name indicates.

While it's not technically an input control, this is a good time to learn about the Label, a control whose function is primarily decorative. Its purpose is to serve as a label for other controls, such as

Text Fields, Text Areas, and others. Because of this, its Caption is also very important. You used labels in the sample project from Chapter Two. In a well designed application, many of your controls will have corresponding Labels.

6.5 Buttons

In this section, you will learn about different types of buttons. You may wonder why you would have more than one kind of button, but each serves a slightly different purpose. Choosing the right interface element for the right task is a fundamental part of user interface design.

Xojo offers several types of buttons, but you will only learn about three of them in this section: Button, Bevel Button, and Segmented Button. The most important commonality along these buttons is the Pressed event. Simply put, the Pressed event is called when the button is pressed.

As with the input controls discussed above, these buttons also have properties related to their size and position: Left, Top, Width, and Height. You will find these to be a common among all controls. Also, each of these buttons has a Name property; again, the button's Name is how you refer to the button in your code. As with all controls, it is best to give your buttons meaningful names that are related to their purpose, such as CancelButton, SendEmailButton, etc.

The Button is the simplest of these controls, and is one of the most common interface elements. In addition to its Name property, it also has a Caption, which is the text that will be displayed on the button. You can also change the font, size, and style of the text in your Caption, but unless you have a compelling reason to do so, it's always best to leave those properties set to the default values; that way, the button will look and behave as it should on any operating system.

Although most interactions with Buttons are done with the mouse, it also has two properties related to the keyboard: Default and Cancel. If Default is set to True, the Button will respond to the enter key on your keyboard. If Cancel is set to True, it will respond to the escape key (as well as Command-Period on MacOS and Control-Period on Windows and Linux).

The Bevel Button, like the Button, also has a Caption. In fact, it can almost be used as a Button. But there are some differences. First, the Bevel Button does not have Default and Cancel properties, so it can't respond to the keyboard in the same way as a regular Button. Second, while you can make it behave like a Button, it doesn't quite *look* like a regular Button, so you need to be careful about where you use them.

Of the many properties that a Bevel Button has which a Button does not, an interesting one is the Value property. Value is a boolean, and when it is set to True, the Bevel Button will take on a darker, "pressed" appearance. This makes the Bevel Button useful as a toggle.

1) **In your Events project, add a Bevel Button to Window1.**

2) **In the Inspector, set the Bevel Button's ButtonStyle property to ToggleButton.**

3) **Give your Bevel Button a Caption of your choosing.**

4) **Run your project.**

5) **Click the Bevel Button. Notice how the appearance changes.**

6) **Quit your application**

In your code, you can determine whether a Bevel Button is in its "pressed" state or not by accessing its Value property. If Value is True, the BevelButton is pressed.

A common use for a BevelButton that toggles is to set styles for text.

7) **In your Events project, add a Text Area to Window1.**

   Name the Text Area StyleDemoField. Make sure its AllowStyledText property is set to true.

8) **Change your Bevel Button's Caption to "Bold" and set its Name to "BoldButton".**

9) **Add another Bevel Button with the Caption "Italic" and the Name "ItalicButton".**

10) **In StyleDemoField's SelectionChanged event, add the following code:**

```
BoldButton.Value = Me.SelectionBold
ItalicButton.Value = Me.SelectionItalic
```

11) **In BoldButton's Pressed event, add this code:**

```
StyleDemoField.SelectionBold = Me.Value
```

12) **In ItalicButton's Pressed event, add this code:**

```
StyleDemoField.SelectionItalic = Me.Value
```

13) Run your project.

14) Type some text into your Text Area.

You should be able to use your Bold and Italic buttons on selected text to modify its style. In addition, when you click on or select styled text, the Bold and Italic buttons should toggle to reflect the text that you have selected.

15) Quit your application.

As you can see from this example, by using the events and properties of your controls, you can begin to build some complicated interactions. Based on this example, it wouldn't take much more work to add more styles, as well as font choices and text sizes.

The Segmented Button has a Pressed event, too, but it has a significant difference from that of the Button and Bevel Button. The Segmented Button's Pressed event provides you with a parameter: segmentIndex As Integer. This is because the Segmented Button is used for making choices, not just a simple action like a Button. More specifically, the Segmented Button is intended for situations where the user must choose among two or more mutually exclusive choices. One and only one choice can be made. The segmentedIndex provided to you in the Pressed event tells you which choice the user made, as indicated by which Segment of the Segmented Button the user clicked on.

A Segmented Button is made up of an array of Segments. You can get the segment the user pressed by passing the segmentIndex parameter to the SegmentedButton's SegmentAt method. You can then determine specifically which segment was pressed by using the Segment's Title property. If this sounds confusing, perhaps a code example is in order.

16) In your Events project, add a Segmented Button to your window.

If your window is getting too crowded, feel free to delete the controls from the older examples.

By default, your Segmented Button will have two items: One and Two. You can edit these or add more using the Inspector, but for now, stay with the default.

17) In the Segmented Button's Pressed event, enter the following code:

```
Var button As Segment
button = Me.SegmentAt(segmentIndex)
MessageBox(button.Title)
```

Before you run your project, let's examine the code. The first line creates a variable called button, with a data type of Segment. Remember from above that the Segmented Button contains an array of Segments, so you'll use this variable to store the item that's chosen by the user. That's what happens in the second line: you use the segmentIndex parameter provided by the Pressed event (the segmentIndex indicates which segment the user chose) to pull the appropriate Segment from the Segments array. Finally, you present the text of the chosen item to the user by displaying its Title property in a message box.

18) **Run your project.**

19) **Switch back and forth between the segments.**

You should see a message box with the item name on each change.

20) **Quit your application.**

6.6 Pickers

Another category of controls is known as pickers. These controls are used to allow the end user to set options and make choices. In this section, you will learn about Checkbox, Radio Groups and Radio Buttons, Sliders, Up Down Arrows, and Popup Menu. As with the other controls discussed in this chapter, these controls have the usual group of properties related to size and position: Left, Top, Width, and Height.

The Checkbox is best used when the end user needs to turn an option or setting on or off. You've most likely encountered these before in applications or even in web forms. The Checkbox has a Name property, which is how you refer to the Checkbox in code, and a Caption property, which is the text label presented to the user.

The most important property of the Checkbox is its Value property. When the Checkbox is checked, its Value property is True. When the CheckBox is unchecked, its Value property is false.

The Checkbox has a ValueChanged event, which is called when the user checks or unchecks it.

A Radio Group is similar to the CheckBox, but the Radio Buttons it contain can't be unchecked. At least, not by the user. A Radio Group contains a set of mutually exclusive choices, as Radio Buttons. Selecting one deselects all of the others.

1) **Create a new Xojo project and save it as "Options".**

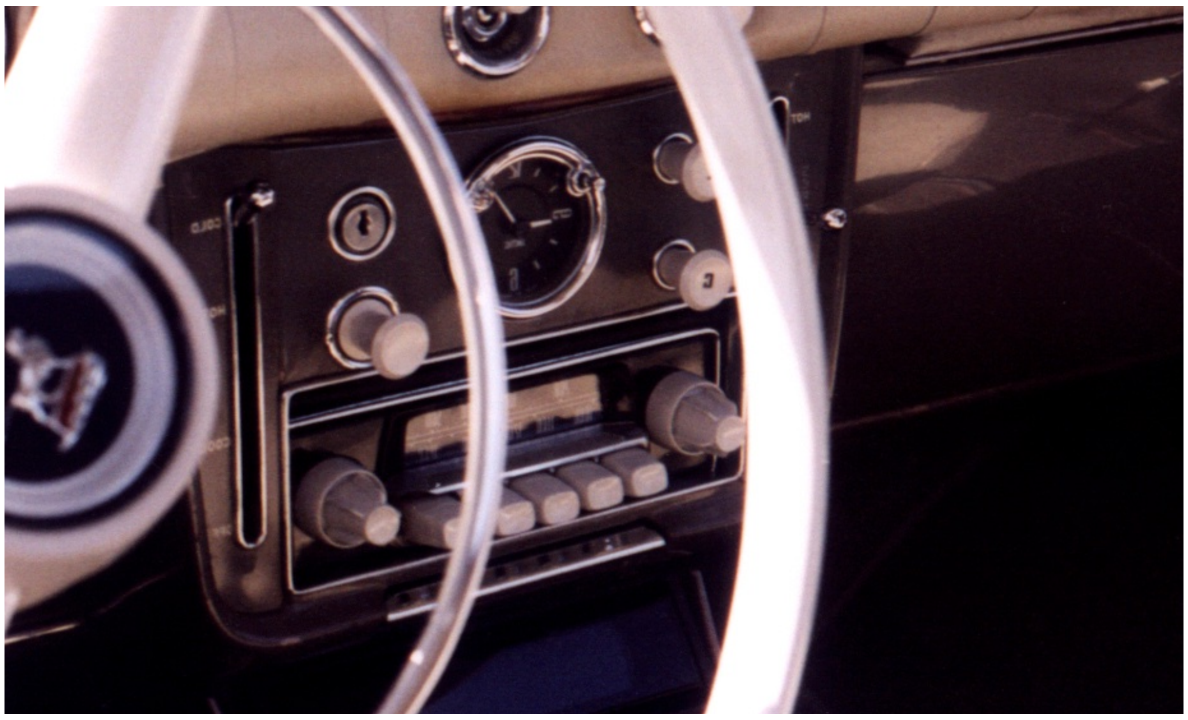
2) **In Window1, add a Radio Group.**

   By default you'll get a Radio Group with two Radio Buttons.

3) **Run your project.**

4) **Notice that when you select one Radio Button, the other is deselected.**

5) **Quit your application.**



Suppose you wanted two distinct sets of choices. Imagine you were writing an application for taking sandwich orders, and each person ordering could order one meat and one cheese. RadioButtons might be a good way to design this interface.

Go back to your Options project.

6) **Click on the Radio Group and then click the "pencil" icon next to Initial Value property.**

In the window, enter "Ham", "Turkey", "Salami", "Swiss", "Cheddar", and "Provolone" on one per line and click OK. Resize the Radio Group so you can see all its values.

7) **Run your project.**

Do you see the problem? Selecting a sandwich meat deselects all of the cheese options, and selecting a cheese deselects all of the meat options. In a real world example, you may wind up with some frustrated customers.

8) **Quit your application.**

Why are they called "radio buttons" anyway? On early car radios, you were given five or six buttons, each of which you could assign to one of your favorite radio stations (remember, this was before the age of iTunes, Sirius, and Spotify). Because you could only listen to one station at a time, when you pushed one button in, any previously selected buttons would "pop out" and become unselected. Radio buttons on the computer work similarly, in that they only allow you to have one selected at a time.

If you want your radio buttons to be independent of others, use a new Radio Group.

The Slider is a great way to allow your user to select a numeric value within a certain range, for example, if you need the user to select a number between 1 and 100. To set these limits, use the Slider's MinimumValue and MaximumValue properties. In the Inspector, you may notice another property listed between them, called Value. This is the numeric value that the Slider will have as a default until the user changes it or until your code changes it.

1) Create a new Xojo project and save it as "Slider".

2) Add a Text Field to Window1.

3) Add a Slider to Window1.

In the Inspector, set the Slider's Minimum Value to 1, its Maximum Value to 100, and its Value to 50. Set the Slider's AllowLiveScrolling property to True.

4) Add the following code to the Slider's ValueChanged event:

```
TextField1.Text = Me.Value.ToString
```

5) Run your project.

6) Change the value of the Slider and watch as its Value is updated in the Text Field.

7) Quit your application.

In addition to responding to the user, you can set the Value of a Slider in your code as well, using this syntax:

```
MySlider.Value = 50
```

Another control used to solicit a numeric value from your user is Up Down Arrows. These are small upward and downward facing arrows. They do not have a Minimum, Maximum, or Value property, so if you need such features, you will either need to manage them yourself or use a Slider.

Up Down Arrows have two important events to note: UpPressed and DownPressed. The UpPressed event is called when the user presses the upward facing arrow, and the DownPressed event is called when the user presses the downward facing arrow.

1) **Create a new Xojo project and save it as “Arrows”.**

2) **Add a Text Field to Window1**

3) **Add an Up Down Arrows to Window1.**

4) **In the DownPressed event of your Up Down Arrows, add this code:**

```
Var i As Integer
i = TextField1.Text.ToInteger
i = i - 1
TextField1.Text = i.ToString
```

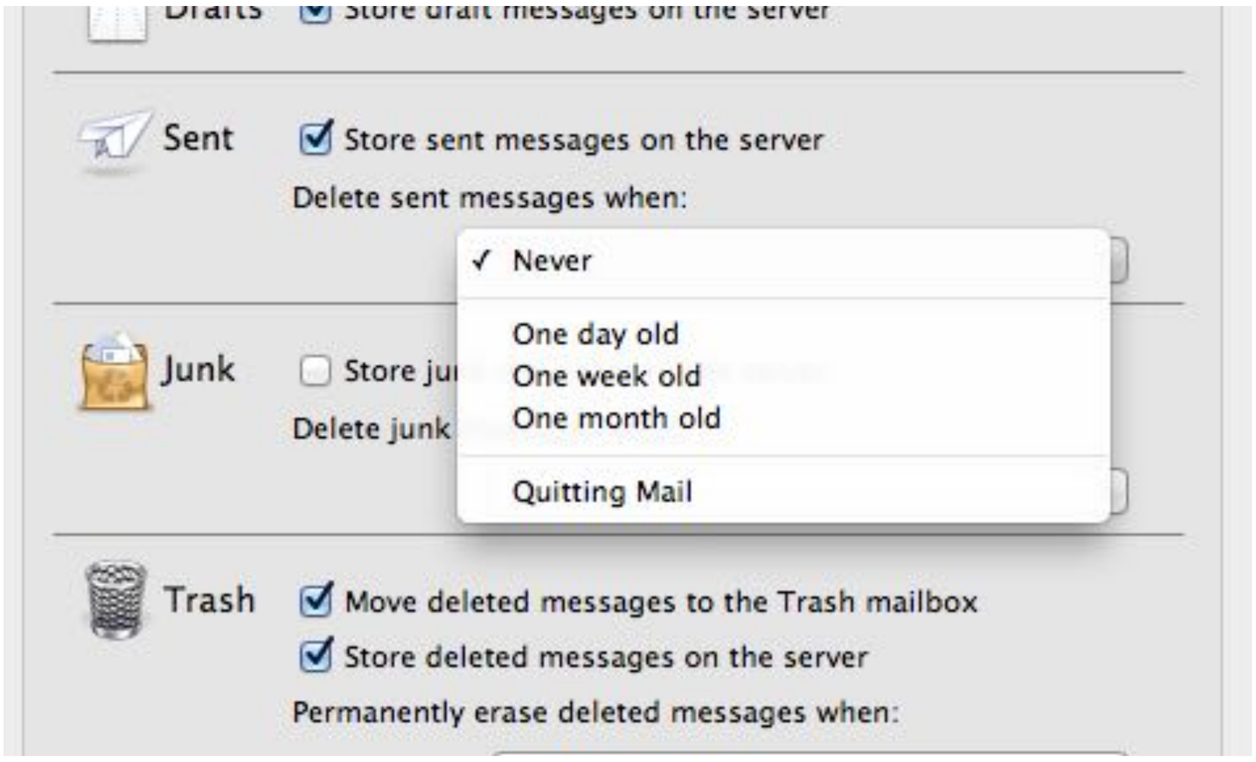
5) **In the UpPressed event of your Up Down Arrows, add this code:**

```
Var i As Integer
i = TextField1.Text.ToInteger
i = i + 1
TextField1.Text = i.ToString
```

6) **Run your project.**

7) **Use the UpDownArrows control to change the value displayed in the Text Field.**

8) **Quit your application.**


Drafts ✓ Store draft messages on the server

Sent ✓ Store sent messages on the server

Delete sent messages when:

Junk ☐ Store ju

Delete junk

Trash ✓ Move deleted messages to the Trash mailbox

✓ Store deleted messages on the server

Permanently erase deleted messages when:

The final control in the “pickers” category that will be covered is the PopupMenu. This is easily the most complicated of the group, but it is also a powerful control. As with all other controls, it has a Name property that you can set in the Inspector; this is the name you will use to refer to it in your code.

Most of the other properties of the PopupMenu that you will use on a regular basis will be accessed through code. The property you may use most often is the SelectedRow property. This property returns the text of the selected row in the PopupMenu.

But before you will have a value to work with, you'll need to add some rows to your PopupMenu. This is done using the AddRow method.

1) **Create a new Xojo project and save it as "Popup".**

2) **Add a Popup Menu to Window1.**

3) **In the Popup Menu's Opening event, add the following code:**

```
Me.AddRow("Popup Menu")
Me.AddRow("Text Area")
Me.AddRow("Up Down Arrows")
Me.AddRow("Window")
```

4) **Add this code to the Popup Menu's SelectionChanged event:**

```
MessageBox("You have chosen " + item.Text + ".")
```

The SelectionChanged event is called when the user makes a selection from the Popup Menu.

5) **Run your project.**

6) **Choose an item from the Popup Menu. You should see a message box with the item you've selected.**

7) **Quit your application.**

As you can see in the example above, the AddRow method takes a string as its parameter. This string is the text that will appear for that menu item. AddRow will always add a menu item to the end of the list. If you need to add a menu item at a specific place, you can use the AddRowAt method. AddRowAt takes two parameters: first, an integer indicating its position in the list (again, this list is zero-based), and then a string that will appear for that menu item.

The Popup Menu also features a RemoveRowAt method, which takes an integer indicating the number of the item you wish to remove. As with the ComboBox, the Popup Menu also includes a RemoveAllRows method.

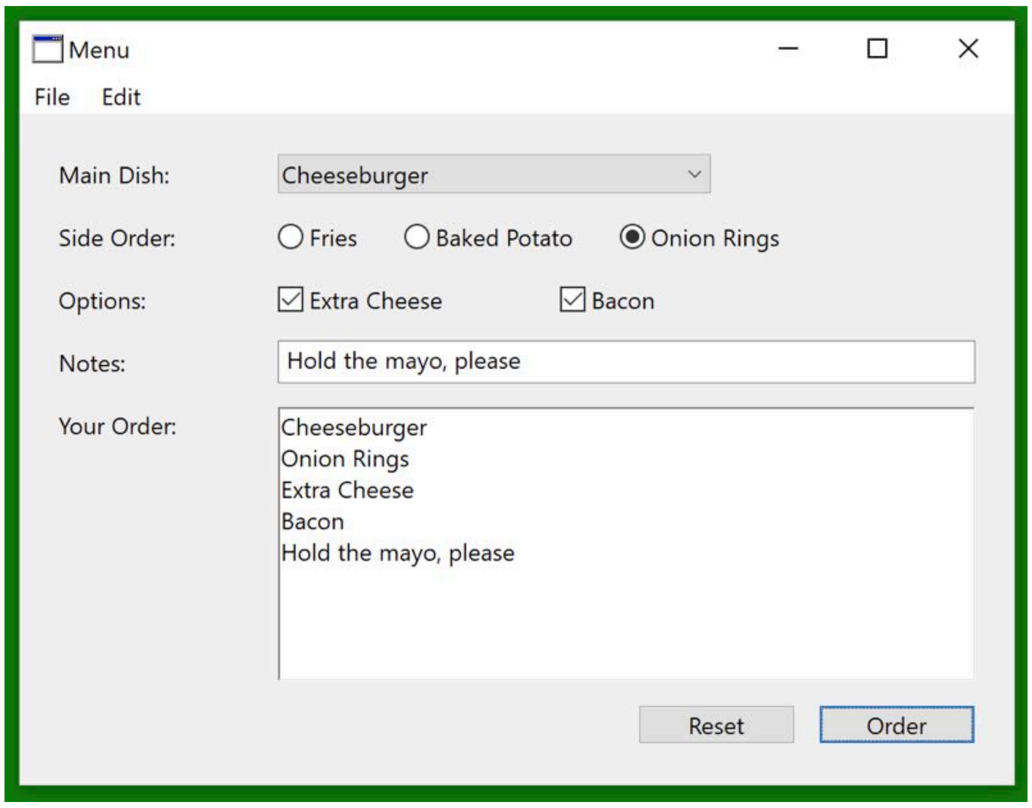
When the user makes a choice, the Popup Menu's SelectedRowIndex is set. The SelectedRowIndex is an integer (starting at 0 for the first row) that indicates the number of the item that was selected. To access the text associated with that item in the menu, use the PopupMenu's SelectedRow property we we did before.:

```
    myChoice = MyPopupMenu.SelectedRow
```

6.7 Hands On With Controls

For this chapter's sample project, you are going to build an electronic meal menu for a fictional restaurant that takes orders through computers rather than hiring waiters and waitresses. Your end user will need to be able to choose a main dish, choose a side order, select options, and leave notes for the cook.

Your application might look something like this:



Hold the mayo, please

```
Cheeseburger
Onion Rings
Extra Cheese
Bacon
Hold the mayo, please
```

1) **Create a new project in Xojo and save it as "Menu".**

2) **Change Window1's title to something meaningful, such as "Menu" or "Order Here".**

3) **Add a Label to Window1 and change its Text property to "Main Dish:".**

4) **Add a Popup Menu near that Label. Change its name to "MainDishMenu".**

5) **Add the following code to MainDishMenu's Opening event:**

```
Me.AddRow("Hamburger")
Me.AddRow("Cheeseburger")
Me.AddRow("Pizza")
```

If you want to get creative with the food options, feel free to do so!

6) **Add another Label to Window1 and change its Text property to "Side Order:".**

7) **Add a Radio Group (name it SidesGroup) and set its Horizontal property to ON.**

Click on Initial Value "pencil" and enter "Fries", "Baked Potato", and "Onion Rings" in the editor window, one per line.

8) **Add another Label to Window1 and change its Text property to “Options:”.**

9) **Add two Checkboxes to Window1, called “CheeseCheckBox” and “BaconCheckBox”. Set their Captions to “Extra Cheese” and “Bacon”.**

10) **Add another Label to Window1 and change its Text property to “Notes:”.**

11) **Add a Text Field next to that Label and change its Name to “NotesField”.**

12) **Add another Label to Window1 and change its Text property to “Your Order:”.**

13) **Add a Text Area near that Label and change its Name to “OrderArea”.**

14) **Add two Buttons near the bottom of Window1. Change their Names to “ResetButton” and “OrderButton”, and change their Captions to “Reset” and “Order”.**

15) **Set OrderButton’s Enabled property to OFF.**

This will prevent users from placing their orders without choosing a main dish.

16) **In MainDishMenu’s SelectionChanged event, add this code:**

```
If Me.SelectedRowIndex = -1 Then
  OrderButton.Enabled = False
Else
  OrderButton.Enabled = True
End If
```

17) **In ResetButton’s Pressed event, enter this code to reset all of the controls on the window:**

```
OrderArea.Text = ""
MainDishMenu.SelectedRowIndex = -1
SidesGroup.SelectedIndex = -1
CheeseCheckBox.Value = False
BaconCheckBox.Value = False
NotesField.Text = ""
```

18) **Add a method to Window1 called “CompileOrder”. Its code follows:**

```
Var theOrder As String
If MainDishMenu.SelectedRowIndex <> -1 Then
  OrderArea.Text = ""
  theOrder = MainDishMenu.SelectedRowValue + EndOfLine
  If SidesGroup.SelectedItem <> Nil Then
    theOrder = theOrder + SidesGroup.SelectedItem.Caption + EndOfLine
  End If

  If CheeseCheckBox.Value Then
     theOrder = theOrder + CheeseCheckBox.Caption
     theOrder = theOrder + EndOfLine
  End If
```

```
      If BaconCheckBox.Value Then
        theOrder = theOrder + BaconCheckBox.Caption
        theOrder = theOrder + EndOfLine
      End If
      theOrder = theOrder + NotesField.Text
      OrderArea.Text = theOrder
    End If
```

19) Add this code to OrderButton's Pressed event:

```
    CompileOrder
```

20) Run your project.

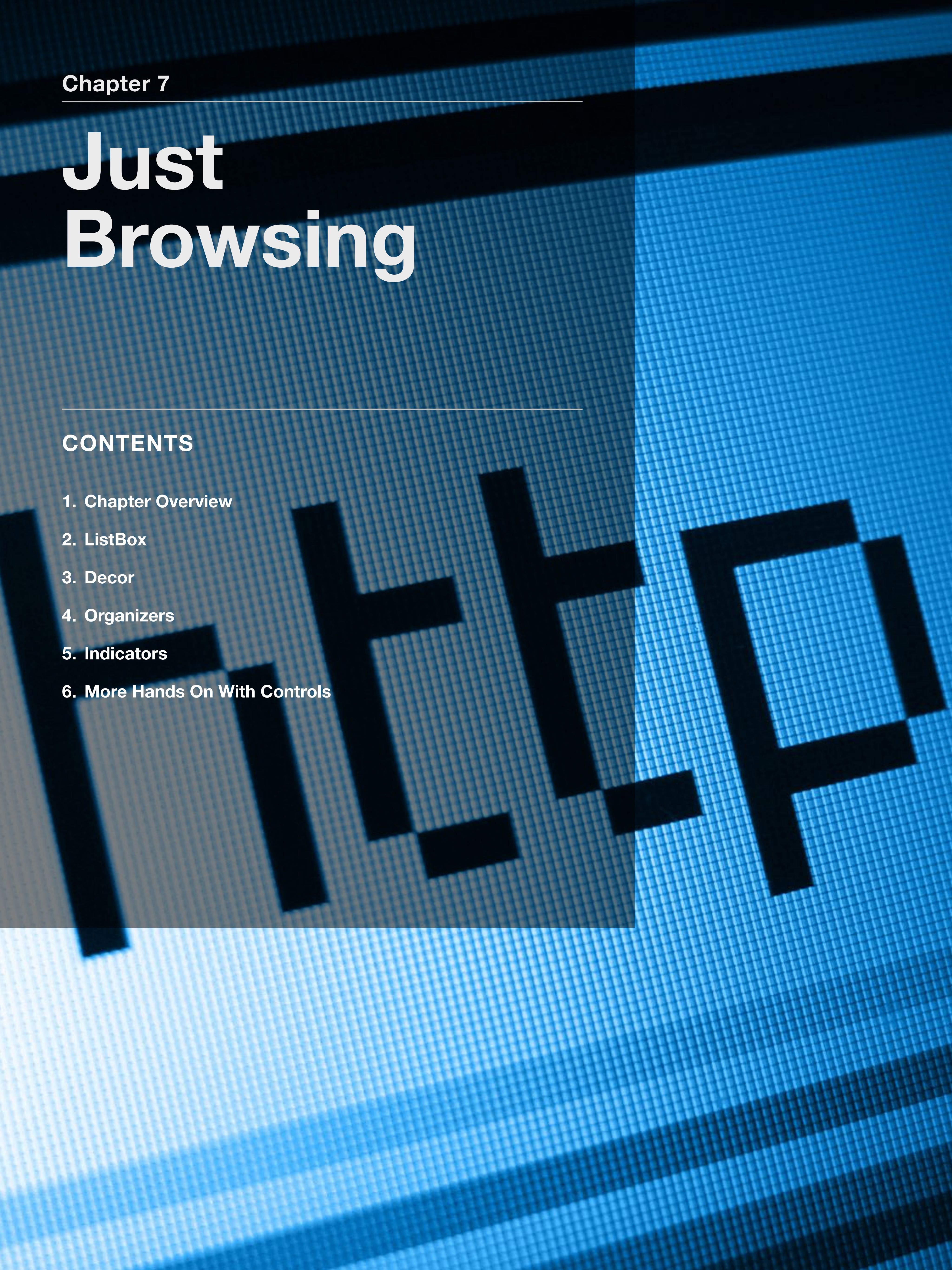
21) Place an order and choose your options.

22) Quit your application.

Obviously, this sample project won't place any real orders for actual food. Its goal is to show you a combination of controls working together in one interface. For extra practice, try refining the user interface of this project by using GroupBoxes and SegmentedButtons.

Chapter 7

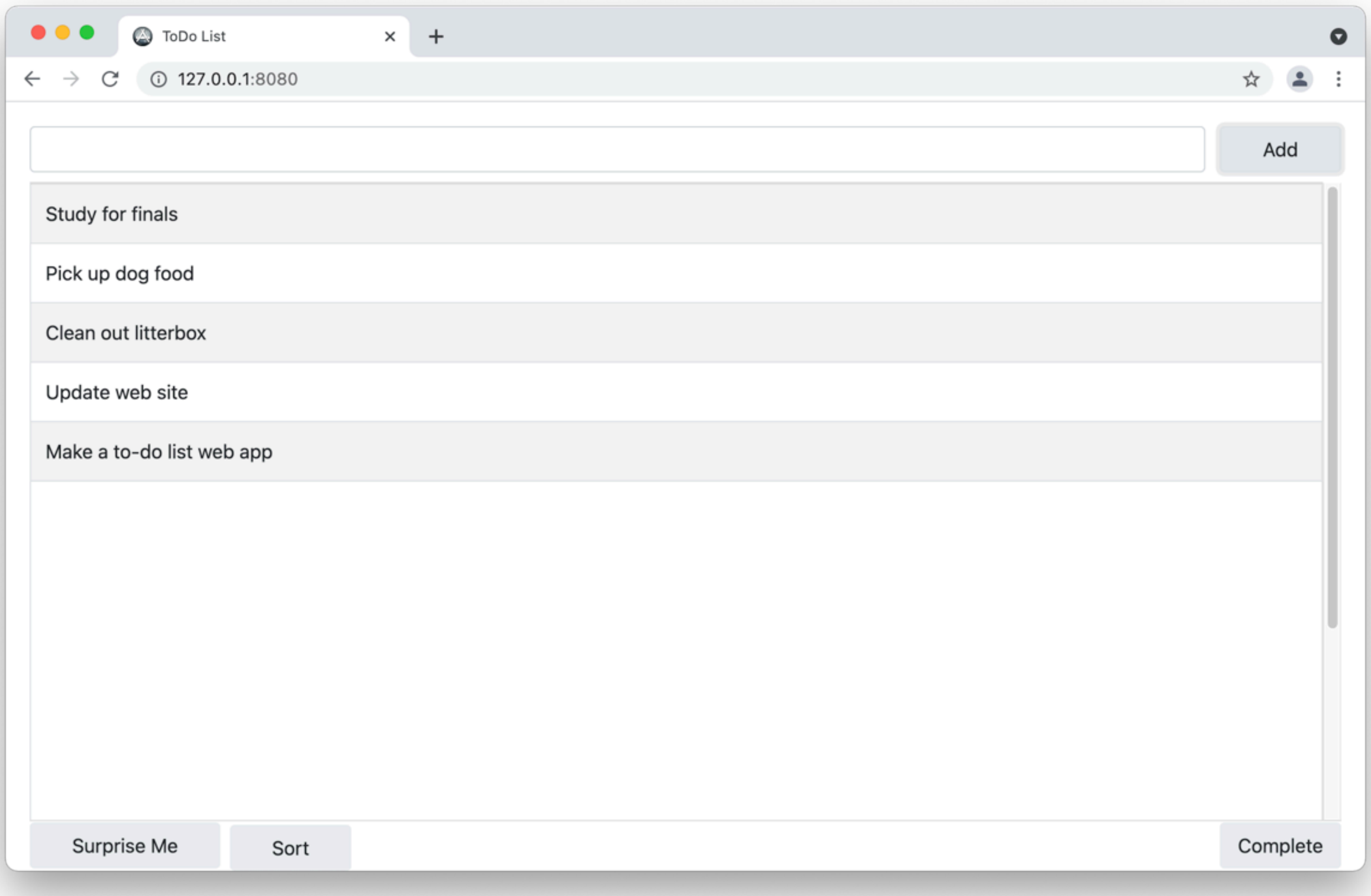
Just Browsing



CONTENTS

7.1 Chapter Overview

This chapter is essentially a continuation of Chapter 6. In it, you will learn about some other controls that are available to you, and you will build a minimal but functional web browser.



Study for finals
Pick up dog food
Clean out litterbox
Update web site
Make a to-do list web app

7.2 ListBox

One major control that has not yet been covered is the List Box. The List Box is an extremely useful control with many events and properties. This section will provide an introduction to it.

You can think of the List Box as a table. In fact, in some languages, it is known as a table or grid. It contains rows and columns, and the intersection of a row and a column is called a cell, just like in a spreadsheet. The contents of the List Box can be scrolled, either vertically or horizontally.

As with the other controls in this chapter, the List Box features the usual set of size and position properties. A quick glance at the Inspector, however, reveals many more properties. Two important properties of the List Box are ColumnCount and HasHeader. ColumnCount is an integer indicating how many columns it has. The columns themselves are in a zero-based list, so a three column List Box will have column 0, column 1, and column 2. HasHeader is a boolean that determines whether a header row is displayed. A header row is visually different from the other rows, and it also maintains its position at the top of the List Box while the other rows are scrolled.

The text that's contained in the header cells can be changed by using the Inspector to the set the InitialValue property. Enter your headings separated by tabs. If you enter text on a second line, new rows will be added to the List Box.

The List Box also has some events and methods that may look familiar from other controls. For example, you saw that the Popup Menu and the Combo Box both have a method called AddRow. The List Box has that method as well. As with the other controls with this event, AddRow takes a string as a parameter. That string will be added to the end of the ListBox, in the first column.

This may lead you to wonder how you populate the other cells in that row. Here's an example of the Opening event of a three column ListBox:

```
    Me.AddRow("Dylan")
    Me.CellTextAt(Me.LastAddedRowIndex, 1) = "Bob"
    Me.CellTextAt(Me.LastAddedRowIndex, 2) = "Musician"
```

That code would add one row with three cells. Add a few more rows, and it might look something like this:

Dylan	Bob	Musician
Cook	Tim	Businessman
Bolt	Usain	Athlete

A List Box's cells are stored in a two dimensional array called Cell (remember, since this is an array, these are zero based). So to update the contents of the cell at Row 1, Column 2 ("Businessman"), you could use this code:

```
    Me.CellTextAt(1, 2) = "Entrepreneur"
```

You may notice in the code just above that you used a property called LastAddedRowIndex. This property is the zero-based number of the last row that was added to the List Box. So using LastAddedRowIndex immediately after AddRow will always allow you to add data to cells beyond the first column.

In addition to the AddRow method, the List Box also supports the AddRowAt method. As with the Popup Menu and Combo Box, AddRowAt takes two parameters: an integer indicating its position in the list, and a string to add to the List Box. LastAddedRowIndex works with AddRowAt the same way it works with AddRow, so it is safe to use no matter how you add data to the List Box.

When the user selects a row in the List Box, its SelectionChanged event is triggered. This event doesn't provide you with the currently selected row, but as you saw in previous chapters, you can access this information using the SelectedRowIndex property. SelectedRowIndex is zero-based, so the first row has an index of zero. If no row is currently selected, then SelectedRowIndex will be -1.

The ListBox also has a DoublePressed event. In many applications that use a List Box, a single click (which triggers the SelectionChanged event) selects a row, while a double click will open a new window for editing or viewing its contents (such as double clicking on an email message in your inbox or double clicking a song in iTunes to have it start playing). In this event, you can use the SelectedRowIndex to determine which row was double-clicked. This has been just a short introduction to the ListBox. You will learn more about it in future chapters.

7.3 Decor

A glance at the Decor section of the Library in Xojo will reveal quite a few controls in this category. In this section, you will learn about the Group Box and the Canvas.

As implied by its name, the Group Box is used to group controls together for aesthetic reasons, such as simply giving your application a more pleasant design. You will often see Group Boxes used in applications' preferences and settings windows to arrange sets of controls into categories.

To organize controls inside a Group Box, add the Group Box to the Window first, then drag the control on top of it. The Group Box's border will turn red, indicating that it has become the "parent" control of the control. Parenting goes further than that, as well; the Group Box's parent is the Window itself.

As far as code execution goes, the Group Box doesn't actually do all that much. In terms of design, it's useful for containing other controls, as with the RadioButton example above. It is capable of holding any other control as well. Also important in the design of your application is setting the GroupBox's Caption property.

The Canvas is one of the most powerful controls offered by Xojo. In this section, you will only scratch the surface of what it can do. You will learn about some of its capabilities in Chapter Ten. In fact, in this section, you will only look at one event in the Canvas: the Paint event.

The Paint event provides you with a parameter: g as Graphics. You can ignore the areas() parameter for now. You will learn much more about the Graphics class in Chapter Ten, but this will provide a short introduction.

1) **Create a new Xojo project and save it as “Canvas”.**

2) **Add a Canvas to Window1.**

3) **Add the following code to Canvas1’s Paint event:**

```
g.DrawText("Hello!", 20, 20)
```

4) **Run your project.**

   When Window1 appears, it should say, “Hello!” on it.

5) **Quit your application.**

6) **Change the code in the Paint event to this:**

```
Var d As DateTime = DateTime.Now
g.DrawText(d.ToString, 20, 20)
```

7) **Run your project.**

   This time, the date and time will appear on the window.

8) **Quit your application.**

9) **Change the code in the Paint event to this:**

```
g.DrawRectangle(20, 20, 20, 20)
g.FillRectangle(40, 40, 40, 40)
g.DrawOval(60, 60, 60, 60)
```

10) **Run your project.**

   You should see several shapes drawn on the window.

11) **Quit your application.**

By now, you may be wondering what’s happening. The Graphics object provided by the Paint event of the Canvas allows you to do almost anything, from displaying text to drawing shapes and pictures. This may not mean much now, but when this book discusses graphics, printing, and subclasses, you will see that you can even create your own custom controls with the Canvas.

7.4 Organizers

This section covers the Tab Panel and the Page Panel. These controls, like the Group Box, are useful for grouping other controls into logical units. But they offer the advantage of hiding and showing these groups of controls as well.

You’ve most likely already encountered the Tab Panel in other applications, especially in their preferences and options windows. The Tab Panel, as its name implies, it used to show certain groups of controls while hiding others. The hidden controls remain active and accessible to your code, but they become invisible to the end user.

1) **Create a new Xojo project and save it as “Tabs”.**

2) **Drag a Tab Panel onto Window1.**

3) **In the Inspector, click the Edit button next to Panels under Appearance.**

4) **Change Tab0 to “Global Settings”.**

5) **Change Tab1 to “User Settings”.**

6) **Back in the window editor, with the first tab of the Tab Panel highlighted, add some controls of your choosing.**

7) **Switch to the second tab of the Tab Panel and add a different group of controls.**

8) **Run your project.**

9) **Switch back and forth between the tabs and notice how only the controls on the selected tab are displayed.**

10) **Quit your application.**

If you need to know which tab has been selected, the TabPanel does provide a PanelChanged event. You can find out which tab panel is selected with the SelectedPanelIndex property.

The Page Panel is similar to the Tab Panel, except that navigation UI is not provided for you. While the Tab Panel provides tabs across the top to allow the user to switch easily, you must provide your own navigation UI for the Page Panel. This makes the Page Panel very useful for a “wizard” type of interface that walks a user through several steps in a certain order.

Navigation is done by setting the Page Panel’s SelectedPanelIndex property. The SelectedPanelIndex property is an integer indicating which panel is currently active (the list of panels is zero-based). The panels are set up by you when you design the window, in the same way that you set up panels for the TabPanel.

There are several ways to design navigation for your Page Panel. The easiest way is to have a Button outside the Page Panel with code similar to this in its Pressed event:

```
PagePanel1.SelectedPanelIndex = PagePanel1.SelectedPanelIndex + 1
```

An additional button could also be provided to go back to the prior panel using this code:

```
PagePanel1.SelectedPanelIndex = PagePanel1.SelectedPanelIndex – 1
```

Another way to provide navigation is to add a dedicated button on each panel of your Page Panel, taking you to the next panel in its Pressed event. This can become difficult to manage as your application grows more complex, however.

A third way is to direct to the user to a specific panel based on their actions. For example, imagine you were building a “wizard” interface to walk the user through setting up an account with a social network. You might have the user choose between creating a new username and password or using an outside authentication system like Google or Facebook. If the user chooses to create a new account, you could set the Page Panel to take them to a panel with a sign up form. If they choose to use outside authentication, the Page Panel can show them a panel where they can enter their credentials for that outside service.

7.5 Indicators

One of the most important aspects of a well designed user interface is responsiveness. However, sometimes code simply takes a while to run. In these cases, it's best to provide your user with feedback to let him or her know what's going on. Often, if an application provides no feedback for more than a few seconds or doesn't seem to be doing anything, the user will assume that the application has frozen or crashed. If your application is processing a large data set, it can be very useful to display a Progress Bar or Progress Wheel. This indicates to the user that the application is still working.

The Progress Bar is used when you can quantify the length of time your application will take or the amount of items that need to be processed. The Progress Bar has a MaximumValue property that should be set to that number. In other words, if your application needs to process an array of 200 items, you should set the Progress Bar's MaximumValue to 200. The Progress Bar's Value, on the other hand, should reflect how far along the process is.

1) **Create a new Xojo project and save it as "Progress".**

2) **Drag a ProgressBar onto Window1.**

3) **In the Inspector, make sure the Progress Bar's MaximumValue is set to 100 and its Value is set to zero.**

4) **Drag a Timer onto Window1.**

   Notice that the Timer positions itself below the Window Editor (in the area called the "Shelf"). This is because it won't be part of the application's user interface. Timers will be covered in depth in a later chapter; for now, just know that a Timer will perform a given task after a specific amount of time has passed.

5) **Double-click on the Timer and in the Action event, enter this code:**

```
ProgressBar1.Value = ProgressBar1.Value + 1
```

6) **Run your project.**

The ProgressBar should begin to “fill up” from left to right, expanding slightly every second.

7) **Quit your application.**

When used in conjunction with Timers and Threads (both of which will be covered in a later chapter), the Progress Bar can be invaluable in providing your user with feedback about your application’s current state.

Many times, however, you will not know how many items need to be processed or how long a given task will take. For example, you may be waiting for data to arrive from an outside server, and the speed will be dependent on many factors outside of your control. In these cases, because it does not indicate how much time remains for a given task, a Progress Wheel is an appropriate way to tell your user that the application is still working.

Rather than having a MaximumValue and a Value, the Progress Wheel is made visible or invisible when needed. When it is visible it displays as a spinning wheel.

1) **Open up your Progress project in Xojo.**

2) **In Window1, delete your Progress Bar and Timer.**

3) **Add a Progress Wheel to Window1.**

4) **In the Inspector, set the Progress Wheel’s Visible property to OFF.**

5) **Add a Button to Window1.**

6) **Add this code in your Button’s Pressed event:**

```
ProgressWheel1.Visible = Not ProgressWheel1.Visible
```

7) **Run your project.**

8) **Click the Button to toggle the Progress Wheel’s visibility.**

9) **Quit your application.**

Like the ProgressBar, the ProgressWheel is much more valuable when used in conjunction with Timers and Threads.

Note that you can also use a Progress Bar to indicate a task with an unknown time period by setting its Indeterminate property in the Inspector to ON.

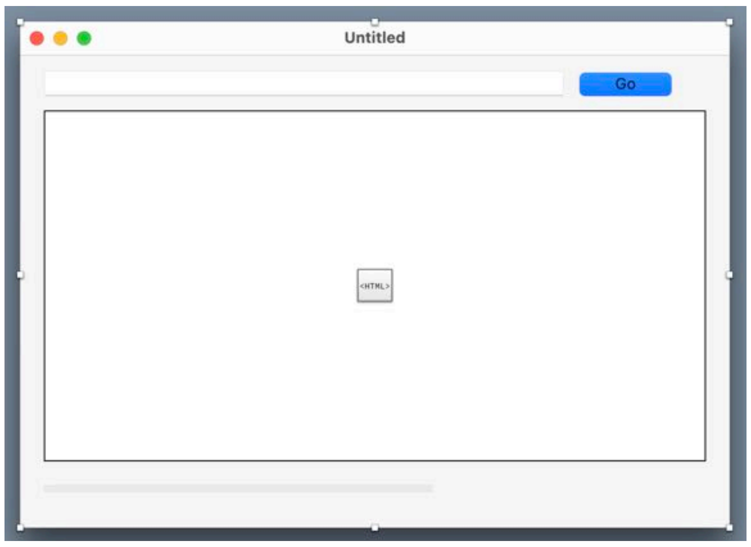
7.6 More Hands On With Controls

Because you haven't yet learned about Timers and Threads, there is another control that you can use quite effectively with the Progress Bar and the Progress Wheel: the HTML Viewer.

The HTML Viewer renders HTML, whether you provide the HTML in your code, from a local file, or as a URL to an outside website.

1) **Create a new project in Xojo and save it as "Browser".**
2) **Add a Text Field to Window1. Set its name to "AddressField".**
3) **Add a Default Button near AddressField. Set its Caption to "Go".**
4) **Add a Progress Wheel near the PushButton and set its Visible property to OFF.**
5) **Add an HTML Viewer to Window1. Set its name to "MainViewer". It should take up most of the space on the window.**
6) **Add a Progress Bar under MainViewer. Set its MaximumValue property to 100 and its Value property to zero.**
7) **Add a Label next to the ProgressBar and set its name to "StatusLabel". By default a Label's Text is set to "Untitled". Delete the Label's Text and leave it blank.**
8) **Select MainViewer. In the Inspector, look for the Locking section. Make sure the top, bottom, left, and right edges of the HTML Viewer are locked.**
9) **Using the Inspector, make sure that the Progress Bar and the Label are locked at the bottom and unlocked at the top.**

   Here is what your interface might look like in the Window Editor:



10) **In the Default Button's Pressed event, add the following code:**

```
MainViewer.LoadURL(AddressField.Text)
```

11) **Run your project.**

12) **Enter “https://en.wikipedia.org” into the AddressField and press the Go button. Note that on macOS, it’s important that you enter “https” and not just “http”.**

It may take a moment, but the Wikipedia website will load into the HTML Viewer. While it loads, however, the user doesn’t know what’s going on or if the app is doing anything at all.

By adding just a few lines of code to this project, you can make it much more responsive and keep the user better informed.

13) **Quit your application.**

14) **In MainViewer’s DocumentBegin event, add this code:**

```
ProgressWheel1.Visible = True
```

15) **In MainViewer’s DocumentProgressChanged event, add this code:**

```
ProgressBar1.Value = percentageComplete
StatusLabel.Text = "Loading " + URL
```

16) **In MainViewer’s TitleChanged event, add this code:**

```
Self.Title = newTitle
```

17) **In MainViewer’s DocumentComplete event, add this code:**

```
ProgressWheel1.Visible = False
ProgressBar1.Value = 0
StatusLabel.Text = ""
```

18) **Run your project.**

19) **As before, enter “https://wikipedia.org” into the AddressField and press the Go.**

This time, when the page loads, the Progress Wheel will tell you when the application is working, the Progress Bar will indicate how much of the page remains to be loaded, and the StatusLabel will tell you what the application is doing. If you have a fast internet connection, these updates may flash past you quite quickly.

20) **Quit your application.**

In above example, you implemented several of the HTML Viewer’s events. The first one was DocumentBegin, which provides you with a string called URL. This is the address of the page that the HTMLViewer is loading.

You also implemented the DocumentProgressChanged event. This event provides you the URL again, as well as an integer called PercentageComplete. PercentageComplete, as implied by its name, tells you how much of the page has been loaded. The fact that it's provided as a percentage is very useful, since that number is then very easy to use with a ProgressBar, which is exactly what you did above. You also updated StatusLabel's Value property with a message that the URL was loading.

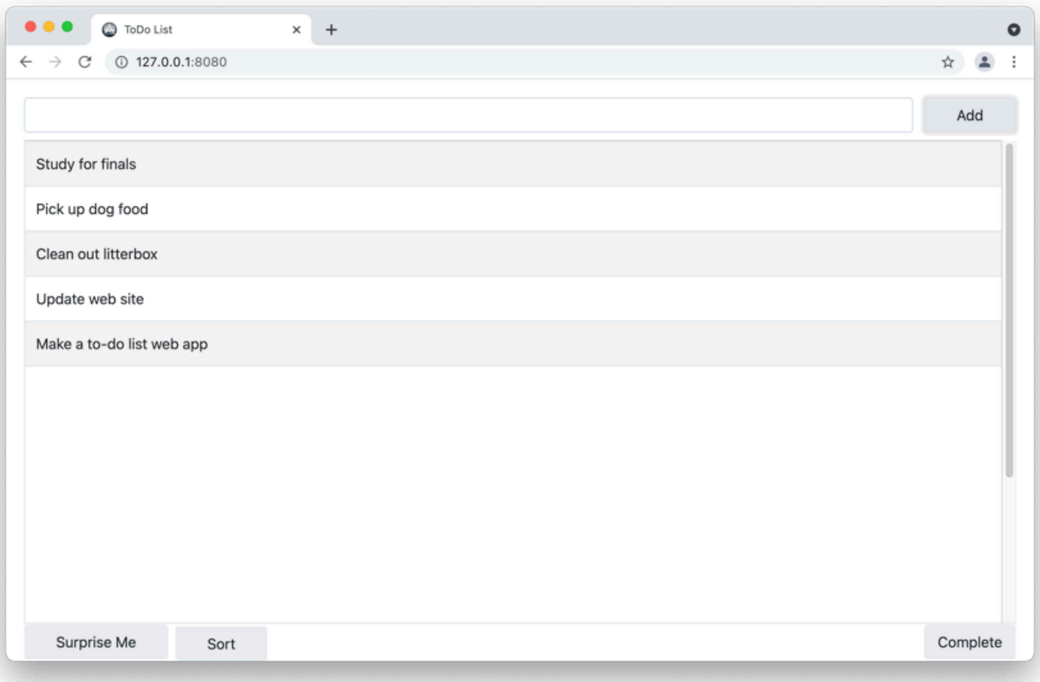
Another event you implemented was TitleChanged, which provides with a string called NewTitle. This is the title of the web page that is being loaded. In the example above, you changed the window's Title to match. That's what this line did:

```
    Self.Title = NewTitle
```

When you're writing code in a control on a window, you can refer to the window as Self (conversely, you can refer to the control whose code you're editing as Me). So to change the window's Title property to the name of the web page, set Self.Title to NewTitle.

Finally, you implemented the DocumentComplete event, which again provides the URL. This event is triggered when the page is finished loading. In your case, you used this event to make the ProgressWheel invisible again, since it's no longer needed to indicate activity. You also reset the ProgressBar's Value back to zero. And you set StatusLabel's Value property to an empty string.

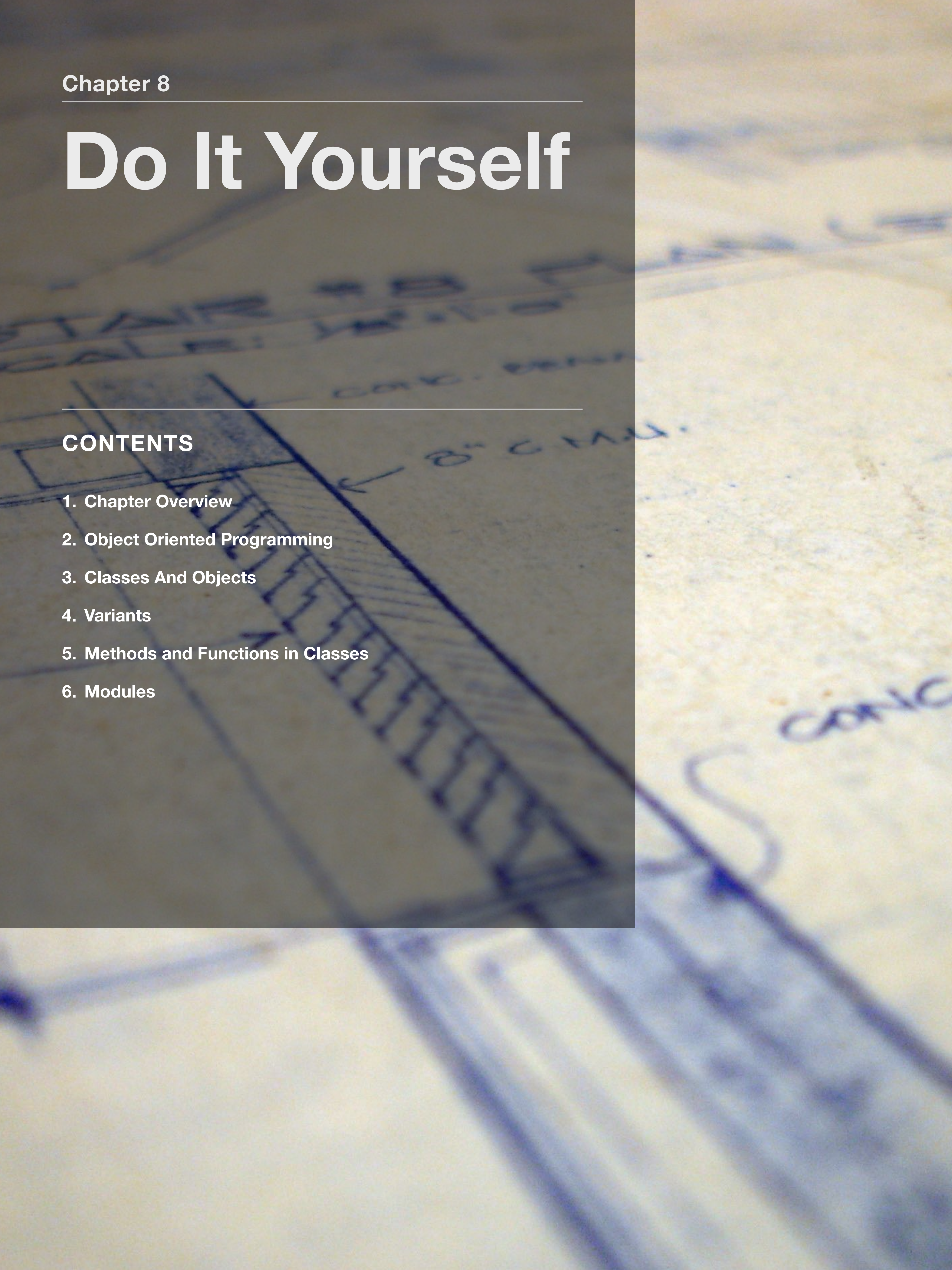
Congratulations! Believe it or not, you just built a web browser. Sure, it's missing some fundamentals like bookmarks, error checking, and tabs, but by implementing a few events in a few controls, you now have a workable, usable web browser:



Chapter 8

Do It Yourself

CONTENTS



8.1 Chapter Overview

So far, you have been introduced to data types, variables, events, controls, arrays, methods, functions, loops, and more. In fact, at this point, you have enough tools in your toolbox to build a sophisticated application, as long as you stick to the predefined data types provided by Xojo. But you will encounter times, and they will be many, when you need to define your own data types with unique properties and methods. For this, you use a class. A class is something that represents a real life object or an abstract idea. For example, you may have a class that represents a car (a real life object) or a bank transaction (an abstract idea).

In this chapter, you will learn to create and use your own classes, and you will also learn about modules, which provide you with a way to provide global data and methods to the rest of your application.

8.2 Object Oriented Programming

There are two basic ways to develop software: Procedural Programming and Object Oriented Programming.

Procedural Programming is best exemplified by the old BASIC, Pascal or C languages. The computer would run through each line of code, in order, and then stop running the app. This approach can certainly get things done, but it doesn't leave a lot of room for error, and it can be very difficult to add features and fix bugs.

Object Oriented Programming, on the other hand, takes a different approach. Essentially, you create objects and you teach them how to behave: how to respond to different events, how to display data, etc. These objects can interact with each other in almost any way you can dream up.

Xojo is an Object Oriented language, like Swift, C#, Objective-C, or Java. While you don't need to use all of the principles of Object Oriented Programming, it can help you create code that is more flexible and far easier to maintain.

A complete reference on Object Oriented Programming is far beyond the scope of this book, but this chapter and some of the following chapters will take advantage of the fact that Xojo is object oriented.

8.3 Classes And Objects

As mentioned above, a class is something that represents a real life object (something you can touch or point to) or an abstract idea (a concept or “intangible” idea). To illustrate a class that represents a real life object, you will create a class called Student.

Before you start writing code, stop and think about the attributes of a student. Some obvious things that come to mind are a first name, a last name, a middle name, a birthdate, and a grade level. Next, think about the data types that you will need to store this information. The first, last, and middle names are all strings. The birthdate is a date. For the sake of simplicity, let’s make Grade level an integer (assume that this is a secondary or intermediate school without a Kindergarten class).

A real life object’s attributes can be expressed as properties of a class. Your student class could have these properties:

Property	Data Type
First Name	String
Last Name	String
Middle Name	String
Birthdate	DateTime
Grade Level	Integer

You could continue to add properties, such as hair color, eye color, height, weight, and on and on. For now, this list of properties will be sufficient.

But property names are subject to the same rules as variable names (no spaces or punctuation except for the underscore), so a cleaned up list of properties would look like this:

Property	Data Type
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
GradeLevel	Integer

An example of a class that represents an abstract idea would be a course. A course is not a physical object, and you can’t touch it, but it’s certainly a concept that impacts your life as a student. Now think about the attributes of a course. A course has a title, an instructor, a room or location, and a subject, like math, social studies, science, etc. These can all be stored as string data. So your course class could have these properties:

Property	Data Type
Title	String
Instructor	String
Room	String
Subject	String

Again, you could continue to add more properties, but this will do for now.

Some of the data types you learned about in previous chapters are examples of classes that Xojo provides for you. Like those data types, classes must be instantiated, or created, using the New keyword, like so:

```
Var s As Student
s = New Student
```

Talking about classes in the abstract is all well and good, but they're not useful to your applications until you add them to your projects.

1) Create a new Xojo Desktop project and save it as "StudentInformation".

2) Go to the Insert menu and choose Class.

The class editor will appear. This is where you will enter a name for your class. Call this class "Student". The other fields (Super and Interfaces) can be left blank for now (you'll learn more about these topics in Chapter Thirteen).

3) With your Student class selected in the Navigator, go to the Insert menu and choose Property.

Your property needs a Name and a Type, and you can optionally set a default value and the property's scope. Name your property FirstName and set its Type to String. The Scope should be set to Public. You can leave Default blank for now.

4) **Repeat this process to add your other properties: LastName As String, MiddleName As String, BirthDate As DateTime, and GradeLevel As Integer.**

5) **Go back to the Insert menu and choose Class to add the next class. Name this class “Course”.**

6) **Add these properties to your Course class: Title As String, Instructor As String, Room As String, and Subject As String.**

7) **In the Navigator, select Window1 and choose Property from the Insert menu.**

This new property will allow you to add some Courses to your application so that you have some data to work with. The property’s name should be MyCourses() and its type should be Course. This will create an array of Courses that you can use. Notice that Course autocompletes in the type field.

8) **Create a new method in Window1 called “GenerateCourses”.**

9) **Add this code to GenerateCourses:**

```
Var c As Course
c = New Course
c.Instructor = "Mr. Smith"
c.Room  = "101"
c.Subject = "Science"
c.Title = "Biology"
MyCourses.Add(c)
c = New Course
c.Instructor = "Mrs. Jones"
c.Room  = "202"
c.Subject = "Mathematics"
c.Title = "Geometry"
MyCourses.Add(c)
c = New Course
c.Instructor = "Ms. Jackson"
c.Room  = "301"
c.Subject = "World Language"
c.Title = "Spanish III"
MyCourses.Add(c)
```

This code will create some sample Courses. Typically, this would be done from a database or other external data source, but for now, you can create them in code.

Feel free to add more Courses to the list if you’d like. Note that each time you use the line “c = New Course”, the old value is discarded from c and the variable is created anew. The value is not lost, however, since it has been added to the MyCourses array. Also note that the properties you added are available in Xojo’s dot notation and autocomplete.

10) **Add a List Box to Window1 and name it “CourseBox”.**

This List Box will be used to display a list of available courses.

11) **Create another method in Window1 called “ListCourses”.**

12) **Add this code to ListCourses:**

```
CourseBox.RemoveAllRows
For Each c As Course In MyCourses
  CourseBox.AddRow(c.Title)
Next
```

This method loops through your array of Courses (using a For...Each loop) and adds each title to the ListBox.

13) **In Window1’s Opening event, you’ll need to run both of those methods:**

```
GenerateCourses
ListCourses
```

14) **Run your project.**

You should see the Courses appear in the ListBox.

15) **Quit your application.**

It might be tempting at this point to think that the ListBox now contains information about your Courses. In a very limited sense, it does, but it only contains the title of each course. Given the title only, it would be difficult to look up other information about the course, unless you looped through the array looking for a match or moved the Course data from an array to a dictionary. And even then, you would have to make sure that you have no duplicate course titles.

The reason you don’t have all of the information about your Courses in the ListBox is because you have only used one of their properties. To have access to all of the Course information, you need to use an object.

If you think of your class as a blueprint, you can think of an object as the actual house.

A class is a description of something, and an object is the thing itself. Whenever you use the New keyword, you're creating a new object in your code. So in the code above, when you created some sample Courses, each of those was an object. In that example, you only grabbed the title from each Course object, but it's possible to store an object itself in the ListBox. Change the ListCourses method to this:

```
CourseBox.RemoveAllRows
For Each c As Course In MyCourses
  CourseBox.AddRow(c.Title)
  CourseBox.CellTagAt(CourseBox.LastAddedRowIndex, 0) = c
Next
```

Now the actual Course object is stored in a special part of the List Box called the CellTag.

8.4 Variants

The obvious question raised by the code change is: what is a CellTag? As you saw earlier, Every List Box has a two dimensional array of Cells. It also has a two dimensional array of CellTags. You can think of a CellTag as a "secret compartment" where you can store data.

While you can only store string data in a Cell (and that string is visible), a CellTag stores a data type called a Variant that is not visible. A Variant is a flexible data type that can store a string, an integer, a double, a datetime, a dictionary, or any other data type, even your custom Course objects. So the code above assigns each Course object to a CellTag in the List Box. Because of the way the code is structured, each Cell in your List Box will contain a CellTag with related course data, stored as a Variant.

This raises another question: how do you get data into and out of a Variant? Assigning data to a Variant is straightforward, and it matches the way it's done with other data types.

1) **Create a new Xojo project and save it as "Variants".**

2) **In Window1's Opening event, add this code:**

```
Var v As Variant
v = "Hello!"
MessageBox(v)
```

3) **Run your project.**

   You should see a message box that says, "Hello!". As you can see, for string data, using a Variant can be just like using other simple data types.

4) **Quit your application.**

5) **Change the code in Window1's Opening event to this:**

```
Var v As Variant
v = 123
MessageBox(v)
```

This time, instead of storing a string in the Variant, you've stored an integer. Recall from Chapter Two that you couldn't present an integer to the end user in a message box without first converting it using the ToString function. When you use a Variant, however, it converts the data for you automatically.

This can be extremely convenient, but it also be dangerous. Take this example:

```
Var first, second, third As Variant
```

```
first = 123
second = "456"
third = first + second
MessageBox(third)
```

What should be displayed in the message box? You could argue that the message box should display “123456” (if the string values of the Variants are combined), but you could also make a case for “579” (if the integer values are added). Change the code in Window1’s Opening event to the code above and try it for yourself. Were you surprised at the result?

This illustrates that while Variants are certainly powerful and useful, they should be used with great care and only when necessary. Storing an object in a List Box’s CellTag is a perfect example of when it’s necessary to do so. Even so, great care is still needed.

That care can be exercised by using the Variant’s properties to force your code to treat the data as a certain type.

6) Change the code in Window1’s Opening event to this:

```
Var first, second, third As Variant
first = 123
second = "456"
third = first.IntegerValue + second.IntegerValue
MessageBox(third)
```

The IntegerValue property of the Variant forces the computer to treat the data as numeric instead of string. The Variant also has a StringValue property, as well as BooleanValue, DataValue, and more.

Before you can use an object stored in a Variant, you must tell the computer what type of object it is. Going back to the StudentInformation example, you must tell the computer that the Variant stored in the CellTag is a Course object. Before doing so, you should make sure that the object is indeed a Course.

7) In the StudentInformation project, add this code to the DoublePressed event of CourseBox:

```
Var c As Course
If Me.SelectedRowIndex <> -1 Then
  If Me.CellTagAt(Me.SelectedRowIndex, 0) IsA Course Then
    c = Course(Me.CellTagAt(Me.SelectedRowIndex, 0))
    MessageBox(c.Instructor)
  End If
End If
```

This code may look very confusing at first, so let’s break it down. First you create a variable, c, which you’ll use to access your Course object. Second, you verify that the user has selected a valid row in the List Box by checking the SelectedRowIndex.

The next line uses the IsA operator to check if the Variant (the CellTag in Column 0 of the selected Row in the List Box) is actually a Course object. IsA returns a boolean value: true if the Variant is really of that data type and false if it is not.

Next, the code does a process called casting, which means it's saying that the CellTag in question should be treated as a Course object, stored in the variable you called c.

After that line, c is a Course object that you can use as you see fit. In example above, the Course's Instructor is displayed to the end user in a message box.

8) **Run your project.**

Try it out for yourself. When you double click on a Course in the List Box, you should see a message box showing the Instructor's name.

9) **Quit your application.**

8.5 Methods and Functions in Classes

In Chapter Four, you learned about methods and functions. In that chapter, you created those methods and functions as part of the window. You can also create method and functions that are part of your custom classes.

1) **Go back to your StudentInformation project.**

2) **With your Course class selected in the Navigator, choose Method from the Insert menu. Call the method “DisplayCourseInfo”.**

3) **Add this code to DisplayCourseInfo:**

```
Var info As String
info = "Course Title: " + Self.Title + " ("
info = info + Self.Subject + ")" + EndOfLine
info = info + "Instructor: " + Self.Instructor + EndOfLine
info = info + "Location: " + Me.Room
MessageBox(info)
```

This method will simply gather some information about the Course and display it to the end user in a message box.

4) **In CourseBox’s DoublePressed event, change this line:**

```
MessageBox(c.Instructor)
```

to this:

```
c.DisplayCourseInfo
```

5) **Run your project.**

6) **Double click on a Course name in the List Box to display information about that Course in a message box.**

7) **Quit your application.**

Methods and functions in a class don’t always have to display information in a message box. They can be as simple or as complex as you need them to be. They can also take parameters, just like any other method or function. The parameters they take can be of any data type, either a data type built into Xojo or a custom data type, such as the Student class you created earlier.

8) **Add a property to your Course class: EnrolledStudents() As Student.**

This is an array of students who are currently enrolled in the Course.

9) **Add a method to your Course class called “EnrollStudent”.**

This method takes one parameter: s As Student. This method’s job is adding the provided student to the EnrolledStudents array.

10) **Add this code to EnrollStudent:**

```
Self.EnrolledStudents.Add(s)
```

11) **Now add a method to your Student class called “Constructor”.**

There are two special names for methods and functions: Constructor and Destructor. If a class has a method called Constructor, that method will run as soon as the class is instantiated. In this case, as soon as you create a Student object with the New operator, the Constructor method will be run. Destructor is similar, but it runs when the object is destroyed rather than when it is created. The Constructor can take parameters.

12) **Give your Student Constructor two parameters: “fName As String” and “lName as String”.**

Remember that multiple parameters should be separated by commas. These two parameters will be used to set the FirstName and LastName properties of the Student.

13) **Add this code to the Constructor:**

```
Self.FirstName = fName
Self.LastName = lName
```

Now you have a way to create Students and a way to add those Students to a Course. There are two critical pieces remaining: first, a way to see which Students are enrolled in a Course, and second, a mechanism for the user to add Students to a Course (the method exists, but it’s not accessible to the user).

14) **Add another List Box to Window1 called “StudentBox”.**

15) **Add two Text Fields called “FirstNameField” and “LastNameField”.**

16) **Add a Button called “EnrollButton”.**

Set its Caption property to “Enroll”. The user will enter a first name and a last name in the Text Fields, then use the Button to add that student to the Course selected in the other List Box.

17) **Set EnrollButton’s Enabled property to OFF in the Inspector.**

18) **Add this code to CourseBox’s SelectionChanged event.**

```
If Me.SelectedRowIndex < 0 Then
  EnrollButton.Enabled = False
  Return
 End If
```

EnrollButton should only be clickable when a Course is selected. In a previous sample project, you used a multi-line If statement to turn a Button on or off, depending on what was selected. This is a simpler, shorter way to do it.

19) **Add this code after the line you just entered:**

```
EnrollButton.Enabled = True
FirstNameField.Text = ""
LastNameField.Text = ""

Var theCourse As Course
StudentBox.RemoveAllRows
If Me.CellTagAt(Me.SelectedRowIndex, 0) IsA Course Then
  theCourse = Course(Me.CellTagAt(Me.SelectedRowIndex, 0))
  For Each s As Student In theCourse.EnrolledStudents
     StudentBox.AddRow(s.LastName + ", " + s.FirstName)
  Next
End If
```

This code will display the Students in a Course. It creates a variable to hold the Course object, gets the Course object from the CellTag, and then loops through the Course's Students and lists them in StudentBox.

20) **Add this code to the EnrollButton's Pressed event:**

```
Var theStudent As Student
Var theCourse As Course
Var newRow As String
theStudent = New Student(FirstNameField.Text, LastNameField.Text)
If CourseBox.CellTagAt(CourseBox.SelectedRowIndex, 0) IsA Course Then
  theCourse = Course(CourseBox.CellTagAt(CourseBox.SelectedRowIndex, 0))
  theCourse.EnrollStudent(theStudent)
  newRow = theStudent.LastName + ", "
  newRow = newRow + theStudent.FirstName
  StudentBox.AddRow(newRow)
End If
```

This code enrolls the Student in the Course. It also adds the Student's name to StudentBox. Pay special attention to the fourth line, which takes advantage of the Constructor to create a new Student based on the first and last names.

21) **Run your project.**

22) **Select a Course and add some Students. Select different Courses and notice that the list of Students updates each time you select a Course.**

23) **Quit your application.**

24) **Give your Student class a new method called "FullName". It will take no parameters and it will return a string. Its code follows:**

```
Return Self.LastName + ", " + Self.FirstName
```

There were several times in the above code where it was necessary to get a Student's LastName, followed by a comma, followed by the Student's FirstName. Since that code was needed more than once, that's a good candidate for a function.

25) **Go back through your StudentInformation project and use that method to shorten your code in CourseBox's SelectionChanged event and EnrollButton's Pressed event.**

8.6 Modules

Right now, your project has an array of available Courses. This array is stored in Window1. There's nothing wrong with this approach, but there will be times when it limits what you can do. For example, if your application's interface required more windows, those new windows would not have access to the list of Courses.

Those are the times when it's appropriate to store information in a global variable. A global variable is a variable that is always available to all objects, windows, and methods.

Global variables are commonly stored in Modules. It can be tempting to think of a Module as similar to a class, but there are some important differences. While a Module can contain properties, methods, and functions just like a class can, a Module never needs to be instantiated with the New operator. In other words, a Module always exists in your application.

1) **In your StudentInformation project, add a Module to the project by choosing Module from the Insert menu while your App class is selected in the Contents pane. Name this Module "Globals".**

2) **Give your Module a property: MyCourses() As Course.**

3) **In Window1, delete the MyCourses() property.**

4) **Run your project.**

   It should function in an identical way. The key difference, which isn't noticeable to the user, is that the list of Courses is now available to your entire application and not just Window1.

5) **Quit your application.**

6) **Add a new window to your project by choosing Window from the Insert menu. Leave its default name of Window2.**

7) **Add a List Box to Window2 named "OtherCourseBox". Add this code to its Opening event:**

```
OtherCourseBox.RemoveAllRows
For Each c As Course In MyCourses
  OtherCourseBox.AddRow(c.Title)
  OtherCourseBox.CellTagAt(OtherCourseBox.LastAddedRowIndex, 0) = c
Next
```

8) **Add a Button to Window1, set its Caption to "Show Other Courses Window" and add this code to its Pressed event:**

```
Window2.Show
```

9) **Run your project**

10) **Click the Button to display Window2.**

You should see the same list of Courses in both windows. This is because they are both pulling from the same array in your Globals module.

11) **Quit your application.**

Chapter 9

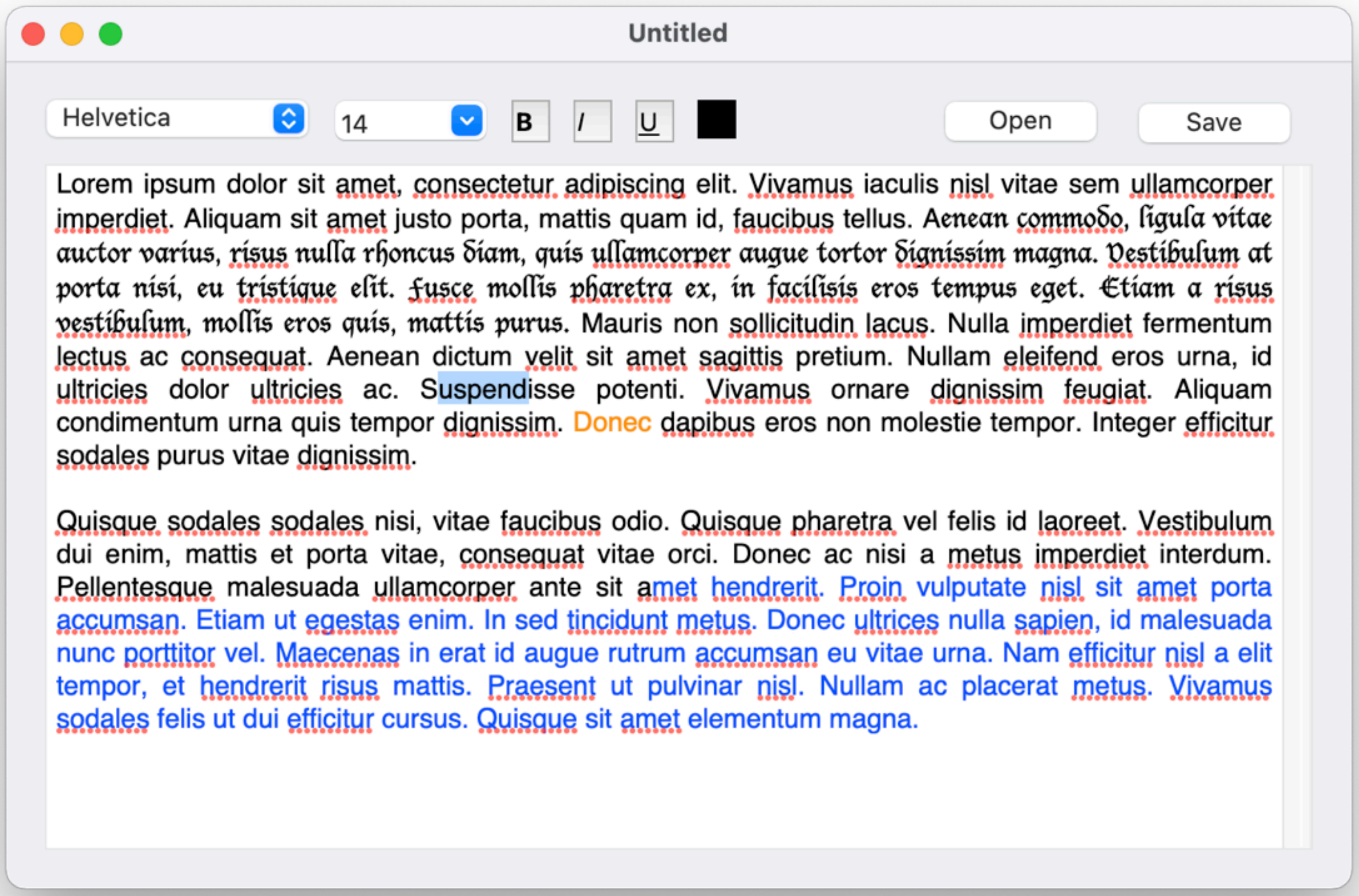
In & Out

CONTENTS

9.1 Chapter Overview

The apps you have created so far in this book have a common, fatal flaw. It's not your fault, though; it's just something you haven't learned yet. All of the projects you have built "reset themselves" every time you run them. They don't store any data and they don't remember anything about the last time they ran. Even the little styled text editor you built a few chapters ago couldn't save or open files.

In this chapter, you will build a styled text editor that opens, edits, and saves files. Your app will also be able to change the font, size, style, and color of the text. Your app may look something like this:



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus iaculis nisl vitae sem ullamcorper imperdiet. Aliquam sit amet justo porta, mattis quam id, faucibus tellus. Aenean commodo, ligula vitae auctor varius, risus nulla rhoncus diam, quis ullamcorper augue tortor dignissim magna. Vestibulum at porta nisi, eu tristique elit. Fusce mollis pharetra ex, in facilisis eros tempus eget. Etiam a risus vestibulum, mollis eros quis, mattis purus. Mauris non sollicitudin lacus. Nulla imperdiet fermentum lectus ac consequat. Aenean dictum velit sit amet sagittis pretium. Nullam eleifend eros urna, id ultricies dolor ultricies ac. Suspendisse potenti. Vivamus ornare dignissim feugiat. Aliquam condimentum urna quis tempor dignissim. Donec dapibus eros non molestie tempor. Integer efficitur sodales purus vitae dignissim.

Quisque sodales sodales nisi, vitae faucibus odio. Quisque pharetra vel felis id laoreet. Vestibulum dui enim, mattis et porta vitae, consequat vitae orci. Donec ac nisi a metus imperdiet interdum. Pellentesque malesuada ullamcorper ante sit amet hendrerit. Proin vulputate nisl sit amet porta accumsan. Etiam ut egestas enim. In sed tincidunt metus. Donec ultrices nulla sapien, id malesuada nunc porttitor vel. Maecenas in erat id augue rutrum accumsan eu vitae urna. Nam efficitur nisl a elit tempor, et hendrerit risus mattis. Praesent ut pulvinar nisl. Nullam ac placerat metus. Vivamus sodales felis ut dui efficitur cursus. Quisque sit amet elementum magna.

If you're wondering what this "lorem ipsum" nonsense is, it's sometimes called fake Latin. It doesn't mean much of anything, but it provides a distribution of letters that's nearly identical to written English, so it's useful for gauging how a design will look without including any "real" text, which could distract the viewer from the design itself. It is often used in prototyping interfaces.

9.2 Types Of Files

Before you start writing code, you need a good understanding of what a file is. In a nutshell, a file is a unique piece of data on your computer. It can contain data for a picture, a story, a song, a movie, or even a collection of settings. When you download a song from iTunes or Amazon, that's a file. When you write a report in Word and save it for later, that's a file.

Almost every file has a type. A file's type indicates what kind of data the file contains. For example, a text file contains plain text data. A JPEG file contains picture data. An MP3 file contains audio data. File types are fairly specific: instead of a "picture" file type, there are JPEGs, PNGs, GIFs, BMPs, and others. Each of these may contain a picture, even the same picture, but your computer needs to open and read each one in a different way. So a file type not only tells the computer what kind of data it will find in the file, but also how to read it.

You can help your application to look for certain types of files by using a File Type Group. To add a File Type Group to your project, choose File Type Group from the Insert Menu. Give your File Type Group a meaningful name, such as PictureTypes (or anything else that describes the files you're specifying). Click the left-most button in the File Type Group Editor command bar to drop down a list of common file types. You can choose something such as "image/png" to create a File Type for using PNG picture files. When you select it, it is added to the File Type Group with many of its properties filled in for you.

You will see how to use a File Type Group to filter for certain file types in section 9.4.

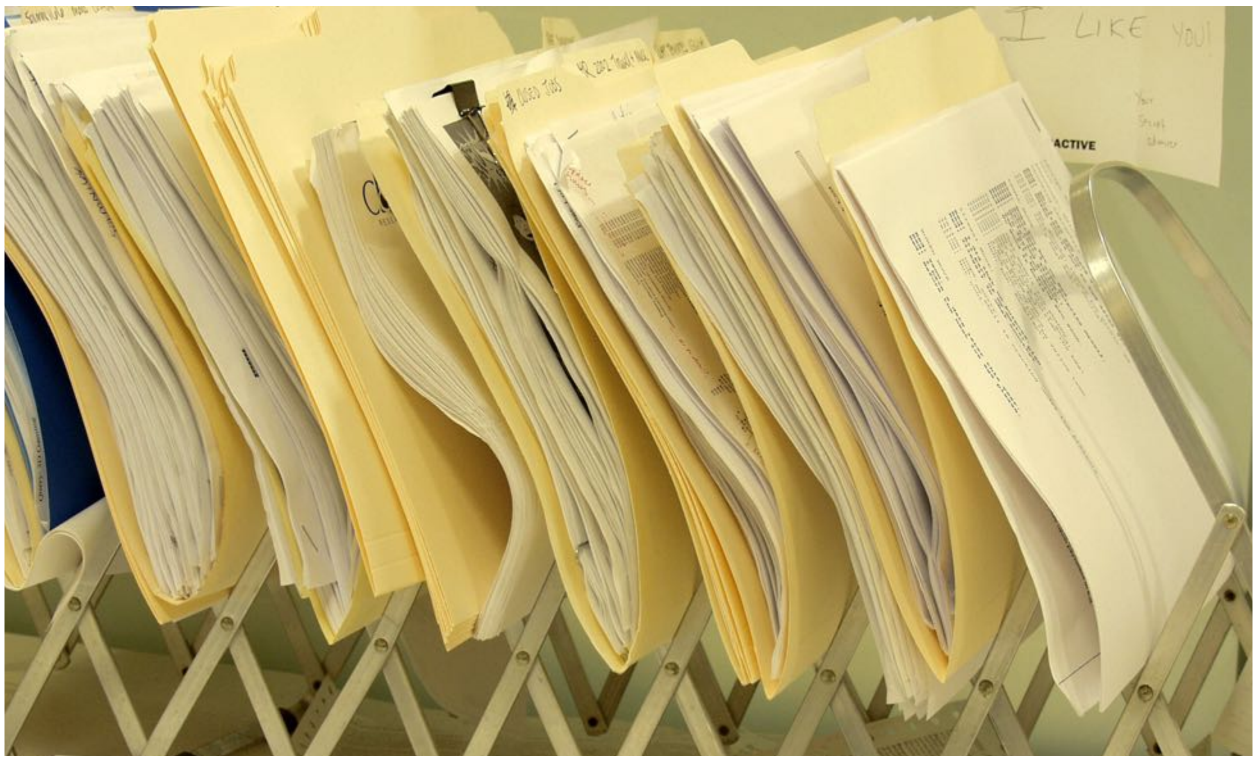
9.3 Working With Files

In Xojo, a file is represented by a class called FolderItem. That name is understandably confusing, and many people initially assume that a FolderItem can only represent a folder. But a FolderItem can indeed represent any file. Think of a FolderItem as any item that can be contained in a folder, whether it's a file or another folder.

A FolderItem, as you might imagine, has many properties and methods. Some commonly used properties include its Name (a string), its CreationDateTime (a datetime indicating the date and time the file came into existence), its ModificationDateTime (a datetime indicating when the file was last changed), IsWriteable (a boolean indicating whether you will be able to save changes to the file), and its Length (an integer indicating the size of the file on disk, in bytes). Most of these are fairly self explanatory, so you are encouraged to experiment with these properties later.

The FolderItem also has some properties whose use and meaning may not be as apparent. One such property is IsFolder. IsFolder is a boolean that tells you whether or not the FolderItem is a folder (also called a directory). This can be very useful to know: you don't want to try to open a folder thinking it's a picture!

Another property that may seem curious is the Exists property. This is a property that, appropriately enough, tells you whether or not the FolderItem exists on your drive. This may seem very odd. After all, how would you have a FolderItem that points to a non-existent file? The short answer is that it's actually quite common. In fact, it's the only way to create a new file or folder that doesn't yet exist. This will become clear later in the chapter.



If the FolderItem that you are working with is a folder and not a file, the Count property will tell you how many items it contains. This number will include other folders, but not the items in those subfolders.

Related to the Count, the FolderItem has a function called Child. Child takes the name of a file or folder and returns a FolderItem representing that file or folder. Alternatively, ChildAt is a function that takes an integer indicating the number of the contained file or folder and returns it to you as another FolderItem. If it’s less confusing, you can think of Item as an array of FolderItems rather than a method.

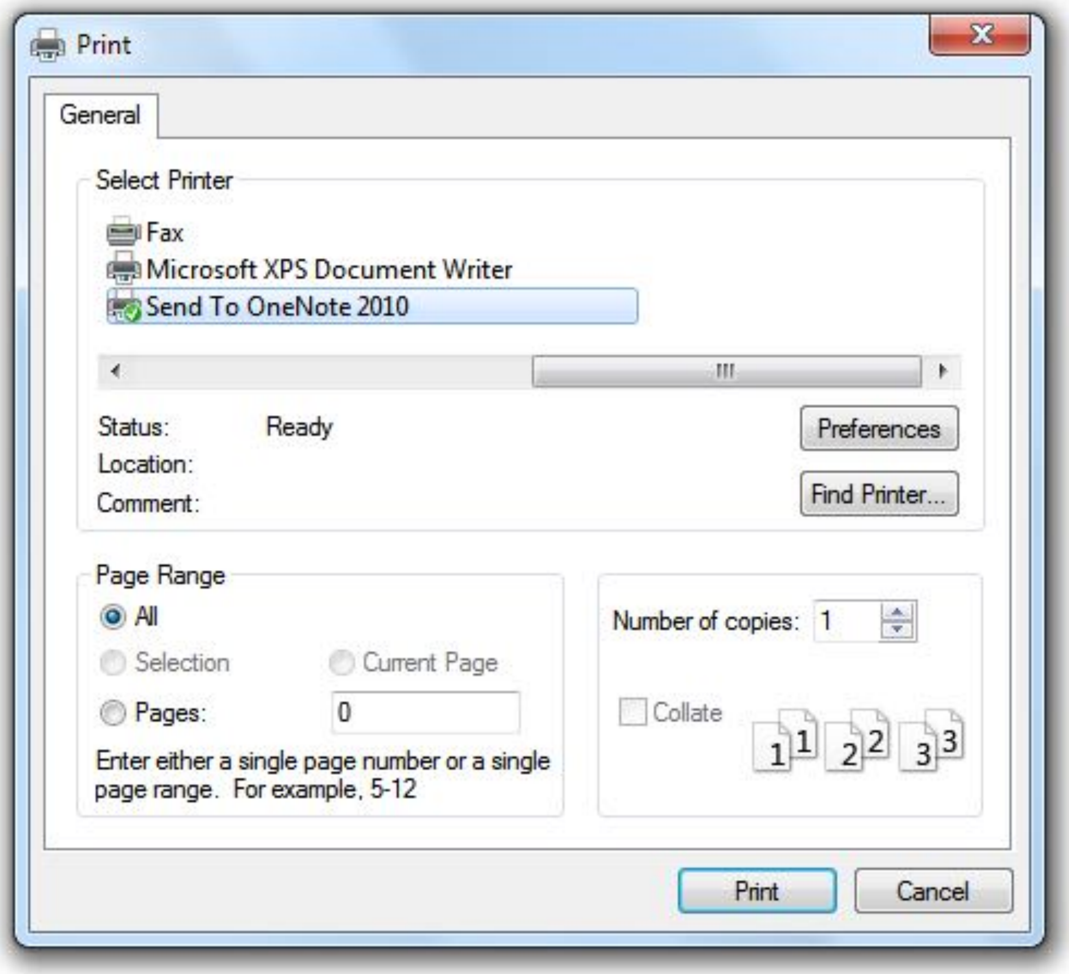
Also note the Parent property. The Parent is the FolderItem containing your FolderItem. So if you had a file called MyXojoProject and it was stored in a folder called My Projects, the Parent property of MyXojoProject would give you a FolderItem representing the My Projects folder.

9.4 Handling Open/Save Dialogs

All this talk about FolderItems is great, but you're probably wondering how to tell Xojo which files you want to work with. This is best accomplished through the FolderItemDialog class.

But before you dig into dialogs, it might be good to explain what a dialog is. In the simplest terms, a dialog is a minimal window that either retrieves information from the user or provides information to the user.

You have almost certainly seen a dialog that retrieves information from the user. A classic example is the print dialog, which asks you to specify a printer and possibly choose some additional settings:



Status: Ready
Location:
Comment:

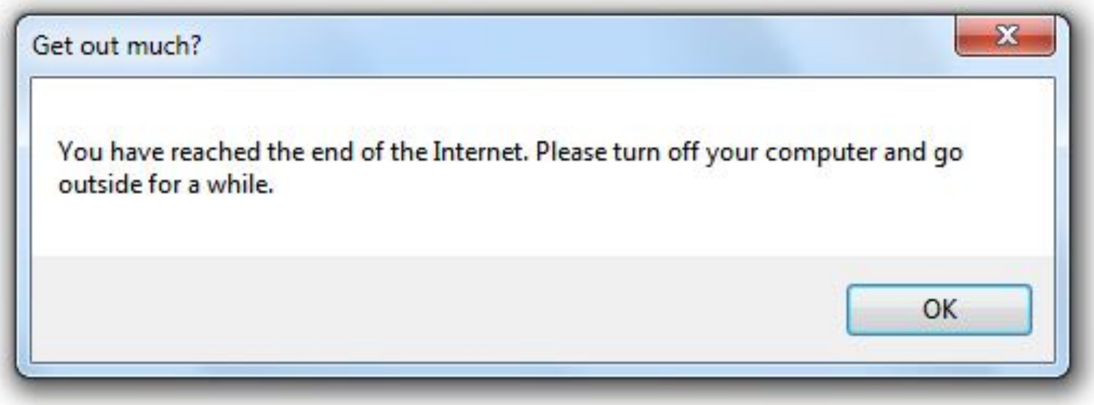
[✓ All]
Selection
Current Page
Pages: 0

Enter either a single page number or a single page range. For example, 5-12

Number of copies: 1

Collate

A dialog that provides information to the user may look something more like this:



Get out much?

You have reached the end of the Internet. Please turn off your computer and go outside for a while.

There are three special versions of the FolderItemDialog class that you will use: OpenFileDialog, SaveFileDialog, and SelectFolderDialog. When you want to prompt the user to open a file, use OpenFileDialog. When you want the user to save a file, use SaveFileDialog. Finally, when you want the user to choose a folder, use SelectFolderDialog.

(There are also ways for you to open and save specific files without the user's help. This will be covered later in this chapter.)

Let's begin with the SelectFolderDialog.

1) **Create a new Xojo project and save it as "Dialogs".**

2) **Add a Button and a List Box to Window1.**

3) **Name the List Box "FileBox" and set "Select Folder" as the Button's Caption.**

4) **Add this code to the Button's Pressed event:**

```
Var myFolder As FolderItem

Var d As SelectFolderDialog
d = New SelectFolderDialog
myFolder = d.ShowModal
If myFolder <> Nil Then
  FileBox.AddRow("Name: " + myFolder.Name)
  FileBox.AddRow("Size: " + myFolder.Length.ToString + " bytes")
  FileBox.AddRow("Items: " + myFolder.Count.ToString)
  FileBox.AddRow("Parent: " + myFolder.Parent.Name)
End If
```

The first thing this code does is create two variables: one for the SelectFolderDialog and one for the FolderItem. Since SelectFolderDialog is a class, it needs to be instantiated with the New operator.

ShowModal is a function in all of the folder item dialog classes. It displays a modal window to the user (a modal window is one that blocks the rest of your application until it is dismissed by the user, whether by selecting an item or canceling the operation; this is in contrast to a standard dialog, which still allows access to some or all of the application). The function will return Nil if the user pressed the Cancel button, which is how you check to see if myFolder is Nil before continuing, or it will return a FolderItem representing the folder that the user selected.

Once the code verifies that the FolderItem is valid, it adds a few rows to FileBox to show you some properties of the selected folder. Note the syntax used to show the FolderItem's Parent's Name: myFolder.Parent.Name. Because the Parent is itself a FolderItem, you can access its properties just as you can do with the FolderItem you initially selected.

5) **Run your project.**

6) **Click the Button, choose a folder, and examine the properties that appear in FileBox:**

The file name and file size should be straightforward.

The Parent was covered above.

Notice the number of items the FolderItem contains. Now verify for yourself how many items it has. The number listed in FileBox is one higher than the number you see in the folder. This is because item number zero is the FolderItem itself. This can cause confusion, so it's something to remember and be aware of as you write code that deals with folders.

7) **Quit your application.**

To open a file, use the OpenFileDialog class. You will add that to your Dialogs project next.

8) **Add another Button to Window1. Set "Open File" as its Caption. Add this code to its Pressed event:**

```
Var myFile As FolderItem
Var d As New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
  FileBox.AddRow("Name: " + myFile.Name)
  FileBox.AddRow("Size: " + myFile.Length.ToString + " bytes")
  FileBox.AddRow("Parent: " + myFile.Parent.Name)
  Var theDate As String
  theDate = myFile.CreationDateTime.SQLDateTime
  FileBox.AddRow("Created on: " + theDate)
  theDate = myFile.ModificationDateTime.SQLDateTime
  FileBox.AddRow("Last modified on: " + theDate)
End If
```

9) **Run your project.**

10) **Click the Button, choose a file, and examine the properties in FileBox.**

This time, you have added its CreationDateTime and ModificationDateTime. Because these are both DateTime objects, you can treat them just like any other DateTime, such as by using SQLDateTime to easily view the date as a string. You also eliminated the FolderItem's Count property this time, since it is not a folder.

11) **Quit your application.**

When opening files, you may want to prohibit your user from choosing inappropriate file types. For example, if you are building an image editor, you probably don't want the user opening a Word document or an XML file. This is where File Type Groups come into play, using the Filter property of the FolderItemDialog.

1) **Create a new Xojo project and save it as "FileTypes".**

2) **Create a File Type Group called "ImageFiles" (choose File Type Group from the Insert Menu).**

3) **Add these FileTypes to the group: image/jpeg, image/png**

4) **Add a Button to Window1. Add this code to its Pressed event:**

```
Var f As FolderItem
Var d As New OpenFileDialog
d.Filter = ImageFiles.All
f = d.ShowModal
If f <> Nil Then
  MessageBox(f.Name)
End If
```

5) **Run your project.**

6) **Click the Button and browse the computer for files to open.**

Your application should allow you to choose only PNG and JPG files.

7) **Quit your application.**

You can also use a File Type Group to allow the user to choose a file format when saving a file.

8) **Add another Button with this code in the its Pressed event:**

```
Var f As FolderItem
Var d As New SaveFileDialog
d.Filter = ImageFiles.All
f = d.ShowModal
If f <> Nil Then
  MessageBox(f.Name)
End If
```

9) **Quit your application.**

9.5 Opening Files

Looking at data about files is one thing, but displaying and possibly editing the contents of those files is much more productive. Data can be read into your application in many different ways, depending on the type of file. An MP3 file will be read much differently from a plain text file, which in turn will be read much differently from a database file (which you'll learn about in Chapter Twelve). Nearly every application you can conceive of will have to read data, write data, or both.

A stream is a common way to read or write data. In fact, several other languages, including Java, use streams for this purpose.

One common way to read a file's data is the BinaryStream. You may be familiar with streaming music services like Spotify, Apple Music, or Pandora. These services send part of the song to your computer, wait for a while, then send some more of the song, instead of sending it all at once. The BinaryStream, and in fact, all file streams, work in a very similar way. Once a file has been chosen, BinaryStream is used to extract data from that file in small chunks, each of which is processed as it is read into the application. When all of the data has been read, it is then presented to the end user.

1) **Create a new Xojo Desktop project and save it as "Streams".**

2) **Add a Button and a Text Area to Window1.**

   Don't worry too much about their position and size, but make the Text Area as large as you can easily fit on the window.

3) **Add this code to the Button's Pressed event:**

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var b As BinaryStream
d = New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
  b = BinaryStream.Open(myFile)
  TextArea1.Text = b.Read(b.Length)
  b.Close
End If
```

   This code allows the user to select a file and then display its contents in the TextArea using a BinaryStream. The syntax for this is a bit different from what you may be used to.

   The first few lines of code should look familiar: creating a few variables, instantiating a new OpenFileDialog, and choosing a file. But once the file is chosen, you'll see this line:

```
b = BinaryStream.Open(myFile)
```

BinaryStream.Open is a shared method. Typically, a class's methods can only be called after you've created an instance of that class using the New operator, but a shared method allows you to call that method at any time. This will be explained in greater detail in Chapter Thirteen. For now, just note the syntax difference.

Once the BinaryStream has been instantiated, it reads the data from myFile and displays it in the Text Area. The BinaryStream's Read method brings a portion of the file's contents into your application. How large that portion needs to be is up to you. The Read method takes an integer telling it how many bytes to read. In this case, you've told it to read everything in one shot by specifying the Length of the BinaryStream (a BinaryStream's Length is like a FolderItem's Length).

4) **Run your project.**

5) **Click the Button and open any file.**

Note what appears in the Text Area. If you've chosen a plain text file, you can read its contents. If you've chosen anything else, such an MP3 or a picture, you will likely not be able to make heads or tails of the data being displayed.

6) **Quit your application.**

This project illustrates something very important about most of the files on your computer: they're not human-readable. A human-readable file is one that you can make sense of by looking at the raw contents. There are a few exceptions, such as XML files, HTML files, or other plain text files, but for the most part, the files on your computer can only be read by an app that is designed to read that kind of file.

A custom file format is somewhat like a map or key that tells the computer which data is stored at which place in the file.

Fortunately, Xojo includes some functions that make it easy for you to open some common file types.

For example, here is how to open a picture file.

1) **Create a new Xojo Desktop project and save it as "OpenPic".**

2) **Add a Button and a Canvas to Window1.**

The Canvas should cover most of the window. The Button will prompt the user to select a picture, which will then be displayed in the Canvas.

3) **Add this code to the Button's Pressed event:**

```
Var myPic As Picture
Var myFile As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
  myPic = Picture.Open(myFile)
```

```
    If myPic <> Nil Then
      Canvas1.Backdrop = myPic
    End If
  End If
```

The Picture class, as you saw with BinaryStream above, has a shared method called Open, which takes a FolderItem as its parameter. Again, because it is a shared method, you may use it at any time; you do not need to instantiate the object first.

The code then sets the Backdrop property of the Canvas. The Backdrop is a Picture in the background of the Canvas. The next chapter will cover graphics, pictures, and the Canvas in greater depth.

Before setting the Backdrop property, however, you must first check to make sure the Picture isn't Nil, just as with the FolderItem.

4) **Run your project.**

5) **Click the Button and select a picture file from your computer.**

If it is a valid picture, the Canvas should display it.

6) **Quit your application.**

Many times, the files you will need to open will only contain text. These are easy to open and read using the TextInputStream class. TextInputStream allows you to read all of the text in a file at one time (using the ReadAll function), read it line by line (using the ReadLine function), or read a certain number of characters at a time (using the Read function). All of these functions return a string. It is up to you to know what to do with that string, whether you choose to display it in a Text Field, store it in a Dictionary, or accomplish some other task.

1) **Create a new Xojo Desktop project and save it as "TextRead".**

2) **Add a Button and a Text Area to Window1.**

The TextArea should fill as much of the window as possible.

3) **Add this code to the Button's Pressed event:**

```
  Var myFile As FolderItem
  Var d As OpenFileDialog
  Var t As TextInputStream
  d = New OpenFileDialog
  myFile = d.ShowModal
  If myFile <> Nil Then
    t = TextInputStream.Open(myFile)
    TextArea1.Text = t.ReadAll
    t.Close
  End If
```

This example uses the ReadAll function from TextInputStream to read the selected file's entire contents.

4) **Run your project.**

5) **Click the Button and select a text file.**

The entire file should be displayed in the Text Area.

6) **Quit your application.**

7) **In the Button's Pressed event, change this line:**

```
TextArea1.Text = t.ReadAll
```

to this:

```
TextArea1.Text = t.ReadLine
```

8) **Run your project.**

9) **Click the PushButton and select a text file.**

This time, only the first line of the file should be displayed. You may be wondering at this point why you would ever want to do something like that. What use is just the first line of a file? It could be used for a number of things. One example would be a quick preview of a file's contents. Or you may want to display each line in a List Box.

10) **Quit your application.**

11) **Remove the Text Area from Window1 and add a List Box.**

12) **In the Button's Pressed event, change this line:**

```
TextArea1.Text = t.ReadLine
```

To this:

```
ListBox1.AddRow(t.ReadLine)
```

13) **Run your project.**

14) **Click the PushButton and select a text file.**

Do you see the problem? The app has only loaded the first line of the file into the ListBox because it only asked for the first line.

15) **Quit your application.**

To read more than one line, you need to use the ReadLine function more than once. You could simply repeat the line, like so:

```
ListBox1.AddRow(t.ReadLine)
ListBox1.AddRow(t.ReadLine)
ListBox1.AddRow(t.ReadLine)
```

Each time you run ReadLine, the app grabs the next line down. But that approach has two problems. First, it's not efficient code at all. Repetitive tasks like these should be done in a loop. That raises the second problem: how do you know how many times to do it?

In a nutshell, you keep reading data until you run out of data to read. That happens when you reach the end of the file, or EndOfFile. EndOfFile is a Boolean: when it's false, you still have data left to read; when it's true, no more data remains. So the solution to this problem is to use a While...Wend loop in conjunction with the TextInputStream's EndOfFile property.

16) Change the code in the Button's Pressed event to this:

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var t As TextInputStream
d = New OpenFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
  t = TextInputStream.Open(myFile)
  While Not t.EndOfFile
    Listbox1.AddRow(t.ReadLine)
  Wend
  t.Close
End If
```

Remember from earlier chapters that the Not operator tells your app that you want to use the opposite of the boolean value that you're referring to. In this case, as long as EndOfFile is not True, the loop will continue.

17) Run your project.

18) Click the Button and select a text file.

Each line of the text file should be listed as a separate row in the ListBox.

19) Quit your application.

9.6 Creating And Saving Files

Reading files is all well and good, but you will likely need to create some as well. In this section, you will learn how to use the TextOutputStream, which is, as its name implies, essentially the opposite of the TextInputStream.

Recall that the TextInputStream has functions to read all text, read one line of text, and read a certain number of characters. The TextOutputStream has similar methods for writing text, although there are only two of them: Write and WriteLine.

The WriteLine method takes one parameter, which is a string. It writes that string to the selected file and thens adds an EndOfLine character (which could be a line feed, carriage return, or a combination of them, depending on what kind of computer you're using). If you'd prefer to use a different delimiter, you can set the TextOutputStream's Delimiter property.

The Write function also takes one parameter, also a string. It writes that string to the selected file, but does not add any delimiters or EndOfLine characters.

Here's an example of the Write method in use.

1) **Create a new Xojo Desktop project and save it as "TextOut".**

2) **Add a Button and a Text Area to Window1.**

   Size the TextArea so that it takes up most of Window1.

3) **Add the following code to the Button's Pressed event:**

```
Var myFile As FolderItem
Var d As SaveFileDialog
Var t As TextOutputStream
d = New SaveFileDialog
myFile = d.ShowModal
If myFile <> Nil Then
  t = TextOutputStream.Create(myFile)
  t.Write(TextArea1.Text)
  t.Close
End If
```

The first few lines of this code should look familiar, although this is the first time you've seen the SaveFileDialog. The SaveFileDialog is similar to the OpenFileDialog, but it allows you to *create* files rather than *open* them. So even though MyFile is not Nil, the file it represents does not yet exist on your computer. The file is written to the disk when the TextOutputStream's Create function is called. Create is a shared method, as you saw in previous examples.

After the file is created and you have a TextOutputStream to work with, you can then add data to the file, using the Write method. In this example, you are adding all of the text in the Text Area to the file at once.

The Close method of the TextOutputStream tells your app that you are done writing data to the file, so it is safe to be used by other applications.

4) **Run your project.**

5) **Add some text to the Text Area and then click the Button.**

You will be prompted to save your file. Give it a name and save it.

6) **Quit your application.**

7) **Find your new file and open it in another text editor.**

Some example text editors include NotePad on Windows and TextEdit on macOS. When you open your file, you will see the same contents that you entered in your application. Congratulations! You can now exchange data with the rest of the world.

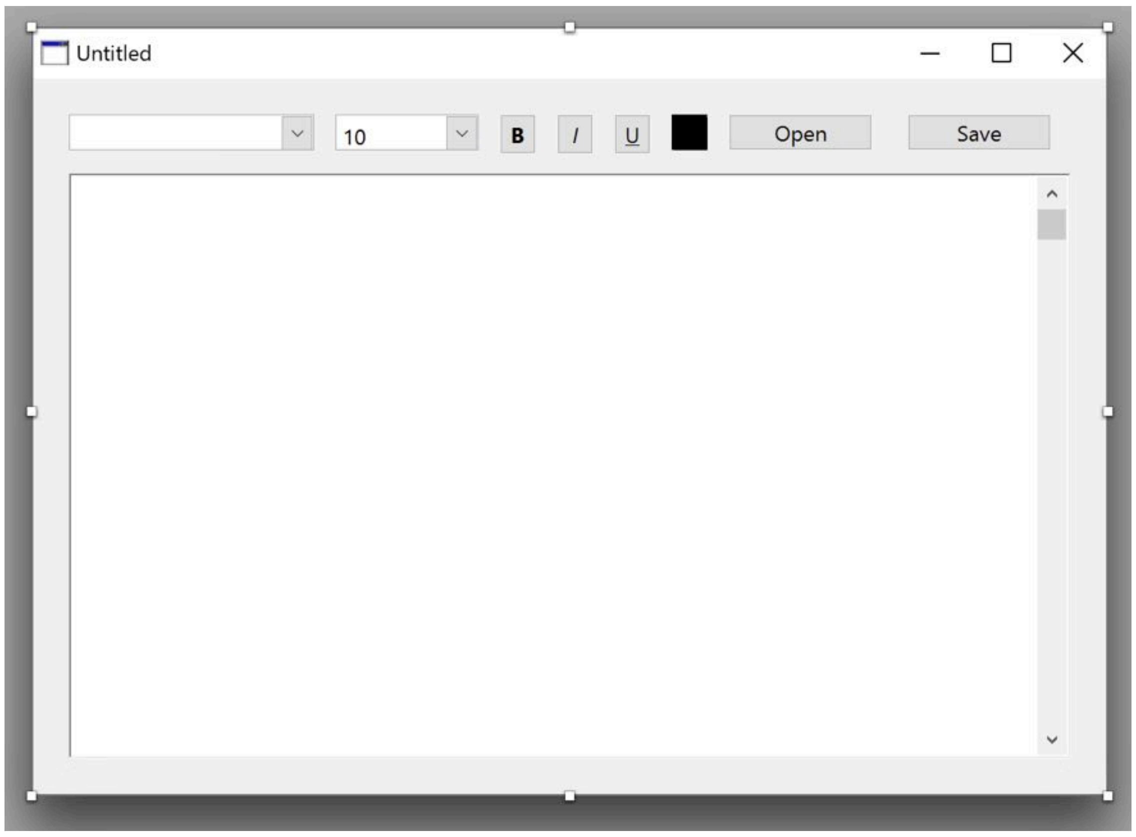
9.7 Hands On With Files

This chapter's sample project is a styled text editor. You will apply the information you learned about opening and saving files, as well as use several of the controls you learned about in earlier chapters.

Styled text is simply text that allows different fonts, sizes, and styles, such as bold, underline, and italic. This project will use a format called RTF, or Rich Text Format, to store the style information. RTF is a cross-platform, standard way to store styled text, and the RTF data itself can be stored as plain text, so you can use TextInputStream and TextOutputStream to open and save not just your files, but any other RTF file you might have on your computer.

1) Create a new Desktop project and save it as "StyledTextEditor".

Here is a sample of what the interface might look like, but feel free to be creative with yours:



2) Here are the controls you'll need on Window1 and the names that the sample code will be using:

Name	Caption	Control
FontMenu	N/A	Popup Menu
FontSizeField	N/A	Combo Box
BoldButton	B	Bevel Button
ItalicButton	I	Bevel Button
UnderlineButton	U	Bevel Button
ColorButton	N/A	Rectangle
OpenButton	Open	Button
SaveButton	Save	Button
EditingField	N/A	Text Area

3) With Window1 selected, go to the Insert Button and choose Property to give Window1 a new property: TheFile As FolderItem.

Since the text editor will need to remember which file it's dealing with when it's time to save the file, it should be a property of the window and not something that is created and thrown away in each button's Pressed events. This is an example of scope. As mentioned earlier, scope indicates how long your variable will last and what else has access to it. If you were to create MyFile in the OpenButton's Pressed event, MyFile would "go out of scope" when the event is complete; later, when the time comes to save the file, the SaveButton wouldn't know which file to use without asking the user again, which would be annoying for the user, not to mention error prone (users make mistakes, and your code should protect them from doing so as much as possible). By making MyFile a property of the window, you can guarantee that it will not go out of scope.

4) Set the ButtonStyle of each Bevel Button to ToggleButton.

To add some visual flair to your text editor, use the Inspector to set each Bevel Button's style to match its purpose. Use the settings Icon (cogwheel) of the Inspector tab to get to Font property then check the Bold property on BoldButton, check the Italic property on ItalicButton, and check the Underline property on UnderlineButton. This gives your users a clear visual indication of each button's purpose. You have probably encountered a similar interface before in a text editor or word processor.

5) In BoldButton's Pressed event, enter this code:

```
EditingField.SelectionBold = Me.Value
```

SelectionBold is a boolean indicating whether a TextArea's selected text is bold or not. By setting SelectionBold to the BevelButton's Value property (which matches its toggle state), you can make sure that the selected text turns bold when the button is pressed.

6) Add this code to ItalicButton's Pressed event:

```
EditingField.SelectionItalic = Me.Value
```

7) Add this code to UnderlineButton's Pressed event:

```
EditingField.SelectionUnderline = Me.Value
```

8) In the Rectangle's MouseDown event, add this code:

```
Return True
```

For the ColorButton, a Rectangle is being used. This control wasn't covered in the chapters on controls and events because it doesn't really do much. For example, it has no Pressed event. You can, however, use it as a button in a pinch. Set its FillColor property to black in the Inspector.

The "Return True" line above is, admittedly, somewhat strange at first glance. To make sure that the Rectangle responds to a mouse click, you need to implement both its MouseDown and MouseUp events. MouseUp is where the click should happen, because that's how buttons work: the action happens when the button is released. However, a Rectangle's MouseUp event doesn't get called unless you have "Return True" in its MouseDown event.

9) In the Rectangle's MouseUp event, add this code:

```
Var c As Color
If color.SelectedFromDialog(c, "Choose a Text Color") Then
  EditingField.SelectionTextColor = c
  Me.FillColor = c
End If
```

This code will use the Color.SelectFromDialog function to prompt the user to choose a color. Color.SelectFromDialog returns True if the user picks a color and False if not. If the user chooses a color, this code sets the selected text's color to the color that the user selected. It also sets the FillColor property of the Rectangle itself so that it matches the text. If the user cancels, and SelectColor returns False, you can safely ignore it.

10) Add this code to FontMenu's Opening event:

```
Var myFontCount As Integer
myFontCount = System.FontCount - 1
For i As Integer = 0 To myFontCount
  Me.AddRow(System.FontAt(i))
Next
```

The FontMenu's code should look familiar, as you worked with similar code in earlier chapters. This code simply loops through each installed font and adds its name to the Popup Menu.

11) Add this code to the FontMenu's SelectionChanged event:

```
If Me.SelectedRowIndex <> -1 Then
  EditingField.SelectionFontName = item.Text
End If
```

This code sets the selected text's font to the font chosen by the user, after making sure the user chose a non-empty row.

12) Add this code to FontSizeField's TextChanged event:

```
EditingField.SelectionFontSize = Me.Text.ToInteger
```

This is a simple line of code that grabs the value in the Combo Box, converts to a numeric value, and sets the TextArea's selected text to that size.

FontSizeField is a Combo Box. This is so that you can provide some preset options for your user while still allowing him or her to enter a custom font size. This, again, is a fairly common way for an application to handle a font size menu. To add values to FontSizeField, click the Pencil icon that appears when you hover over the control with your mouse (it appears in the lower right corner). In the dialog that appears, add some values to that you want to see in your menu. You can use whatever values you like here, but some common font sizes are 10, 11, 12, 14, 18, 24, 36, and 48. And for additional values, your user can always enter a custom size.

The Combo Box does not have a Pressed event; when the user makes a selection or changes the value, the TextChanged event is called.

13) Run your project.

You can't yet open or save documents, but you can test the editor itself. Enter some text into the TextArea and play with the styles, fonts, sizes, and colors. Now highlight some text whose style you have already changed. Do you see the problem? The TextArea is responding to the style changes, but the style buttons, font menu, and other controls are not reflecting the style of the selected text, which they should.

14) Quit your application.

15) Add this code to the Text Area's SelectionChanged event:

```
Var fontListCount As Integer
ColorButton.FillColor = Me.SelectionTextColor
BoldButton.Value = Me.SelectionBold
ItalicButton.Value = Me.SelectionItalic
UnderlineButton.Value = Me.SelectionUnderline
Var fontSize As Integer = Me.SelectionFontSize
FontSizeField.Text = fontSize.ToString
fontListCount = FontMenu.RowCount - 1
For i As Integer = 0 To fontListCount
  If Me.SelectionFontName = FontMenu.RowValueAt(i) Then
    FontMenu.SelectedRowIndex = i
      Exit
  End If
Next
```

The SelectionChanged event is called whenever the user changes his or her selection, whether by clicking in a word, using the arrow keys to navigate through the text, or highlighting some text. The font menu, font size field, style buttons, and color button should all change to match whatever text is highlighted.

Most of this code simply works backward from what the various buttons and menus do, making sure the buttons and menus match what is selected. Note that changing the text displayed in the FontMenu is more involved than simply setting its RowValueAt property. The RowValueAt property can't be set directly, so you need to set up a loop to walk through each item in the menu and check for a match. When you find one, you set the PopupMenu's SelectedRowValue and then Exit the loop.

16) **Run your project again.**

17) **Enter some text into the Text Area and play with the styles, fonts, sizes, and colors.**

This time, the menus and buttons should change when you select different text. But you still can't open or save documents.

Note that on macOS, not all fonts have support for bold and italic variations. A different font (such as Arial) may be necessary to use all the styles.

18) **Quit your application.**

19) **Add this code to OpenButton's Pressed event:**

```
Var myFile As FolderItem
Var d As OpenFileDialog
Var textData As TextInputStream
d = New OpenFileDialog
d.Title = "Select A File"
myFile = d.ShowModal
If myFile <> Nil Then
  textData = TextInputStream.Open(myFile)
  EditingField.StyledText.RTFData = textData.ReadAll
  textData.Close
  TheFile = myFile
  Self.Title = TheFile.Name
End If
```

This code will prompt the user to select a file and use a TextInputStream to get its contents. Because the data will be RTF, you'll need to read the data into the RTFData property of the Text Area's StyledText property rather than into the Text property directly. This sounds more confusing than it is in practice.

Aside from the addition of dealing with RTFData, this code should look very similar to what you've already learned in this chapter. One nice interface consideration is that this code also sets the Title of the window to the Name of the file being edited. It also sets the window's TheFile property to the selected file.

20) **Add this code to SaveButton's Pressed event:**

```
Var myFile As FolderItem
Var d As SaveFileDialog
Var textData As TextOutputStream
If TheFile = Nil Then
  d = New SaveFileDialog
  d.Title = "Save Your File"
```

```
      myFile = d.ShowModal
    Else
      myFile = TheFile
    End If
    If myFile <> Nil Then
      textData = TextOutputStream.Create(myFile)
      textData.Write(EditingField.StyledText.RTFData)
      Self.Title = MyFile.Name
    End If
```

Here in SaveButton's Pressed event, things aren't quite as simple. If the user has already opened a file, you can save data to the file using the window's TheFile property. However, if the user has started from scratch, you'll need to ask the user where to save the file and what to call it. Once that part is done, it's a matter of writing the RTFData to the file.

Note that when the file is saved, the window's Title is updated, in case the user created a new file.

21) **Run your project.**

22) **Experiment with opening and saving files and with editing the text and style data. If you have access to other RTF files, open them as well.**

23) **Quit your application.**

In this chapter, you have learned the fundamentals of reading and writing data. You should have a solid handle on reading and writing files, especially text files. There is still much to learn, especially when it comes to pictures and databases, both of which you'll learn about in the coming chapters.

Chapter 10

Picture This (Then Print It Out!)

CONTENTS

10.1 Chapter Overview

Now that you've learned how to store your user's data by creating files, the next step is allowing your users to print their information. Printing in Xojo is accomplished through the Graphics class.

In addition, as user interfaces become more sophisticated, it becomes more and more important to offer high quality graphics in your app. Sometimes this is in the form of pictures that you import into your project, and sometimes this is in the form of pictures that you generate with your code.

In Chapter 2, you learned about some of the data types that Xojo supports, including strings, numeric data types, dates, and colors. There are primarily two data types that you will use when dealing with image data. One of these data types is Picture. As you probably guessed from its name, the Picture class represents a picture. That picture can be loaded from a file that you already have or drawn by your code.

The other data type you will learn about is the Graphics class. It may seem odd to have both a Graphics class and a Picture class, but the reasons for this will become clear as you read this chapter. In reality, the Graphics and Picture classes work together to help you manage image data.

For this chapter's example project, you'll be making a change to the online food menu you created in an earlier chapter and giving it the capability to print the user's order.

A Note about HiDPI (Retina) Screens

If you have a HiDPI (Retina) display, you can turn on the HiDPI property to make text and images look sharper. To make sure it's on, go to the Shared Build Settings and check the Supports Retina / HiDPI property to make sure it's ON.

10.2 Working With Images

The Picture class, as noted above, holds an image. Rather than discuss it in the abstract, let's dig right in with an example.

1) **Create a new Desktop project in Xojo and save it as "Pictures".**

2) **Add a Button to Window1. Add this code to its Pressed event:**

```
Var f As FolderItem
Var d As OpenFileDialog
Var myPic As Picture
d = New OpenFileDialog
f = d.ShowModal
If f <> Nil Then
  myPic = Picture.Open(f)
End If
```

Some of this code should certainly look familiar. The code creates a few variables and then asks the end user to choose a file. One thing that may be unfamiliar, however, is that one of those variables has the data type of Picture.

The Picture variable, myPic, is assigned when Picture.Open is used. Picture.Open is a shared method of the Picture class, just like the shared methods you saw in the last chapter. It takes a FolderItem as its parameter and returns a Picture object, which is assigned to myPic (assuming the FolderItem is valid and not Nil).

3) **Run your project.**

4) **Click the Button and select a picture file (any common image format will do, such as JPEG, GIF, PNG, or TIFF).**

Notice what happens after you open the file: nothing. Nothing has happened because you haven't told your code to do anything with the picture object yet.

5) **Quit your application.**

6) **Add an Image Viewer to Window1.**

The Image Viewer is used to display a picture. Size the Image Viewer so that it covers most of Window1.

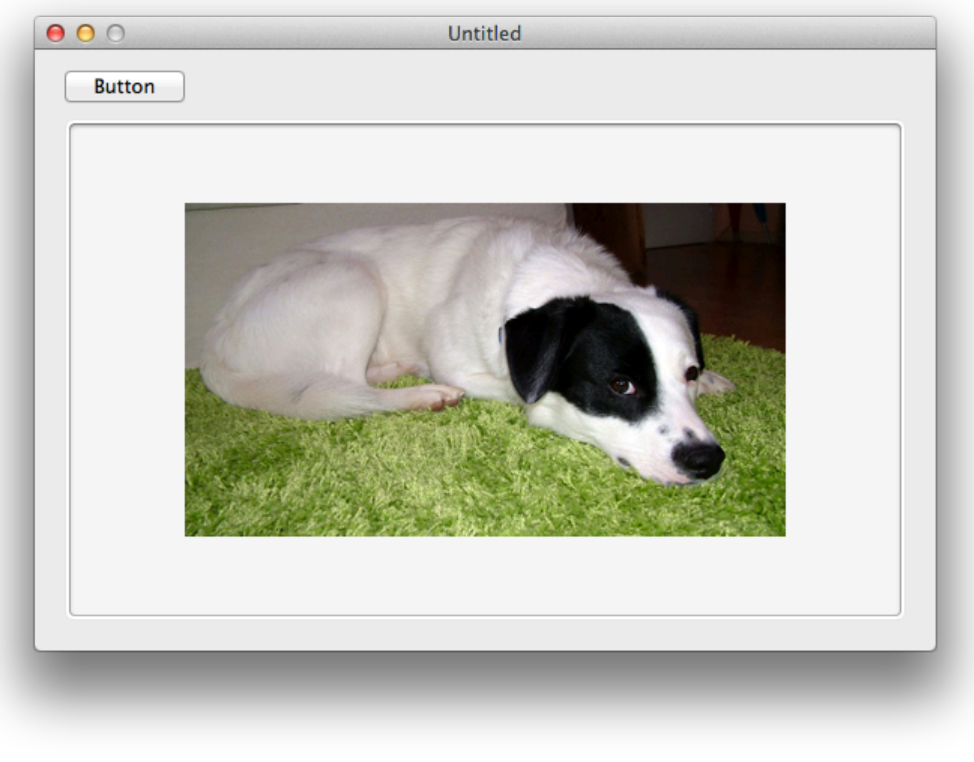
7) **Just above the "End If" line in the Button's Pressed event, add this code:**

```
ImageViewer1.Image = myPic
```

8) **Run your project.**

Again, click the Button and select a picture file.

9) **This time, you should see your selected picture displayed in the Image Viewer:**



Obviously, your picture will most likely be quite different.

10) **Quit your application.**

10.3 Drawing From Code

Now that you know the basics of opening and displaying a picture, it's time to learn about the Graphics class. While the Picture class is extremely useful for storing image data, the Graphics class does most of the "heavy lifting" when it comes to displaying and manipulating image data.

1) **Create a new Desktop project and call it Pictures-Canvas.**

2) **Add a Button and add a Canvas. Size the Canvas so that it covers most of the window.**

3) **Add a property to the Window: MyPic As Picture.**

4) **Add this code in the Button's Pressed event:**

```
Var f As FolderItem
Var d As OpenFileDialog
d = New OpenFileDialog
f = d.ShowModal()
If f <> Nil Then
  MyPic = Picture.Open(f)
  Canvas1.Refresh
End If
```

This is almost identical to the code you entered earlier; only a few lines have changed. Now the picture you selected is assigned to the property and the Canvas is told to update itself (by calling Refresh).

5) **In the Paint event handler for the Canvas, add this code:**

```
If MyPic <> Nil Then
  g.DrawPicture(MyPic, 10, 10)
End If
```

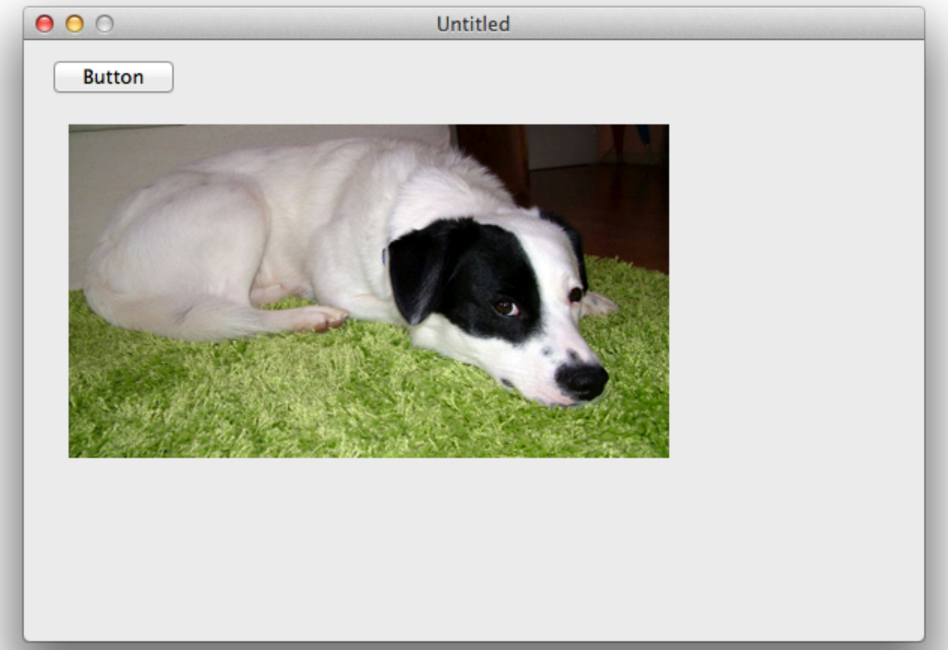
This code draws the picture in the Canvas (as long as a picture is available).

The Graphics class (provide by the g parameter to the Paint event) has many methods for manipulating image data. In this example, the DrawPicture method is being used to draw the selected Picture object into the Canvas (whatever is in the Graphics property of the Canvas will be displayed onscreen). You will learn much more about the DrawPicture method in this section, but for now, you should know that the three parameters you have given are the picture to draw followed by the X and Y coordinates indicating where to draw it.

6) **Run your project.**

7) **Click the Button and select a picture file.**

This time, you should see your selected picture displayed in the Canvas, such as in the screenshot below:



Note that the image displayed in the Canvas doesn't have the special insert border like it did in the Image Viewer. This is because a Canvas doesn't provide that frame automatically. Also note that while the ImageWell centered the picture for you, the Canvas does not. At this point, you may wonder why you would bother with a Canvas instead of ImageWell. In short, the Canvas gives you far greater control over the display of your image.

8) Quit your application.

The above example used the DrawPicture method with three parameters: the Picture to draw, followed by the X and Y coordinates at which to draw it. The coordinates are relative to the top left corner of the Canvas, so drawing the image at 0,0 would result in the image being tight against the upper left corner of the Canvas, while drawing the image at 72,36 would result in the image being approximately one inch from the left edge of the Canvas and one half inch from the top. Take a few minutes to experiment with different values for the X and Y coordinates. Notice how the position of the image changes.

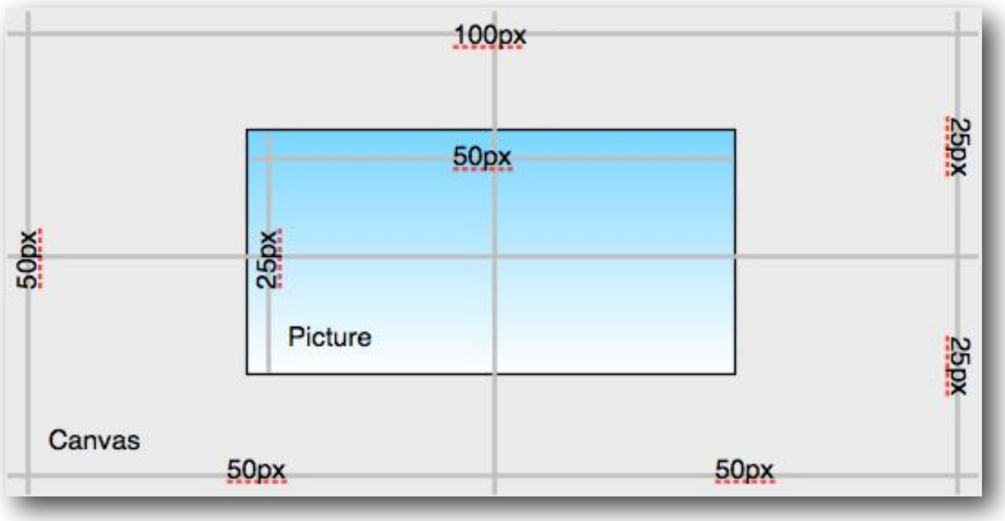
But don't limit yourself by thinking that you can only provide predetermined numbers as the coordinates. After all, they're just integers, so you can give DrawPicture any value that you can calculate. For example, suppose you wanted to center the image inside the Canvas. If you knew the size of the image and the size of the Canvas ahead of time, it would be relatively easy to do the math to center the image. The challenge comes in when you do not know either size ahead of time. Fortunately, there's a fairly simple way to center an image. The X coordinate should be half of the width of the Canvas minus half of the width of the Picture. The Y coordinate should be the same, only using the height instead of the width. Conveniently, the Canvas and Picture classes helpfully provide you with this information with their Height and Width properties.

9) Return to your Pictures-Canvas project in Xojo.

10) Change the Paint event to calculate the center position for the image. The code now looks like this:

```
If MyPic <> Nil Then
  Var x, y As Integer
  x = g.Width / 2 – MyPic.Width / 2
  y = g.Height / 2 – MyPic.Height / 2
  g.DrawPicture(MyPic, x, y)
End If
```

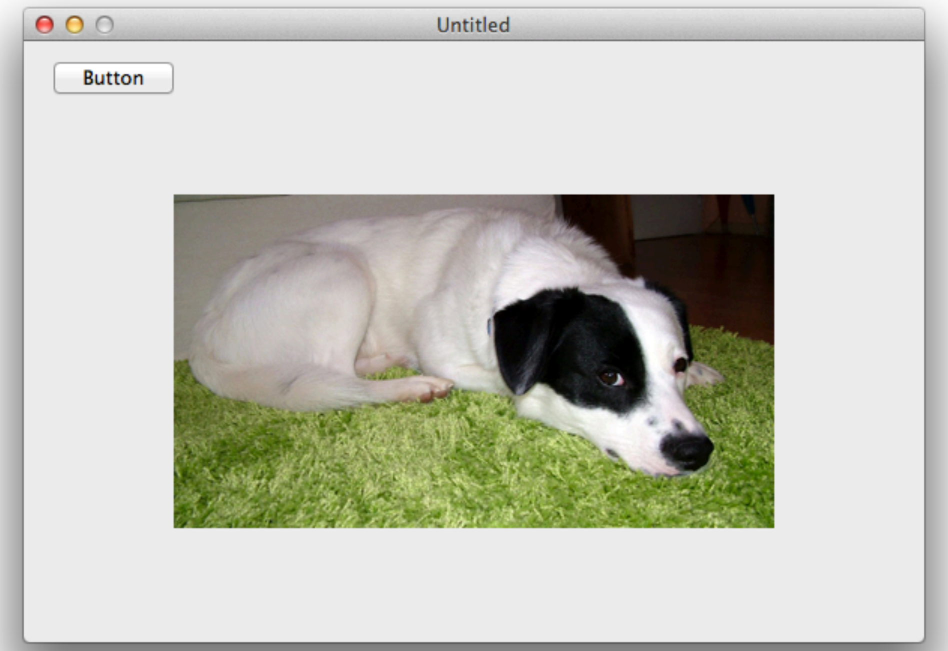
The above code calculates X and Y. X is half of the width of the Canvas minus half of the width of the Picture. And Y is half of the height of the Canvas minus half of the height of the Picture. If that formula is confusing, this diagram may help:



11) Run your project.

12) Click the Button and select a picture file.

This time, you should see your selected picture displayed in the Canvas, but centered within it:



Remember that the image is centered within the Canvas, not within the window.

13) Quit your application.

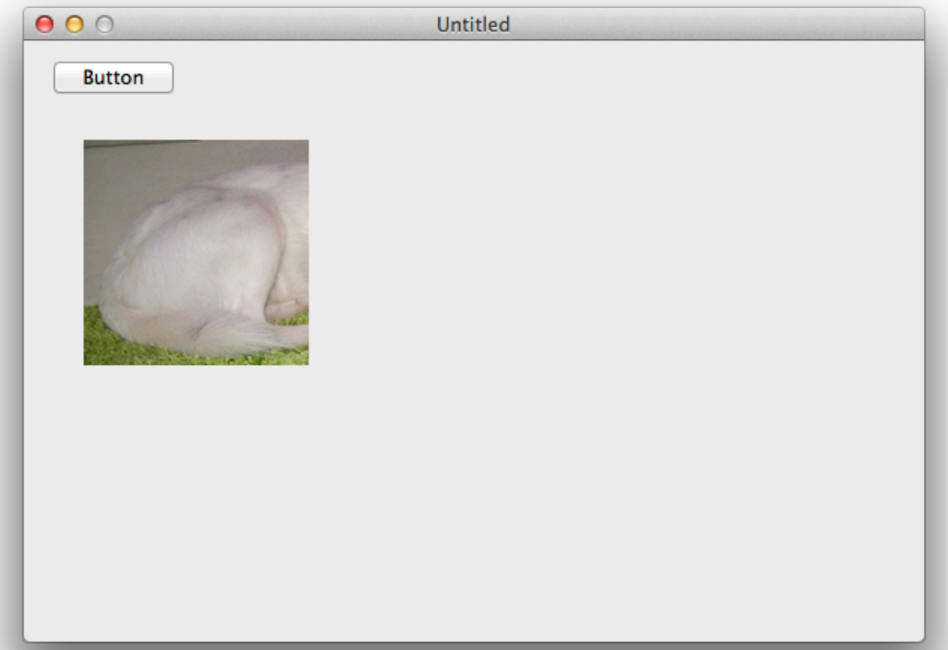
Now that you've learned more about how to position your images using the DrawPicture method, you may be wondering if it's possible to crop or scale your images. And it is possible.

DrawPicture requires the three parameters you've already seen, but it can take more than that if you need more control over how the image is displayed. In fact, DrawPicture can take up to nine parameters.

In your Pictures-Canvas project, go to the Paint event and change the line of code containing DrawPicture to this:

```
    g.DrawPicture(MyPic, 20, 20, 150, 150)
```

Those two additional parameters are called DestWidth and DestHeight, and they determine the width and height of the image when it's displayed. When you run your project, this time, you'll see that the image you select does not go beyond 150 points wide or 150 points tall (if the image is smaller than 150 points in either dimension, you won't see the difference). In short, DestWidth and DestHeight allow you to crop the image:

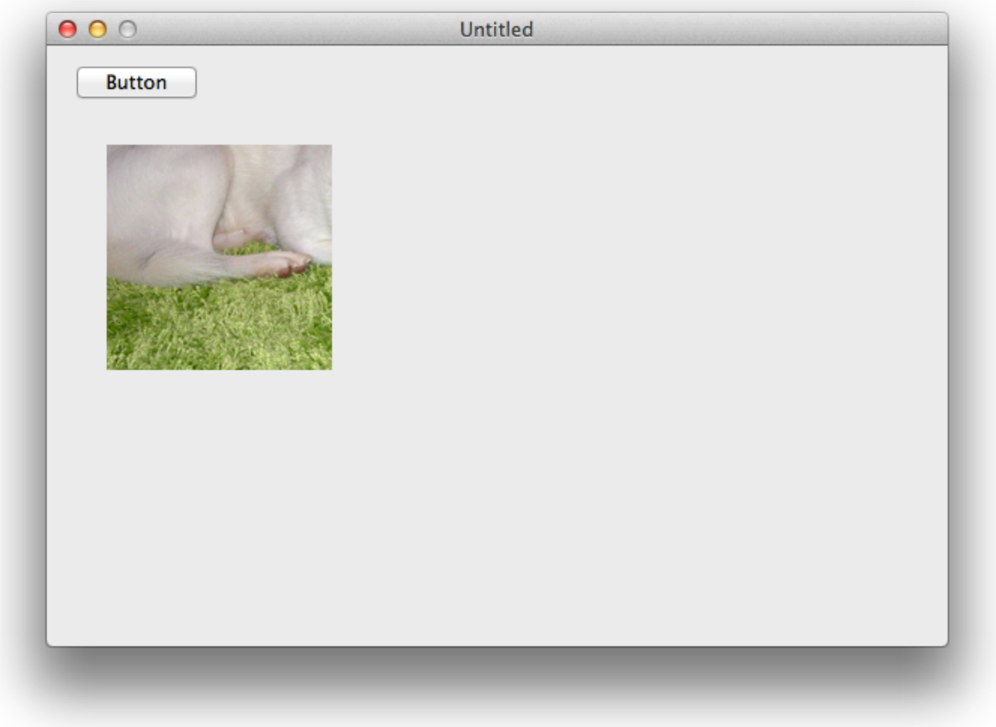


Now add two more parameters, so that your line of code looks like this:

```
    g.DrawPicture(MyPic, 20, 20, 150, 150, 50, 50)
```

These next two parameters are called SourceX and SourceY, and they tell DrawPicture where in the original image to start drawing *from*. By default, these are both zero, so that DrawPicture starts at the upper left hand corner of the image.

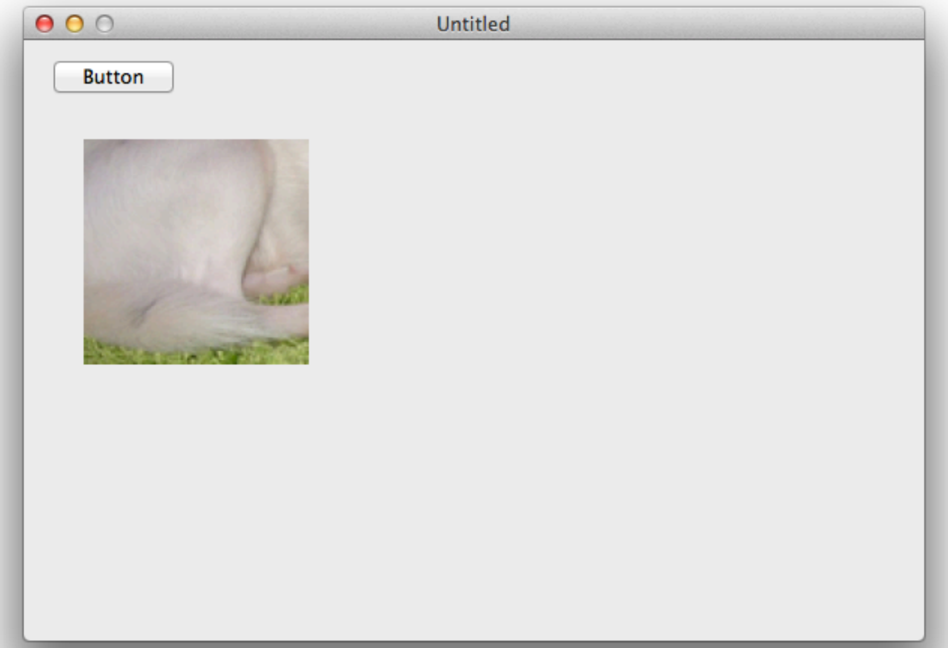
With SourceX and SourceY in place, you might see an image more like this:



So far, that's seven parameters. As mentioned above, DrawPicture can take up to nine. The final two are called SourceWidth and SourceHeight. These allow you to scale the image. Change the DrawPicture line one more time:

```
    g.DrawPicture(myPic, 20, 20, 150, 150, 50, 50, 100, 100)
```

This time, the image may look something like this:



Note that the image is zoomed in, because it's been scaled up.

These additional parameters to DrawPicture will almost always be calculated values rather than hard-coded numbers as has been demonstrated here. But used in combination, they allow you great control over how your image is displayed.

This is all great if you have some picture handy on your computer already. But you may have to draw your own at some point. The Graphics class has several methods that allow you to create images.

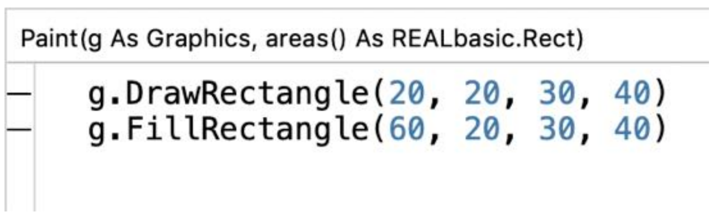
1) **Create new Xojo Desktop project and save it as "Drawing".**

2) **Add a Canvas to Window1. Size it so that it covers almost all of the window.**

3) **Add this code to the Paint event of the Canvas:**

```
g.DrawRectangle(20, 20, 30, 40)
g.FillRectangle(60, 20, 30, 40)
```

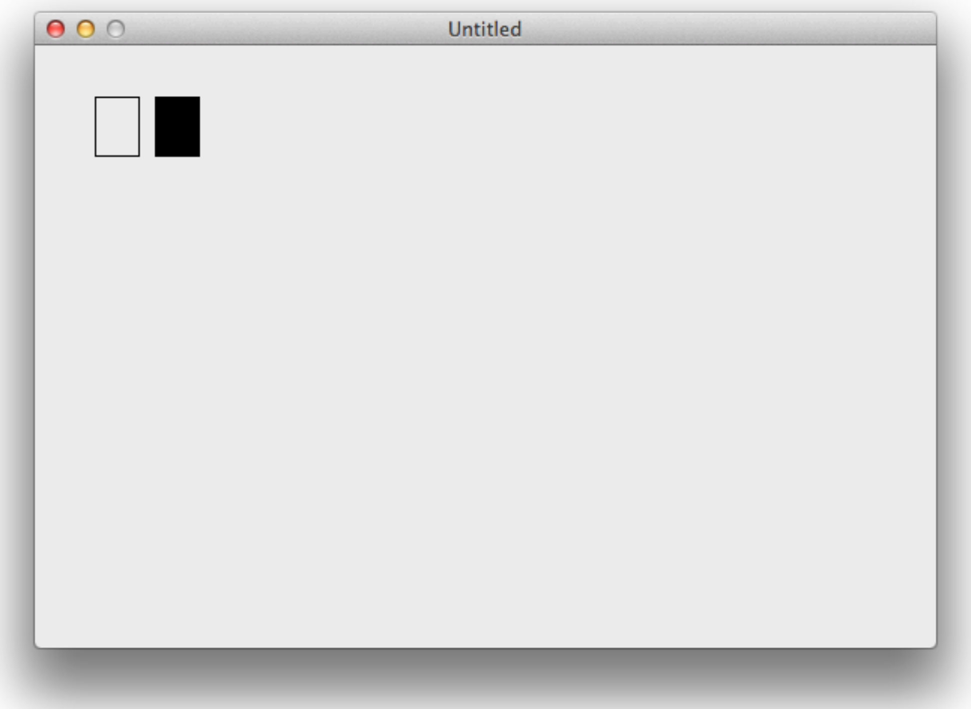
The "g" variable here is the Graphics property of Canvas1. The Paint event provides that property for you:



```
Paint(g As Graphics, areas() As REALbasic.Rect)
    g.DrawRectangle(20, 20, 30, 40)
    g.FillRectangle(60, 20, 30, 40)
```

4) **Run your project.**

Canvas1's Paint event will fire right away, so there's no need to click any buttons. You should see a window that looks like this:



Note that one rectangle is empty while the other is filled. That's the difference between the DrawRectangle method, which only draws the outline of a rectangle, and the FillRectangle method, which colors in the rectangle. They both take the same four parameters: X, Y, Width, and Height. So to draw a square, you would make sure that width and height were the same number.

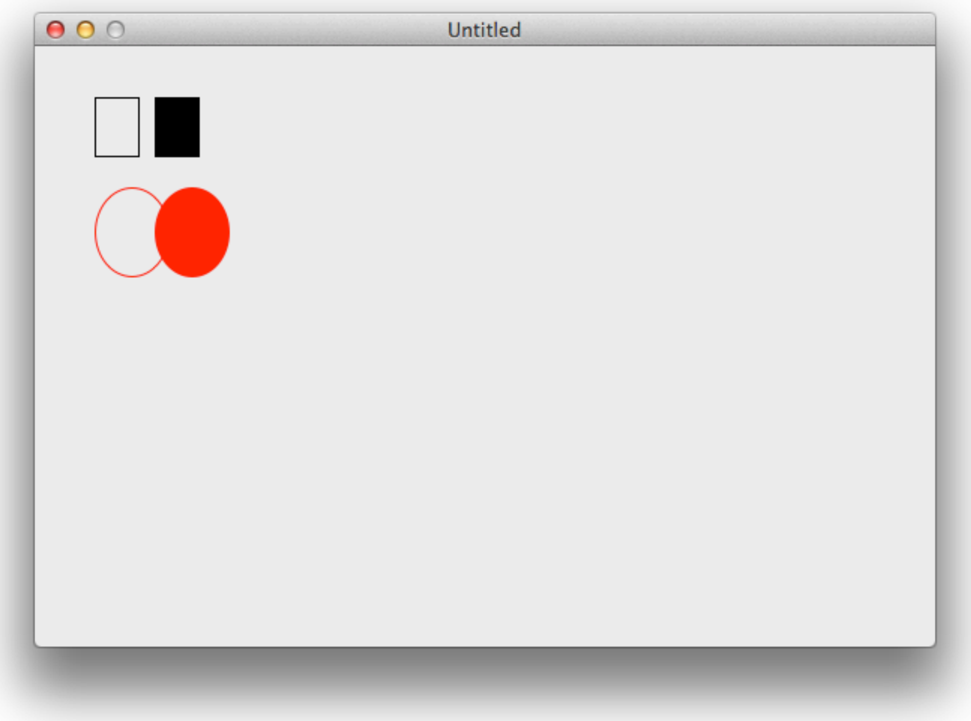
5) **Quit your application.**

6) **Add these lines of code to the end of Canvas1's Paint event:**

```
g.DrawingColor = Color.RGB(255, 0, 0)
g.DrawOval(20, 80, 50, 60)
g.FillOval(60, 80, 50, 60)
```

7) **Run your project again**

You should see a window that looks like this:



Notice that the ovals are red. This is because you set the DrawingColor property of the Graphics class before drawing the ovals. DrawOval and FillOval, like DrawRectangle and FillRectangle, take four parameters: X, Y, Width, and Height.

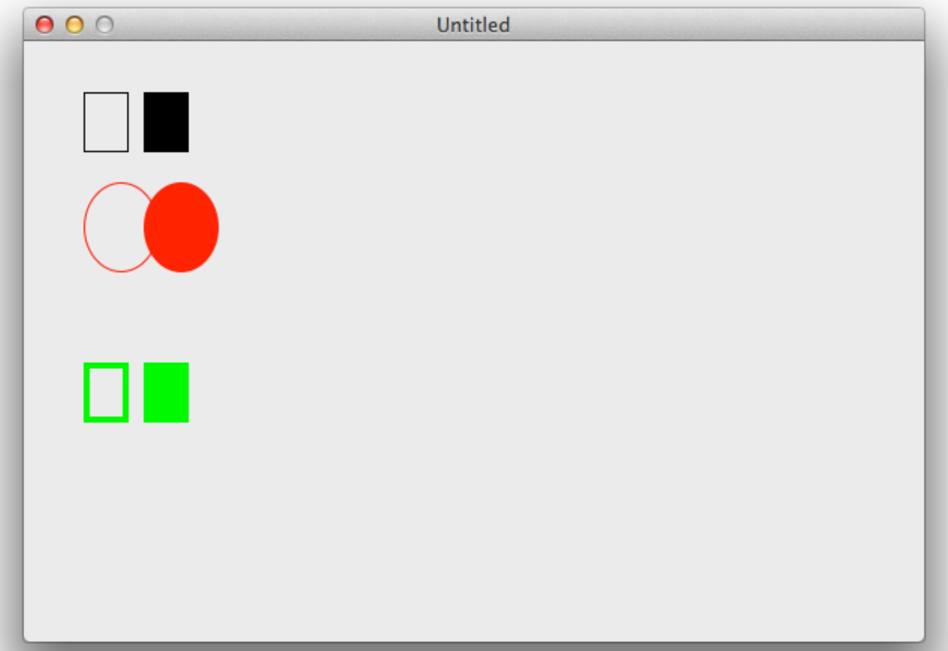
8) Quit your application.

9) Add these lines of code to the end of Canvas1's Paint event:

```
g.DrawingColor = Color.RGB(0, 255, 0)
g.PenSize = 4
g.DrawRectangle(20, 200, 30, 40)
g.FillRectangle(60, 200, 30, 40)
```

This time, you have set the DrawingColor to green and also made the "pen" thicker by setting the PenSize to 4.

10) **Run your project to see the result:**



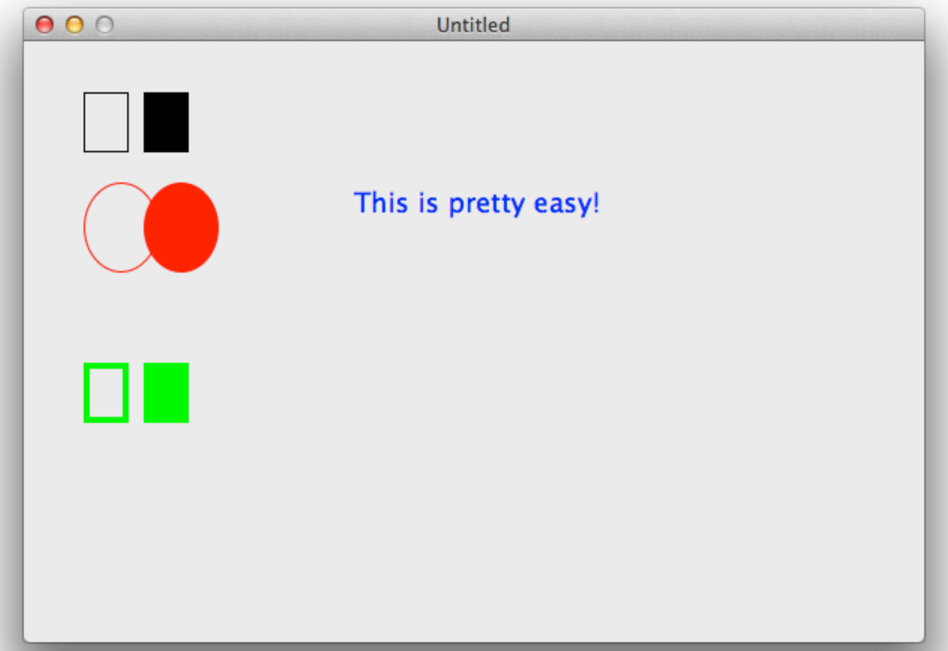
Notice that the green rectangle outline is thicker.

11) **Quit your application.**

12) **Add these lines of code to the end of Canvas1's Paint event:**

```
g.DrawingColor = Color.RGB(0, 0, 255)
g.FontSize = 18
g.DrawText("This is pretty easy!", 200, 100)
```

13) **Run your application, and you should see a window that looks like this:**



The DrawText method draws the text you pass it, as its name implies.

14) Quit your application.

The Graphics class also has several properties related to drawing strings. For example you can set its bold property to true before calling DrawText in order to draw bold text (but don't forget to set it back to false when you no longer need your text to be bold). The same goes for italic and underline. In addition, you can set the FontName and FontSize, and as you saw earlier, the DrawingColor, to control the color of the text.

Now that you've created a masterpiece, you may want to save it. While it might seem strange, there's no way to save the contents of a Graphics object. Only a Picture object can be saved to disk. This is easily dealt with, however, by drawing to a Picture instead of directly to the Canvas.

1) Return to your Drawing project in Xojo and use Save As to create a new project called Drawing-Save.

2) Add a property to the Window: MyPic As Picture.

3) Change the code in Canvas1's Paint event to this:

```
MyPic = New Picture(g.Width, g.Height)
MyPic.Graphics.DrawRectangle(20, 20, 30, 40)
MyPic.Graphics.FillRectangle(60, 20, 30, 40)
MyPic.Graphics.DrawingColor = Color.RGB(255, 0, 0)
MyPic.Graphics.DrawOval(20, 80, 50, 60)
MyPic.Graphics.FillOval(60, 80, 50, 60)
MyPic.Graphics.DrawingColor = Color.RGB(0, 255, 0)
MyPic.Graphics.PenSize = 4
MyPic.Graphics.DrawRectangle(20, 200, 30, 40)
MyPic.Graphics.FillRectangle(60, 200, 30, 40)
MyPic.Graphics.DrawingColor = Color.RGB(0, 0, 255)
MyPic.Graphics.FontSize = 18
MyPic.Graphics.DrawText("This is pretty easy!", 200, 100)
g.DrawPicture(MyPic, 0, 0)
```

4) Run your project.

You should see a window very similar to what you saw before.

5) Quit your application.

6) Improve the quality for HiDPI (Retina) screens.

The above code works fine, but if you have a HiDPI (Retina) screen (and have turned Supports Retina / HiDPI to ON) you may notice that the drawings look a bit fuzzy. This is because the new Picture object (p) that you created does not have the correct scale for a HiDPI screen. To instead get a Picture object with the right scale, call the Window's BitMapForCaching method. Change the "New Picture" line to this:

```
    MyPic = Self.BitmapForCaching(g.Width, g.Height)
```

7) Run your project again and note that the graphics and text appears much sharper.

8) To save the picture, add a Button with this code in its Pressed event:

```
Var f As FolderItem
Var d As SaveFileDialog
d = New SaveFileDialog
f = d.ShowModal
If f <> Nil Then
  MyPic.Save(f, Picture.Formats.JPEG)
End If
```

Now that you have your drawing in a Picture object (the property MyPic), it can be saved as a file. This should look somewhat familiar. The code prompts the user to create a new file, then uses the Picture's Save method to write the picture to a file. Save requires two parameters: the FolderItem to save the Picture as, and the file format. In this example, you will save the Picture as a JPEG.

9) Run your project and click the button to save the drawing.

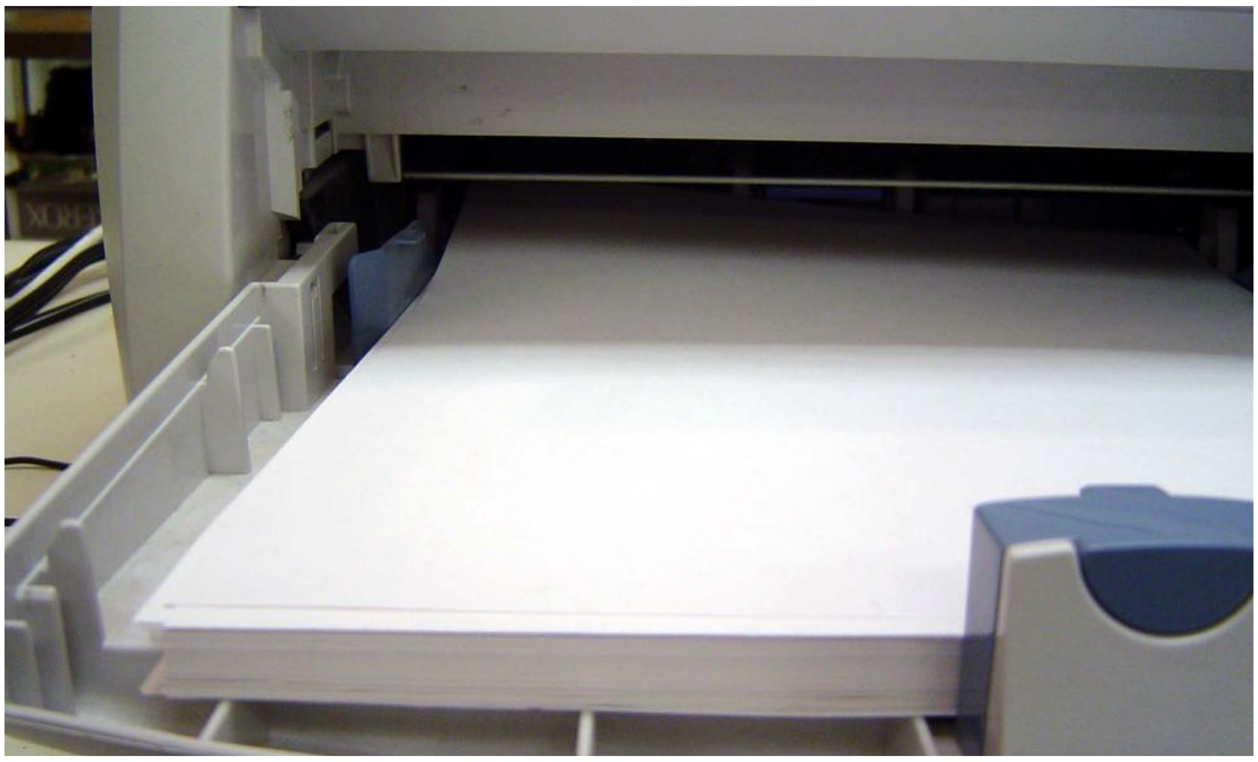
You should be prompted to save your file. Save it as masterpiece.jpg.

10) Quit your application.

Open masterpiece.jpg in any image editor, and you should see the drawing you made using the various methods of the Graphics class.

10.4 Printing

Now that you know how to handle Pictures and Graphics, you know most of what you need to know to print. Printing is done with a Graphics object as well, but it's not part of a Canvas or a Picture. This Graphics object is returned by the PrinterSetup class' ShowPrinterDialog function. ShowPrinterDialog asks the user to confirm that they intend to print, and then gives them a Graphics object. Whatever is drawn onto that Graphics object is sent to the printer.



1) Create a new Xojo desktop project and save it as "Printing".

2) Add a Button to Window1. Add this code to its Pressed event:

```
Var g As Graphics
Var ps as New PrinterSetup
g = ps.ShowPrinterDialog()
If g <> Nil Then
  g.DrawText("This is my first print job!", 100, 100)
End If
```

As you can see, ShowPrinterDialog gives you a Graphics object to work with. You must check to be sure it's not Nil before trying to draw to the Graphics object. If the user presses the Cancel button in the print dialog, g will be Nil, and no drawing or printing should be done.

Once you've verified that g is a valid Graphics object, you may use any of the methods and properties of the Graphics class. The only difference is whatever you draw will be printed rather than displayed onscreen.

3) Run your project.

4) Click the Button.

Your application should print your message.

5) Quit your application.

6) Change the code in the Button’s Pressed event to this:

```
Var g As Graphics
Var ps As New PrinterSetup
g = ps.ShowPrinterDialog()
If g <> Nil Then
  g.DrawText("This is my first print job!", 100, 100)
  g.NextPage
  g.DrawText("This is my second page!", 100, 100)
  g.NextPage
  g.DrawText("This is my third page!", 100, 100)
End If
```

7) Run your project.

8) Click the Button again.

You should see a three page document. You now know how to print multipage documents.

9) Quit your application.

One general rule of printing is this: never assume. You cannot predict what the user will do, so your code needs to be prepared to handle many different situations, such as different page sizes, different margins, and even different page orientations. That’s why it’s best not to use hard-coded positions and sizes for the objects you draw, but to use relative values and scale your drawing proportionally.

You may be thinking that printing a lot of styled text in this fashion would be very tedious, and you’d be right. That’s why the StyledTextPrinter exists.

1) Open the StyledTextEditor project you created in the previous chapter.

2) Add a Button to the window. Set its Caption to “Print” and place this code in its Pressed event:

```
Var g As Graphics
Var ps As New PrinterSetup
Var stp As StyledTextPrinter
g = ps.ShowPrinterDialog
If g <> Nil Then
  stp = EditingField.StyledTextPrinter(g, g.Width)
  stp.DrawBlock(0, 0, g.Height)
End If
```

The TextArea has a function called StyledTextPrinter that returns an instance of the StyledTextPrinter class. The function takes two parameters: a Graphics object (which the ShowPrinterDialog provided) and the desired width of the print area. In this example, the print area will be the entire width of the page, but you could easily reduce that number to print in a narrow column, or even in multiple columns.

The StyledTextPrinter class has a method called DrawBlock, which draws its StyledText into the Graphics object that was specified earlier. DrawBlock takes three parameters: the X coordinate, the Y coordinate, and the height of the block to be printed. Again, in this example, the entire page will be used if needed.

Note: StyledTextPrinter is not supported on Windows.

3) **Run your project.**

4) **Either add some styled text to EditingField or open up another RTF document.**

5) **Once you have some styled text to work with, click the Print button.**

   You should see a printout of your styled text.

6) **Quit your application.**

The Graphics object being used by StyledTextPrinter is just like any other Graphics object, which means that in addition to your styled text, you may draw other shapes and object to it as well.

10.5 Hands On With Printing

Open the Menu project you worked on in Chapter 6. For this chapter's sample project, you will enable the end user to print their food order in addition to displaying it onscreen.

1) **Add a new method to Window1 called "PrintOrder". It will take no parameters.**

2) **In OrderButton's Pressed event, add this line:**

```
PrintOrder
```

3) **Add this code to the PrintOrder method:**

```
Var g As Graphics
Var yOffSet As Integer
Var ps As New PrinterSetup
g = ps.ShowPrinterDialog
If g <> Nil Then
  If MainDishMenu.SelectedRowIndex <> -1 Then
    g.Bold = True
    g.DrawText("Main Dish:", 20, 20)
    g.Bold = False
    g.DrawText(MainDishMenu.SelectedRowValue, 100, 20)
    g.Bold = True
    g.DrawText("Side Order:", 20, 40)
    g.Bold = False
    If FriesRadio.Value Then
      g.DrawText(FriesRadio.Caption, 100, 40)
    End If
    If PotatoRadio.Value Then
      g.DrawText(PotatoRadio.Caption, 100, 40)
    End If
    If OnionRingRadio.Value Then
      g.DrawText(OnionRingRadio.Caption, 100, 40)
    End If
    yOffSet = 60
    If CheeseCheckBox.Value Then
      g.Bold = True
      g.DrawText("Extra:", 20, yOffSet)
      g.Bold = False
      g.DrawText(CheeseCheckBox.Caption, 100, yOffSet)
      yOffSet = yOffSet + 20
    End If
    If BaconCheckBox.Value Then
      g.Bold = True
      g.DrawText("Extra:", 20, yOffSet)
      g.Bold = False
      g.DrawText(BaconCheckBox.Caption, 100, yOffSet)
      yOffSet = yOffSet + 20
    End If
    g.Bold = True
    g.DrawText("Notes:", 20, yOffSet)
    g.Bold = False
    g.DrawText(NotesField.Text, 100, yOffSet, (g.Width - 40))
  End If
End If
```

That's certainly a lot of code - more than you've seen so far in this book. But based on everything you've learned so far, there's nothing new.

One approach to printing this order would be to draw each element of the order using DrawText, working your way down the page and increasing the Y coordinate each time. That's the basic approach this code takes, but bear in mind that your code needs to be smart. If a user doesn't choose a side order or an option, you need to ensure that your application doesn't print a blank line where that part of the order would have been. Also, since the user can pick zero, one, or two options, you need to make sure that there's enough room for all of them.

For this reason, this code keeps track of the vertical position on the page by using a variable called yOffSet. If a line needs to be printed, yOffSet will be increased, and if a line needs to be skipped, yOffSet will be left alone.

This code is very similar to the code that's in the CompileOrder method. The only difference is that instead of adding items to the TextArea, it draws them to the Graphics object. It also toggles Bold to true for the labels and to False for the actual items.

4) **Run your project.**

5) **Choose your order and click the OrderButton.**

You should be prompted to print your order.

6) **Quit your application.**

In this chapter, you learned some valuable skills for dealing with images and graphics, as well as for printing your user's data. You now have another option to offer to your end users as they use your solutions.

Chapter 11

Connections

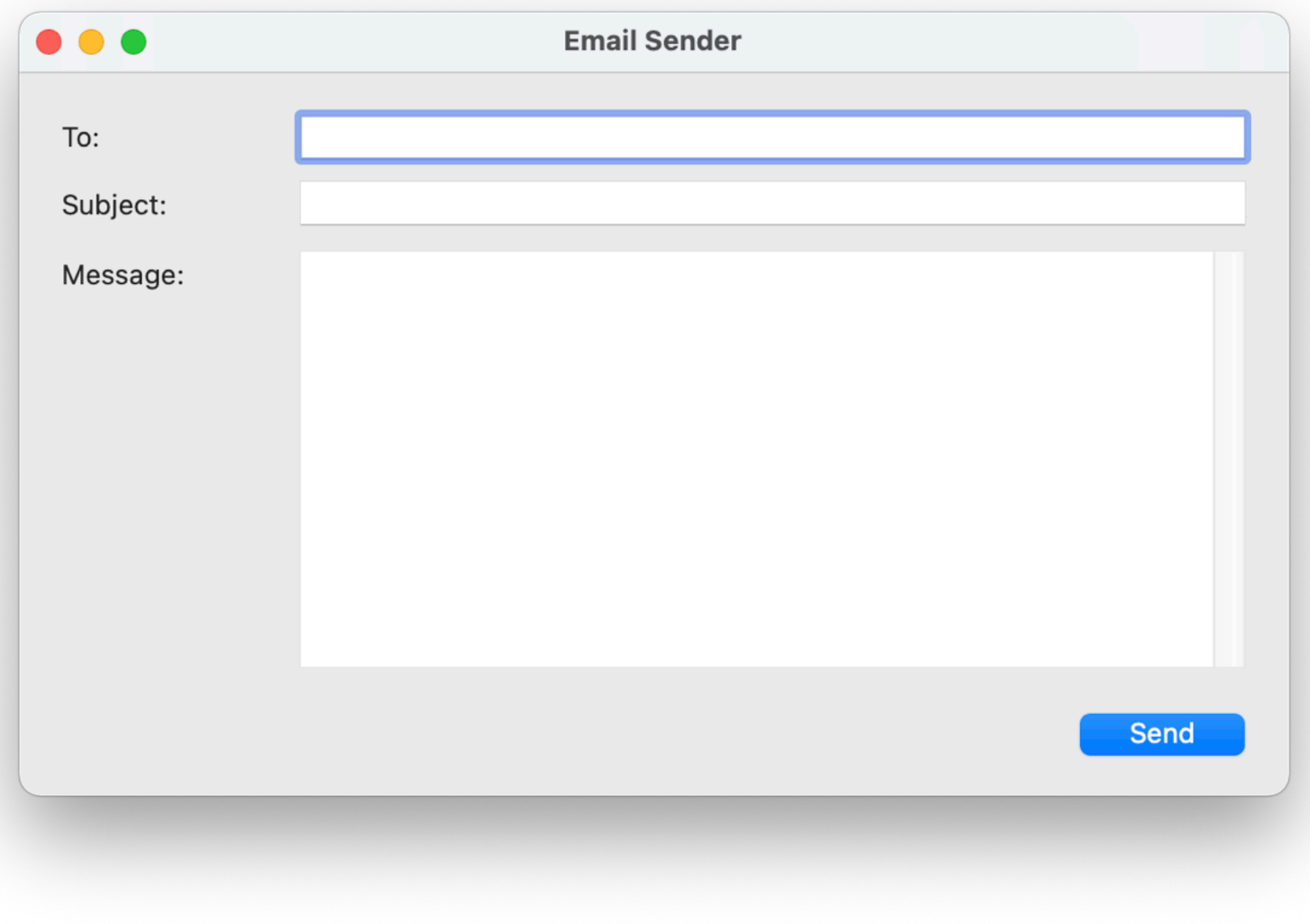
CONTENTS

11.1 Chapter Overview

A lot of what we do with computers and mobile devices today has been enabled by networking, and the Internet in particular.

Networking is the word used to describe how two or more computers or devices “talk” to each other. This may be a “one to one” conversation between two computers transferring a file. It could be a dozen friends playing a multiplayer game over the Internet. Or it could be millions of people using Facebook at the same time to chat and share photos and status updates. At the core, each of these scenarios involves networking.

In this chapter, you will learn how to add some networking capabilities to your own projects. You will also create a sample project called Email Sender, which will, as you probably guessed by the name, send email messages. Your project may look something like this:

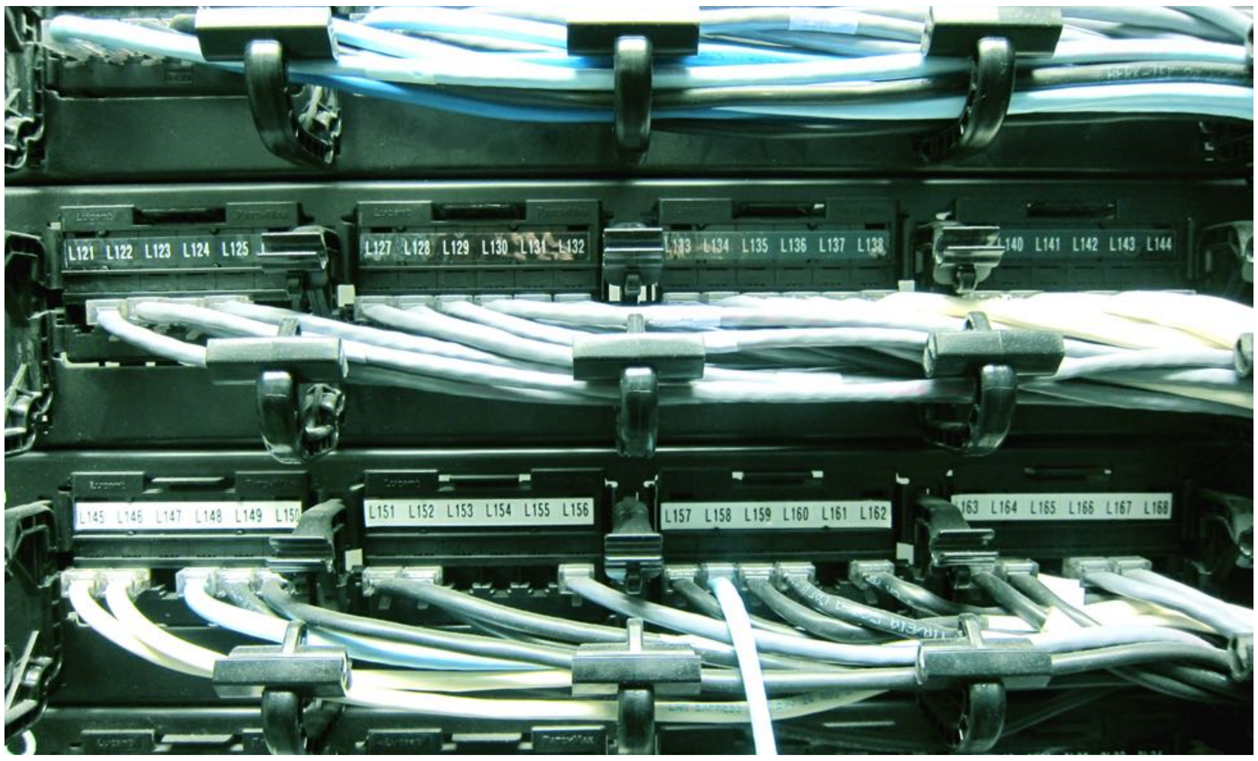


11.2 Networking 101: Protocols, Ports, and Addresses

This section may be, admittedly, a bit dull. But when it comes down to it, there are a few networking terms you need to know and understand before you will be able to add networking to your application.

Imagine going to another country where another language is spoken and customs are different. Now imagine that you do not speak that language, and you are unaware of their customs. If you were to go for a job interview under those circumstances, it is very unlikely that you would get the job. You would be unable to communicate with the interviewer and you may even offend him or her if you try to shake hands.

Networking is similar. Every time a computer or device talks to another computer or device over a network, certain things have to happen. First, both devices must agree on how to communicate. This is called a protocol. A protocol is a set of instructions that dictate how certain communications are supposed to happen. For example, when you use a web browser, you are using a protocol called HyperText Transfer Protocol (also known as HTTP, which is why you often see website addresses beginning with “http”). The documentation for a protocol is typically extremely technical and rather difficult for non-engineers to read. This is not to say that you can’t read up on a protocol and implement it yourself; just be warned that it’s hard, time-consuming work. Fortunately, Xojo includes built-in support for a number of common protocols.



Now imagine that you were standing outside a friend’s house, trying to talk to him or her through a window. For this to be successful, you’d both need to be at the same window. If you were yelling at the living room window while your friend was upstairs, you'd be unable to communicate. But if you both went to the same window, your conversation would be limited only by your imagination. Similarly, a port is "where" the two devices try to communicate. The port is a number, not an actual physical port on your computer (such as the USB port or video port). Every computer connected to a network has thousands of port numbers available. Typically, the first 1,024 ports are reserved for use by the operating system, but many thousands remain available. Most protocols define the port number on which they are designed to operate. For example, the HyperText Transfer Protocol mentioned above almost always operates on port 80. Protocols can operate on other ports when needed, but each protocol has an expected port; if you wish to operate a protocol on a non-standard port, you just need to make sure that both devices are aware of the change.

Finally, each device involved in the communication has an address. Typically, this address can take two forms: the IP address and the DNS name. You may have seen an IP address before (IP stands for Internet Protocol). It's a set of four numbers, each between zero and 255. For example, there's a good chance that the IP address on your home wireless router (if you have one, of course) is similar to 192.168.1.1.

Because these numbers can be difficult to remember, DNS names are often used instead. A DNS name is a series of words (or syllables) that take the place of an IP address. For example, you can easily remember "google.com" as a website address, but you might find yourself doing fewer searches if you had to remember and type in "72.14.204.102" every time you needed Google. DNS stands for Domain Name Service, and there are "master" DNS servers on the Internet that keep track of which names go with which numbers.

Putting all of these together, you may find that your application needs to talk to google.com (or 72.14.204.102) on port 80 using the HyperText Transfer Protocol.

The IP version that most devices use today is version 4, but there is currently a lot of momentum behind switching networks over to IPv6. IPv4 provides 4,294,967,296 possible addresses. That may sound like a lot, but every device on the Internet must have a unique number: every computer, every smartphone, every tablet, and so on. There are a few creative ways to "double up" on IP addresses, but the truth is that we'll soon be out of numbers. IPv6 solves this problem by providing 2^{128} possible unique addresses. That's two to the one hundred twenty eighth power, which is known, in mathematical terms, as an RDN, or Ridiculously Large Number. Formatted as an integer, that number is 340,282,366,920,938,000,000,000,000,000,000,000,000. That's enough for each person on the planet to have 51,557,934,381,960,373,252,026,455,671 addresses of his or her own.

11.3 Making Connections

If that all seems overwhelming, take heart. Xojo has a class called SocketCore that takes care of a lot of these details for you. You don't use SocketCore directly, but rather one of its subclasses (a subclass is a derivative of another class; you'll learn more about subclasses in Chapter 13). These subclasses of SocketCore are generally referred to as Sockets.

In earlier chapters, you learned about using different kinds of controls to create your application's user interface. The Socket is a control, too, but it's an example of a non-visual control. In other words, even after you drag it to your window, your end user won't see it in your application. It still provides you with events that you can respond to, but it has no user interface of its own.

1) Create a new desktop project in Xojo and save it as "Sockets".

2) Add a Button to Window1. Also add a TCPSocket.

Notice that the TCPSocket positions itself below the window editor (in the "Shelf"). This is because it won't be part of your application's user interface.

3) Add this code to the Button's Pressed event:

```
TCPSocket1.Address = "http://www.google.com"
TCPSocket1.Port = 80
TCPSocket1.Connect
```

The code tells the Socket which address ("http://www.google.com") and which port (80) to use. It then tells the Socket to connect to that address. Note that you can also set the Socket's address and port in the Inspector.

4) Add an event handler to the Socket to handle the Connected event. Add this code:

```
MessageBox("You are connected!")
```

This event is called when the Socket has successfully connected to the device at the specified address using the specified port.

5) Run your project. Click the Button.

After a moment, you should see a message box telling you that you are connected.

6) Quit your application.

So far, so good, but admittedly, that project is far from impressive. After all, the whole point of networking is to send information back and forth between devices. To send data with a Socket, you use the Write method.

1) Add a Text Area to Window1. Size it so that it covers most of the window.

2) Change the code in the TCPSocket's Connected event to this:

```
TextArea1.AddText("You are connected!" + EndOfLine)
Me.Write("GET")
```

3) In the TCPSocket's SendProgress event, add this code:

```
Var s As String
s = "Sent " + BytesSent.ToString
s = s + " bytes so far..."
s = s + EndOfLine
TextArea1.AddText(s)
```

4) Run your project and click the PushButton.

The Text Area should show you that your socket has connected successfully, and then it should reveal that you have sent three bytes of data across the network.

5) Quit your application.

Again, this is still less than impressive, but you now know how to send data via a Socket using the Write method. What would be even more impressive, however, would of course be getting some data back. This is where things get decidedly more difficult. And, perhaps not coincidentally, this is where protocols come into play as well. As noted above, a protocol is a set of rules for two devices to communicate over a network, and without a protocol, such "conversations" are difficult at best and impossible at worst.

The TCPSocket you have been working with so far has been operating in something of a vacuum. Without an established protocol, it doesn't know how to talk to the other device, or how to respond to it. More importantly, you don't know how to tell it to do so, unless you've decided to use a protocol.

11.4 Web Connections

As mentioned earlier, Xojo comes with built-in support for some common networking protocols, among them the HyperText Transfer Protocol, or HTTP. The URLConnection control already knows and understands the HTTP protocol, freeing you up to concern yourself with other areas of your app, rather than dealing with the intricacies of managing the network communications.

1) **Create a new Xojo desktop project and save it as "HTTPStuff".**

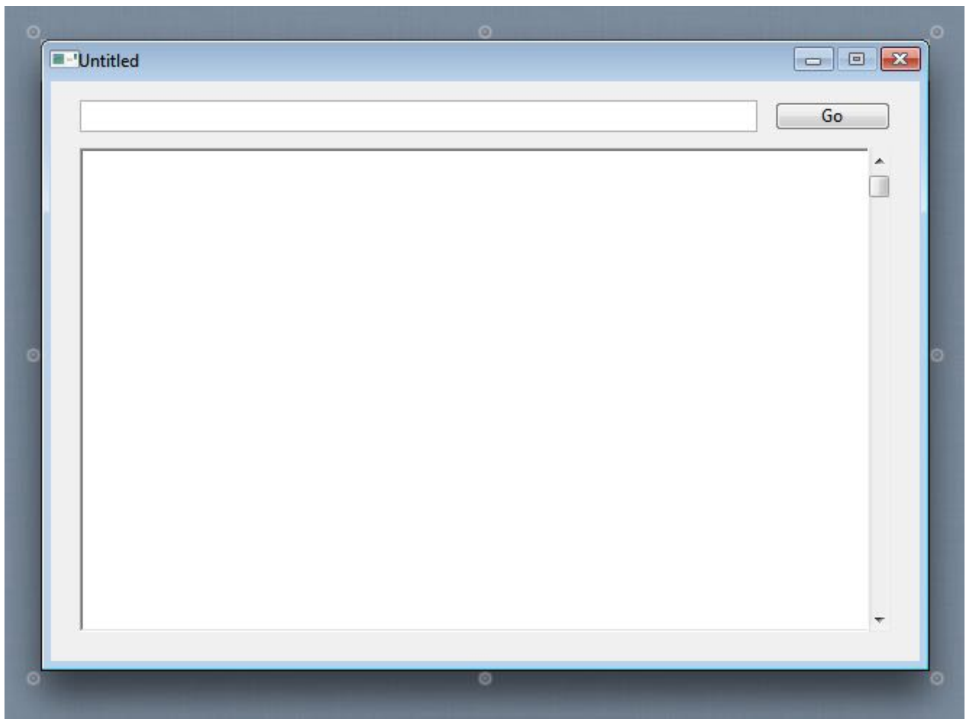
2) **Drag a URLConnection control onto Window1.**

   As before, notice that the URLConnection positions itself below the window editor (in the Shelf). This is because it won't be part of your application's user interface.

3) **Add a Text Field, a Button, and a Text Area to Window1.**

4) **Change the Button's caption to "Go" and size the Text Area so that it takes up most of Window1.**

   As far as the exact arrangement, your interface might look something like this, but feel free to use your own creativity:



5) **Add this code to the Button's Pressed event:**

```
URLConnection1.Send("GET", TextField1.Text)
```

   This code tells the URLConnection to *get* the data (webpage) at the URL in TextField1I. The Get method will result in one of two outcomes: either the URLConnection will successfully retrieve the contents of the page, or it will produce an error.

6) **Enter this code into URLConnection1's Error event:**

```
TextArea1.Text = "Error: " + e.Message
```

In the case of an error, the URLConnection's Error event will fire. The Error event provides you with a RuntimeException (in the e parameter) and the reason for the error is then found in the its Message property.

Of course, the preferable outcome is the successful retrieval of the page contents. This happens in the PageReceived event. The ContentReceived event provides you with four variables: URL, a string indicating the address of the page; HTTPStatus, an integer indicating the status of the transmission; Headers, an instance of the InternetHeaders class, which gives you additional details about the page; and Content, a string which is the actual content of the page in question.

7) **Enter this code into URLConnection's ContentReceived event:**

```
TextArea1.Text = content
```

That code will simply fill up TextArea1 with the contents of the page.

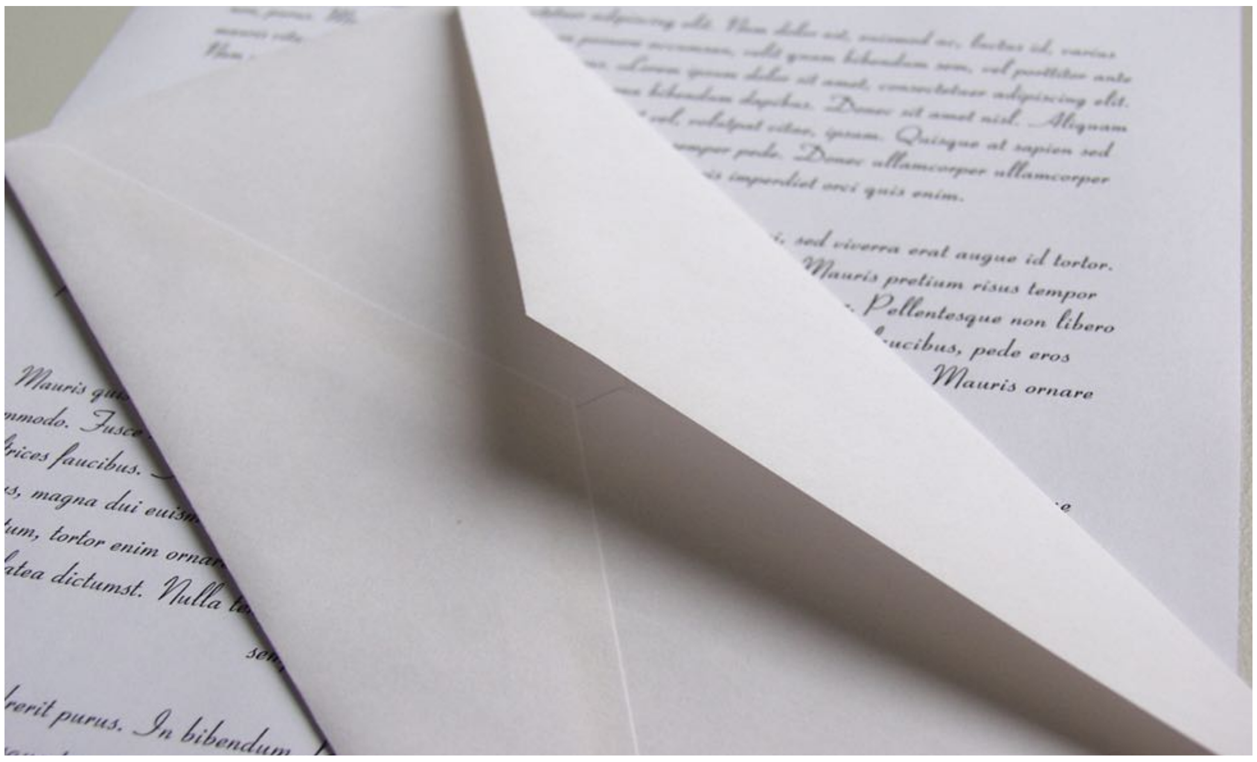
8) **Run your project.**

9) **Enter a website address (such as "https://www.google.com" or "https://www.xojo.com") into the TextField and press the PushButton. Note: on MacOS, you must use HTTPS.**

The page's contents should be displayed as HTML text in the Text Area. If not, you should see an error message. To see an example, enter "microsoft.apple.google" into the TextField and click the PushButton. You will likely see an "Unsupported URL" message in the TextArea. That means URLConnection doesn't understand the protocol because you didn't start it with HTTPS://.

10) **Quit your application.**

11.5 Hands On With Sending Email



In this section, you will create the Email Sender project and also learn about Xojo's SMTPSocket. SMTP stands for Simple Mail Transfer Protocol, and it's the protocol most commonly used to send email messages across the Internet. For this section, you will need an email account through Yahoo or iCloud.

Note that SMTP is only used for sending messages. Receiving messages is usually done through one of two protocols: POP (Post Office Protocol) and IMAP (Internet Message Access Protocol). Because receiving email messages is decidedly more complicated than sending them, this project will focus on sending messages. On the surface, it may seem silly to create an application that can send emails without receiving them, but it's actually quite common. Many applications have built-in support for sending bug reports to the developer; this is often done by sending an email behind the scenes. In addition, many applications have a "Share" option that often includes emailing a link to your friends.

1) **Create a new Xojo desktop project and save it as "MailSender".**

2) **Add a TCPSocket to Window1 and set its name to "MailSocket".**

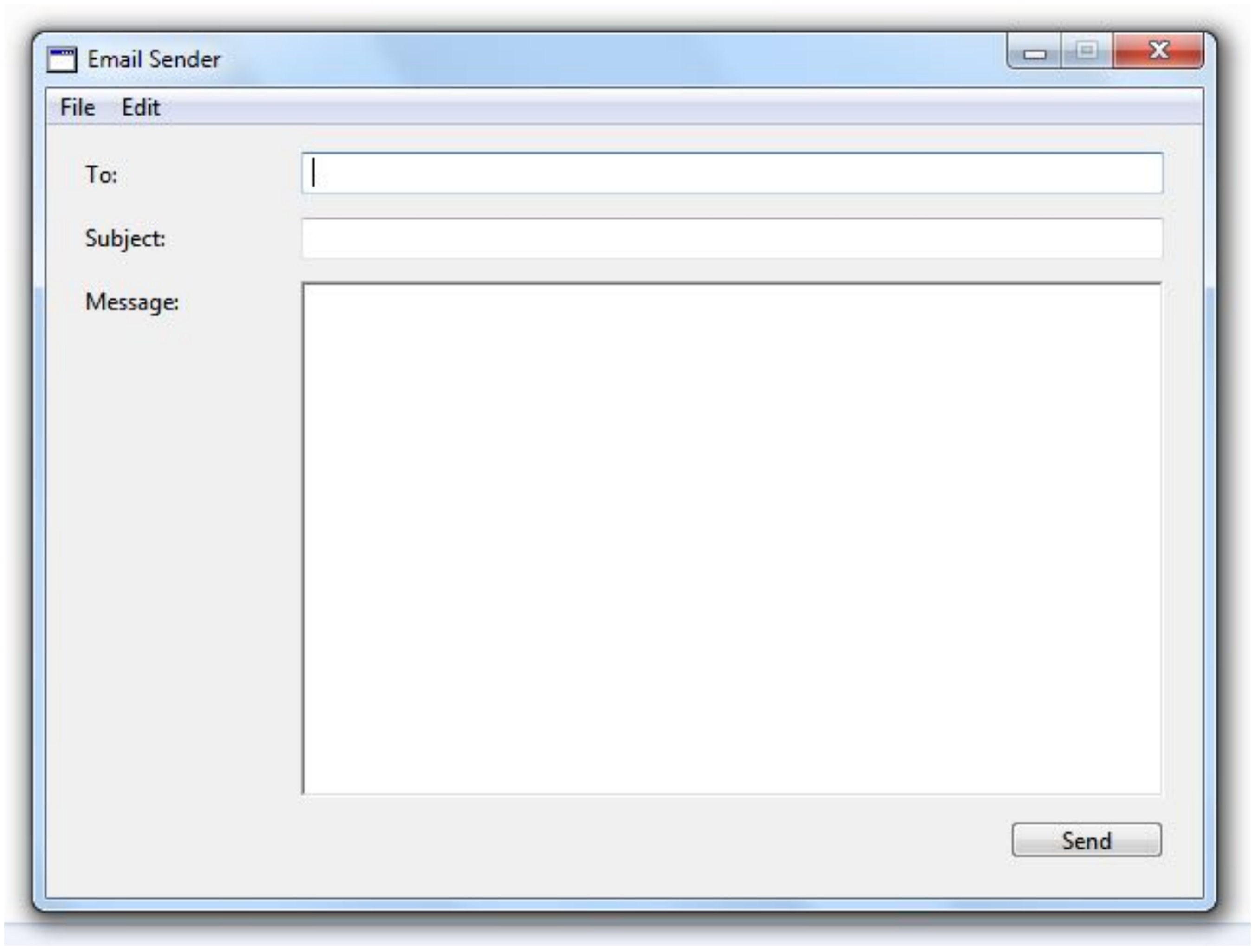
3) **Using the Inspector, set its Super property to SMTPSecureSocket.**

4) **Add two Text Fields, named "ToField" and "SubjectField" to Window1. Add appropriate Labels near them.**

5) **Add a Text Area named "MessageArea", also with an appropriate Label.**

6) **Add a Button with "Send" as its Caption.**

Your interface may look something like this, but as always, feel free to use your own creativity:



7) **Create a new method called “SendTheMessage”.**

```
Var m As EmailMessage
m = New EmailMessage
m.AddRecipient(ToField.Text)
m.Subject = SubjectField.Text
m.BodyPlainText = MessageArea.Text
m.FromAddress = “example@DoNotUseThisExample.com”
MailSocket.Messages.Add(m)
MailSocket.SendMail
```

This method will gather the information from your interface and put it into an EmailMessage object, which will then be handed over to MailSocket for sending. Most of the properties of the EmailMessage class are self-explanatory based on their names. Note that FromAddress property should be set to your own email address.

One interesting method of the EmailMessage class is AddRecipient. This can be used multiple times on a single message to add multiple people to the “to” field of an email.

The SMTPSecureSocket has an array of EmailMessages called Messages. You add your own message to the queue by appending an EmailMessage to that array, as seen in the code above. Once all of the necessary messages have been added, the SendMail method of the SMTPSecureSocket is called. The SendMail method sends all of the EmailMessages in the Messages array. With each message sent, the Socket calls the MessageSent event. This event provides you with a variable called Email, which is an instance of the EmailMessage class representing the last message that was sent. When all messages have been sent, the Socket calls the MailSent event. At this point, the Messages array is empty and the SMTPSecureSocket is ready to be used again.

8) Add this code to MailSocket's MailSent event:

```
MessageBox("All mail has been sent!")
```

9) Add this code to MailSocket's ServerError event:

```
MessageBox(ErrorMessage)
```

Using the MailSent event will help you to keep your end user informed of the app's status instead of wondering if anything has happened. The ServerError event will let your user know if something goes wrong.

Now comes the tricky part. As mentioned earlier, this project requires you to have an email account through iCloud or Yahoo. However, each of these email systems uses a slightly different method of connecting and authenticating, so the code in the Button's Pressed event will be different for each.

10) If you are using iCloud, go to Step 11. For Yahoo! Mail, go to Step 12.

11) iCloud - Enter this code into the PushButton's Pressed event:

```
MailSocket.Address = "smtp.mail.me.com"
MailSocket.Username = "YourICLOUDusername@icloud.com"
MailSocket.Password = "YOUR ICLOUD PASSWORD"
MailSocket.Port = 587
MailSocket.SSLConnectionType = SMTPSecureSocket.SSLConnectionTypes.TLSv1
MailSocket.SSLEnabled = True
MailSocket.Connect
SendTheMessage
```

Note that your user name should be your full email address, including the @ sign. Skip to Step 14.

12) Yahoo! Mail - Enter this code into the PushButton's Pressed event:

```
MailSocket.Address = "smtp.mail.yahoo.com"
MailSocket.Username = "YourYAHOOUsername@yahoo.com"
MailSocket.Password = "YOUR YAHOO PASSWORD"
MailSocket.Port = 465
MailSocket.SSLEnabled = True
MailSocket.Connect
SendTheMessage
```

Note that your user name should be just the portion of your email address that comes before the @ sign.

13) **Run your project.**

14) **Enter an email address, a subject, and a message.**

15) **Click the Button to send your message.**

Did you see a success message or an error?

16) **Quit your application.**

In all the above code listings, your username and password should be surrounded by quotation marks, since they are strings. Also, please note that including passwords and usernames in your source code is usually frowned upon as a bad practice. It doesn't matter much for the purposes of this project, but an interesting extension to this project would be adding additional fields to your interface for your username and password so that they're not stored as part of your source code.

If the above examples for iCloud or Yahoo are not working for you then you may need to check the SMTP settings on those services to make sure you are using their recommended port, security and servers as these can change from time to time.

Yahoo details: https://help.yahoo.com/kb/pop-access-settings-instructions-yahoo-mail-sln4724.html

11.6 A Few Closing Notes About Protocols

Protocols basically come in two forms: existing, established protocols, and protocols that you create on your own. An existing protocol could be something like HyperText Transfer Protocol (HTTP), used to send data for web browsing, or the protocols used for email messaging: Post Office Protocol (POP) and Simple Mail Transfer Protocol (SMTP). These established protocols are usually accepted by technology industry groups.

You can create your own protocols, however taking on such a task is a huge undertaking.

Xojo does offer you something to make creating your own protocol quite a bit easier, as long as the only devices using the protocol will be other computers running applications built with Xojo.

The EasyTCPSocket class allows you to create your own Xojo-only protocols. If you want to learn about the EasyTCPSocket, please see the sample projects that come with Xojo.

Chapter 12

Rows & Columns

CONTENTS

12.1 Chapter Overview

Many people find the thought of working with databases to be intimidating, but databases are everywhere. You may not realize it, but you use databases almost every single day. Every time you search with Google, scroll through the contacts on your phone, or use Twitter, you're accessing a database.

In the simplest terms, a database is an organized collection of data. One key feature of a database is the ability to ask it questions about the data and receive answers. Of course, you must learn to ask the right questions in the right way.

In this chapter, you'll learn a bit about database theory, learn some basics of Structured Query Language (SQL), the language used to talk to databases, and create a sample Address Book application.

12.2 Introduction to Databases

For now, you need to understand four key concepts about databases: tables, columns, rows, and queries.

A database table is somewhat analogous to a class in Xojo (or any other programming language). Just like a class, it represents a real world object (such as a person) or an abstract concept (such as a hotel room reservation).

If you remember back to Chapter 8, you may recall that classes can have properties, and that each property has a certain data type. A database table is very similar, except that these properties are called columns instead of properties. Each column in a table represents some attribute of that object or concepts that it represents. For example, a database table called “people” might have columns for things like the person’s first and last name, the person’s phone number, or the person’s title. As with the properties of a class, the columns in a database table must also conform to a certain data type. Some common data types are listed below, along with their equivalent in Xojo.

Database Data Type	Xojo Equivalent
VARCHAR / TEXT	String
INTEGER	Integer
DATE	DateTime
DOUBLE	Double
BOOLEAN	Boolean

Hopefully you can see that most data types match up almost exactly. Only the textual data types have different names (although some databases support many more data types, these are all you need to worry about until you get into very advanced database work).

With these data types in mind, you could have a database table that looked something like this:

Column Name	Data Type
id	INTEGER
last_name	VARCHAR
first_name	VARCHAR
nickname	VARCHAR
birthdate	DATE

Notice that the sample column names listed above use underscores. Just like variables in Xojo and all other programming languages, database column names may not contain spaces. While most databases support upper and lower case letters in column names, it's common practice in database development to use underscores to separate words in a column name. You are more likely to see mixed-case column names in databases such as SQLite.

You may also recall that a class is really a blueprint, and not really the concept that it represents. In your code, you work with instances of the class, sometimes called objects, created with the New operator. A database, again, is very similar. The table is the blueprint, and each instance of what it represents is called a row. A row in our people table might look like this:

id	last_name	first_name	nickname	birth_date
1	Hewson	Paul	Bono	1960-05-10

You may be wondering what on Earth is up with that birth date. In databases, dates are almost universally stored in what's called ISO format: the four digit year, followed by the two digit month, followed by the two digit day. So the date in the table above is May 10, 1960. The ISO format is a standard adopted by the International Standards Organization. Other date formats can be ambiguous. For example, 12/10/2005 is December 10 to Americans, but October 12 to many other countries.

These three concepts — tables, columns, and rows — are the most critical parts of databases to understand. And in reality, they're not complicated.

12.3 Introduction to Database Queries

But, as mentioned above, there are four concepts you need to know for now, not just three. The fourth is the query. A query is a way of communicating with the database. This is done with something called Structured Query Language, or SQL. Almost every database system in existence speaks SQL or some variant of it. Many database systems have slightly customized versions of SQL, but the basics are the same.

SQL is not the only game in town when it comes to databases. There are a number of SQL alternatives such as Cassandra, MongoDB, CouchDB, and Memcached. While some of these databases are very fast and are designed to operate on a massive scale, it's not very common to find them in use in desktop applications, mobile apps, or websites smaller than Facebook and Google.

Every good language has verbs and SQL is no exception. SQL has four main verbs that you need to know: SELECT, INSERT, UPDATE, and DELETE. If you're wondering why the words are capitalized, it's because it's common convention to use all capital letters for SQL commands. Lower case works fine, too, but as you begin to mix SQL queries in with your Xojo code, the upper case words are easier to pick out.

These four verbs describe their own functions. SELECT is used to get data from the database. INSERT is used to add data to the database. UPDATE is used to modify existing data in the database. And DELETE is used to remove data from the database. These four verbs work on columns and rows. There are a few other verbs for working with tables: CREATE is used to add a table, ALTER is used to modify a table, and DROP is used to delete a table.

To put several of these pieces together, here is some SQL code to create our people table:

```
CREATE TABLE people (
  id INTEGER NOT NULL UNIQUE,
  last_name VARCHAR NOT NULL,
  first_name VARCHAR NOT NULL,
  nickname VARCHAR,
  birthdate DATE,
  PRIMARY KEY(id));
```

The CREATE TABLE part is self-explanatory: it creates a table in the database. The next word, "people", is the name of the table to be created. Inside the parentheses is a list of column names and their respective data types, along with some optional information about each column. For example, you can see that the "last_name" column will be VARCHAR (or text), but what does that NOT NULL following it mean?

That's an example of a constraint. NOT NULL means that the column has to have data in it, even if it's just an empty string. In databases, NULL is equivalent to nothingness, so even an empty string is NOT NULL. NULL means that the value doesn't exist in any way whatsoever. To clarify, if someone asked you a question and you had no answer, that would be similar to NULL, or nothingness. On the other hand, if you had an answer, but opted not to state it, that's more like an empty string (NOT NULL).

Another constraint listed above is on the "id" column: UNIQUE. This means that each row in the table must have a value that no other row has. It has to be, in other words, unique. Using an "id" column in this way is very common in database design, as it gives you a unique identifier for each row, which makes it easier to select, update, and delete specific data later on.

The last line, "PRIMARY KEY(id)", is similar to the NOT NULL UNIQUE constraint on the "id" column. PRIMARY KEY tells the database that it should make sure that "id" exists, is unique, and can always be guaranteed to refer to its row. That is also a constraint, but it's a table constraint rather than a column constraint.

All of these constraints may lead you to wonder why to bother with them. After all, you could write your code in such a way as to make sure that a certain value always exists or that it never conflicts with another value. But the beauty of constraints is that they make the database do that work for you. With a one time setup, you can now trust that the database will raise an error when something goes wrong, rather than relying on yourself to remember every detail later on.

Now that your (admittedly imaginary) "people" table has been created, you need to add some data to it. This is done using the INSERT command. The basic structure of an INSERT looks like this:

```
INSERT INTO [table_name]
  (column1, column2, column3, ... columnN)
VALUES
  ('value1', 'value2', 'value3', ... 'valueN')
```

Even though this example spilled onto multiple lines, a SQL command can be all on line as long as you have a space between each element (such as after the table name).

Obviously, [table_name] should be replaced by the actual name of the table into which you're inserting data. Also, each of the columns listed should be an actual column name. The values listed in the second set of parentheses each correspond to a column in the first set of parentheses, so in the example above, they match up like this:

Column	Value
column1	value1
column2	value2
column3	value3
columnN	valueN

You may specify any number of columns in your INSERT, as long as each one has a matching value.

You may have noticed that our values in the INSERT were surrounded by single quotes. In SQL, every string value (or VARCHAR in database terms) must be surrounded by single quotes. The same rule applies to date values and booleans. Numeric data does not need single quotes. So the above example with real data might look like this:

```
INSERT INTO people
  (id, last_name, first_name, nickname, birthdate)
VALUES
  (1, 'Hewson', 'Paul', 'Bono', '1960-05-10')
```

This raises an interesting question: what happens if your text data has a single quote in it already? If you recall what you learned about string variables in earlier chapters, you can escape a double quote in Xojo by using two double quotes in a row. The same goes for single quotes in SQL (note the two single quotes in the last name):

```
INSERT INTO people
  (id, last_name, first_name)
VALUES
  (2, 'O''Henry', 'Thomas')
```

After running those two INSERT commands, our "people" table would look like this:

id	last_name	first_name	nickname	birth_date
1	Hewson	Paul	Bono	1960-05-10
2	O'Henry	Thomas	NULL	NULL

Note the two NULL values in the second row; that's because those values were not specified during the INSERT.

Now that you have data in the table, you can use the SELECT command to retrieve the data. The SELECT command lists which columns you want, from the table you specify:

```
    SELECT last_name, first_name FROM people
```

That query would return this set of data:

last_name	first_name
Hewson	Paul
O'Henry	Thomas

Note that the database only returns the columns you specify, in the order you specify. The order of the rows, however, is not specified, unless you use the ORDER BY clause:

```
    SELECT last_name, first_name
    FROM people
    ORDER BY last_name, first_name
```

In this particular example, the data set would look the same, but the order of the rows would be guaranteed.

You will sometimes see SQL queries that begin with SELECT *, such as "SELECT * FROM PEOPLE". The * tells the database that you want every single column in the table. This is useful for debugging, but it's not recommended for production code for two reasons. First, you can no longer be sure of the order of the columns you get back from the database. Second, you may be retrieving more data (sometimes far more data) from the database than you need, thereby slowing down your application and using unnecessary disk or network resources.

You may also search for particular rows using the WHERE clause:

```
    SELECT last_name, first_name
    FROM people
    WHERE last_name = 'Hewson'
```

That would return only the first row, because that is the only row in which the last_name column equals 'Hewson'.

The WHERE clause can also be used with ranges of data:

```
    SELECT last_name, first_name, birthdate
    FROM people
    WHERE birthdate > '1900-01-01'
    AND birthdate < '2012-01-01'
```

You can also search for a partial match on a string, using the % symbol (sometimes called a wildcard since it matches any text) and the LIKE operator:

```
    SELECT last_name, first_name
    FROM people
    WHERE last_name LIKE '%nr%'
```

That would return Thomas O'Henry's record.

The wildcard symbol, or %, matches text of any length (including zero characters), so a search for records that match 'B%' would find any records that begin with an uppercase B. A search for records that match '%b' would find any records that end with a lowercase b. A search for records that match '%b%' would find any records that contain a lowercase b anywhere.

Another thing to be aware of when doing SELECTs is that almost every SQL database is case-sensitive, so it treats 'Hewson' differently from the way it treats 'hewson' and the way it treats 'HEWSON'. Each is a different string as far as the database is concerned. To force a case-insensitive search, most databases support the LOWER function, which forces everything to lowercase:

```
    SELECT last_name, first_name
    FROM people
    WHERE LOWER(last_name) LIKE '%hewson%'
```

That would return Bono's record, because the LOWER function has converted everything to lowercase before comparing the strings.

Finally, you can also look for NULL values using the IS operator:

```
    SELECT last_name, first_name
    FROM people
    WHERE birthdate IS NULL
```

Or conversely:

```
    SELECT last_name, first_name
    FROM people
    WHERE birthdate IS NOT NULL
```

Updating existing data is a similar process, using the UPDATE command. Please note that 99.9% of the time you issue an UPDATE command, you will want to use a WHERE clause as well.

```
    UPDATE people
    SET birthdate = '1980-10-01'
    WHERE id = 2
```

You can update multiple columns as well:

```
    UPDATE people
    SET nickname = 'Tommy',
        birthdate = '1980-10-01'
    WHERE id = 2
```

If you eliminate the WHERE clause and issue a command like this then you would change the birthdate on every row in the table:

```
    UPDATE people
    SET birthdate = '1980-10-01'
```

An UPDATE or DELETE command, lacking the WHERE clause, would update or delete every row in the table, which is almost certainly not what you want to do most of the time.

That's worth repeating, with emphasis: *An UPDATE or DELETE command, lacking the WHERE clause, would update or delete every row in the table, which is almost certainly not what you want to do most of the time.*

The WHERE clause in an UPDATE command works the same as it does in a SELECT command, so you may use LIKE, IS NULL, and the other examples you saw above.

Finally, the DELETE command is almost identical to the UPDATE command, with one exception: you do not need to specify your columns, since the entire row is being deleted:

```
    DELETE FROM people WHERE id = 2
```

That would delete Thomas O'Henry's record. To clear out all rows:

```
    DELETE FROM people
```

Remember, omitting the WHERE clause on a DELETE command will delete everything in the table. If that's what you intend to do, that's very useful, but use such a command with extreme caution.

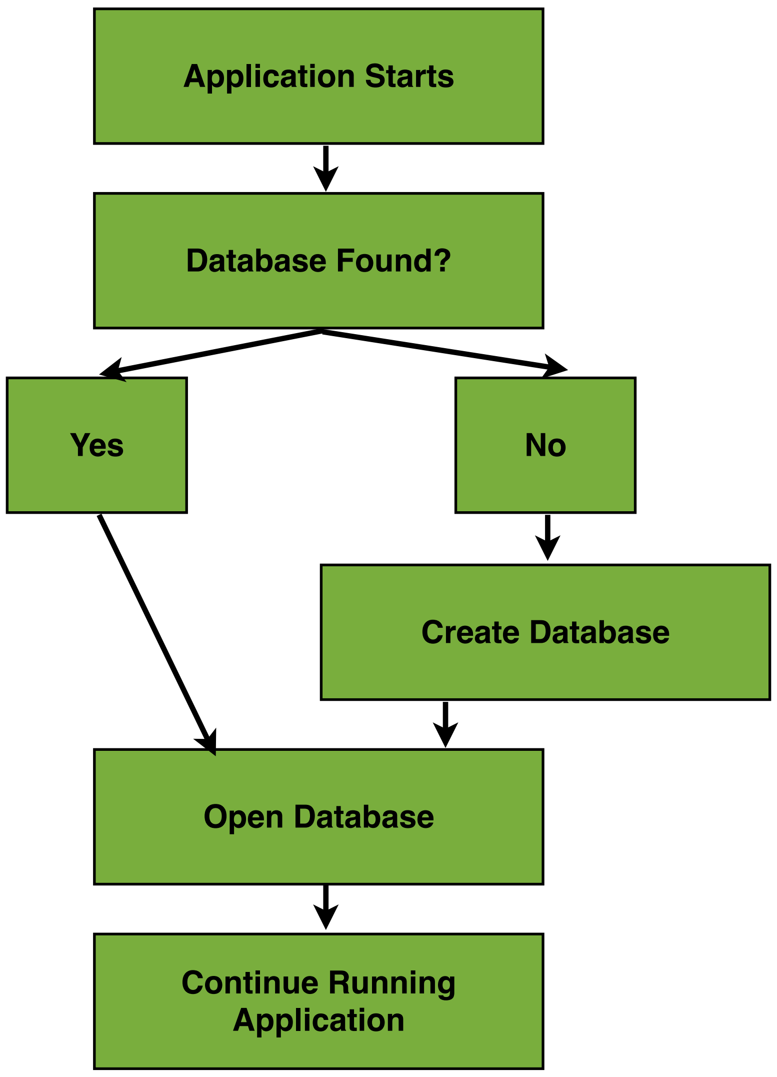
12.4 Creating and Connecting to a Local Database

Now that you have some SQL basics down, it's time to learn how to create a database file in Xojo. There are two types of database connections: local and remote. This textbook will help you learn how to create and work with a local database file, which is simply a file on your computer that holds all of your data.

A remote database, or database server, is a different beast. Although you interact with it the same way, a remote database is running on a database server, usually on a different computer. Typically, multiple people can access the database server simultaneously. Some common database servers you might encounter include Microsoft SQL Server, Oracle, and open source databases like MySQL and PostgreSQL. These types of databases may require specialized hardware and software, and usually need an experienced database administrator to maintain them because they can be quite complex.

But a local database just needs a file. Over the next few sections, you'll learn how to create a local database, how to connect to it, and how to extract data from it. In the process, you'll build this chapter's sample project, a simple email address book.

Create a new Xojo project and save it as "AddressBook". The following flowchart explains how the application will work:



1) Give Window1 two new properties: MyDatabase as SQLiteDatabase and MyDBFile As FolderItem.

MyDatabase will be the variable that holds a reference to the database you'll be using, and MyDBFile will represent the database file on your computer.

2) Create a new method in Window1 called "CreateSchema".

```
Var sql As String
sql = "CREATE TABLE addressbook ("
sql = sql + "name varchar, "
sql = sql + "email varchar"
sql = sql + ")"
Try
  MyDatabase.ExecuteSQL(sql)
Catch error As DatabaseException
  MessageBox(error.Message)
End Try
```

This method will have the job of creating our database table. To do this, you will create a string containing the SQL commands. To create the table (as you saw above) you pass the sql string to a method of the database called ExecuteSQL, which, as its name implies, executes a SQL command on the database.

The Try Catch block above tells your code what to do should ExecuteSQL fail and cause an exception (error). You'll learn more about this in chapter 14.

Creating a schema in code isn't uncommon, but many people prefer to use a visual tool. Many applications for creating schemas are available, including MySQL Workbench, Oracle's SQL Developer, and Microsoft's SQL Server Management Studio. Most of these tools are specific to one particular database server, but there are also tools that are more "database agnostic," especially in the open source community.

3) Create another method in Window1 called "CreateDatabase" with a return type of Boolean.

```
MyDBFile = SpecialFolder.ApplicationData.Child("Ch12.sqlite")
If MyDBFile <> Nil Then
  MyDatabase = New SQLiteDatabase
  MyDatabase.DatabaseFile = MyDBFile
  Try
    MyDatabase.CreateDatabase
    MyDatabase.Connect
    CreateSchema
    Return True
  Catch error As DatabaseException
    Return False
  End Try
End If
```

This method will return True if it successfully creates the database, and False if not. This method works by assigning a FolderItem to the database's DatabaseFile property. Creating the database, connecting to it and creating the schema are all done in a Try Catch statement. This is designed to try to execute commands and if any fail, catch that failure and execute whatever code you want when a failure occurs. In this case, it returns False when a failure happens. Creating the database is done with a method called CreateDatabase. Next it tries to connect to the new database using the Connect method. Last but not least it calls the CreateSchema method you wrote to build the addressbook table.

This code uses the SpecialFolder module to get a system file location. In this case SpecialFolder.ApplicationData refers to these folders, depending on OS:

	Mac	Windows	Linux
ApplicationData	/Users/UserName/Library/Application Support	\Users\UserName\AppData\Roaming\	/home/UserName/

4) Add a method called “OpenDatabase” with a return type of Boolean.

```
MyDBFile = SpecialFolder.ApplicationData.Child(“Ch12.sqlite")
If MyDBFile.Exists Then
  MyDatabase = New SQLiteDatabase
  MyDatabase.DatabaseFile = MyDBFile
   Try
    MyDatabase.Connect
    Return True
  Catch error As DatabaseException
    MessageBox(“Error connecting to database.”)
  End Try
Else
  Return CreateDatabase
End If
```

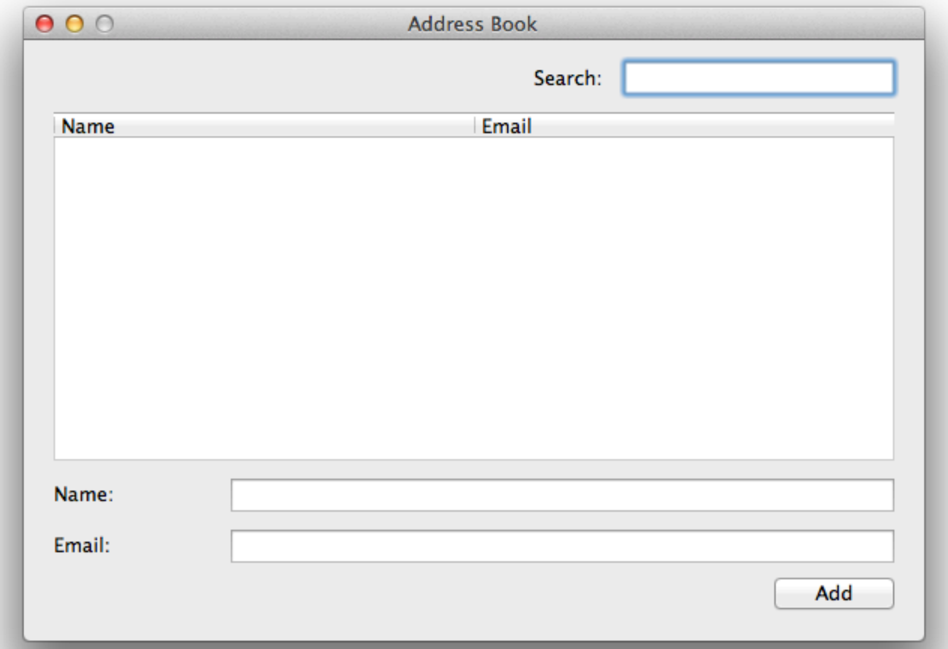
This method, which returns a Boolean, will attempt to open the database file. If it succeeds, the app continues as normal. If not, it runs the CreateDatabase method and returns its boolean result.

5) To start this process, add this code to Window1’s Opening event:

```
If Not OpenDatabase Then
  MessageBox("Unable to open or create database.")
  Quit
Else
  Populate // You’ll create this method next
End If
```

12.5 Accessing The Xojo Database

Now that your database file is setup and your schema is in place, you need to learn how to read information from the database and insert information into the database using Xojo. The code you entered in Window1's Open event, at the end of the last chapter refers to a method you haven't yet created, called Populate. It will be the job of the Populate method to talk to the database and retrieve any data it can find, then place that data into a ListBox. So, before continuing, you need to add a ListBox to Window1. You'll also need a few other controls, and while you're adding the ListBox, it's a good time to lay out the rest of the sample project. Use your own creativity to design your interface. It might look something like this:



Here is a list of the controls you'll need, as well as the names you should give them:

Control	Name
List Box	AddressBox
Text Field	FindField
Text Field	NameField
Text Field	EmailField
Button	AddButton
Label	SearchLabel

Also, add a Label for NameField, a Label for EmailField, and a Label for FindField.

You're almost ready for the Populate method. One thing you need first, however, is a small function called SQLify. If you recall from earlier in the chapter, SQL uses the single quote as a text delimiter, meaning any time you have a single in your data, things can go terribly wrong. That's why you need to “escape” every single quote by doubling it. You could do that manually every time you talk to the database by using Xojo’s ReplaceAll function, but you’d end up typing a lot of the same code over and over, which violates one of the laws of programming: Don’t Repeat Yourself. Instead, create a method in Window1 called “SQLify”. It will take one parameter, **source As String**, and it will return a String. Its code uses the ReplaceAll function to find any single quote and replace it with two single quotes:

```
    Var result As String
    result = source.ReplaceAll("'", "''")
    Return result
```

So if you ran this code, for example:

```
    MessageBox(SQLify("If it ain't broke, don't fix it!"))
```

You would see a message box with this text:

```
    If it ain''t broke, don''t fix it!
```

Note that you’re not seeing regular quotation marks embedded in the text: those are two single quotes side by side. It can be difficult to see in certain fonts.

Now that SQLify is in place, you’re ready for the Populate method. The Populate method will call upon your existing knowledge of the ListBox (specifically the RemoveAllRows and AddRow methods) and will introduce the RowSet class. A RowSet is a batch of data returned by a database query (using the Database class’s SelectSQL function). A RowSet can contain multiple rows. Each row is dealt with one at a time. You navigate through the rows using the RowSet’s MoveToNextRow method. The RowSet’s AfterLastRow property will be true when your code moves beyond the last row. Each row within the RowSet can contain multiple columns, which you can retrieve by name using the RowSet’s Column method. In that method, you use the same column name that is specified in the database schema.

Once the RowSet is retrieved, the Populate method will create a new row in the ListBox for each row in the RowSet, filling in its data as it goes. But before adding new rows, all existing rows will be removed. If you were to skip this step, your ListBox would display duplicate data, and would continue to add more duplicates each time it’s populated.

The Populate method will also build a SQL query. If nothing is entered in FindField, the query will be straightforward, simply selecting all data from the “addressbook” table. However, if the user has entered anything into FindField, Populate will take that into account and return only matching rows (FindField will operate on both the name and email columns).

With that introduction in mind, here is the code for the Populate method:

```
    Var sql As String
    Var rs As RowSet
    sql = "SELECT name, email "
    sql = sql + "FROM addressbook "
    If FindField.Text <> "" Then
      sql = sql + "WHERE LOWER(name) LIKE LOWER('%" + _
         SQLify(FindField.Text) + "%') "
      sql = sql + "OR LOWER(email) LIKE LOWER('%" + _
         SQLify(FindField.Text) + "%') "
    End If
    sql = sql + "ORDER BY name"
    Try
      rs = MyDatabase.SelectSQL(sql)
      AddressBox.RemoveAllRows
      While Not rs.AfterLastRow
        AddressBox.AddRow(rs.Column("name").StringValue)
         AddressBox.CellTextAt(AddressBox.LastAddedRowIndex, 1) = _
           rs.Column("email").StringValue
        rs.MoveToNextRow
      Wend
    Catch error As DatabaseException
      MessageBox(error.Message)
    End Try
```

If you haven't done so recently, this is a good time to save your project.

Now that you can retrieve data from the database, you need to add some data to be retrieved. That will be accomplished using the Pressed event of AddButton. The code that you place there will build a SQL statement to insert the data from the NameField and EmailField controls into the database, using the Database class's ExecuteSQL method. If you're wondering about the difference between SelectSQL and ExecuteSQL, it's primarily about what you want back from the database. If you're trying to retrieve data and need a RowSet, use SelectSQL. If you're inserting, updating, or deleting data, ExecuteSQL is fine, since you don't need a RowSet in those cases.

The code will also display any errors that may occur. If no errors are found, it will commit the database, clear out the data entry fields, and run the Populate method.

Here is the code for AddButton's Pressed event:

```
    Var sql As String
    sql = "INSERT INTO addressbook ("
    sql = sql + "name, email"
    sql = sql + ") VALUES ("
    sql = sql + "'" + SQLify(NameField.Text) + "', "
    sql = sql + "'" + SQLify(EmailField.Text) + "'"
    sql = sql + ")"
    Try
      MyDatabase.ExecuteSQL(sql)
      NameField.Text = ""
      EmailField.Text = ""
      NameField.SetFocus
      Populate
    Catch error As DatabaseException
      MessageBox(error.Message)
    End Try
```

If you find that you're not a fan of putting all of those SQL queries together, you can also use Xojo's Database API to insert the record without using SQL. This code is an alternate to the code above (don't run them both!).

```
    Var row As DatabaseRow
    row = New DatabaseRow
    row.Column("name") = NameField.Text
    row.Column("email") = EmailField.Text
    Try
      MyDatabase.AddRow("addressbook", row)
    Catch error As DatabaseException
      MessageBox(error.Message)
    End Try
```

Finally, you need to make sure that the Populate method is run every time your end user enters text into FindField. This is easily accomplished by adding this line of code to FindField's TextChange event:

```
    Populate
```

Because the Populate method accounts for whatever text may be in FindField, no other work is necessary there.

Run your project and try to add some data to your database. After adding some rows, try out your search function as well. Quit your application.

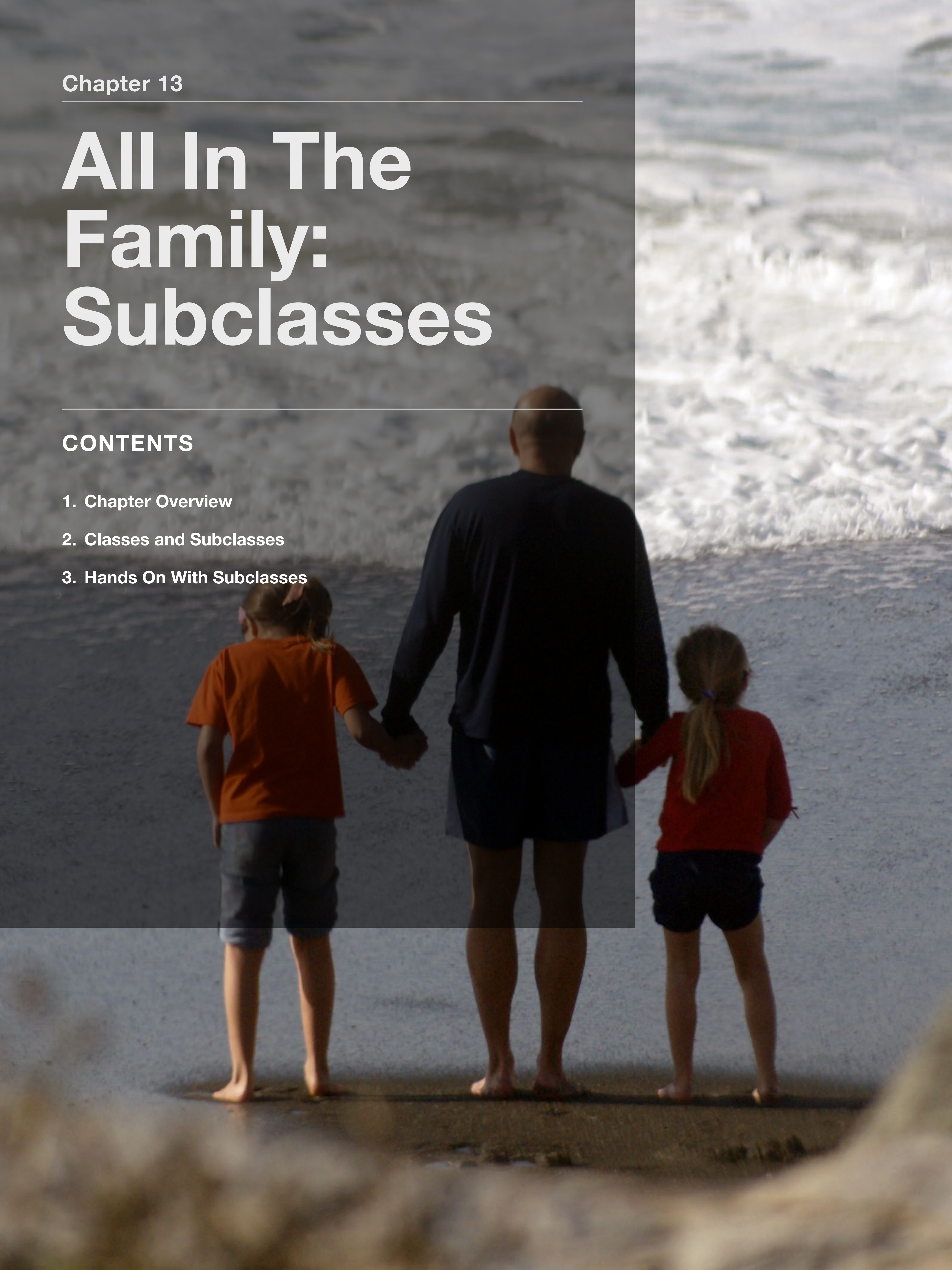
12.6 More About Databases

In this chapter, you have learned a great deal about databases, and how to interact with them using Xojo. Although you have encountered a lot of information, you have only begun to scratch the surface of databases. Since databases are often an integral part of modern applications, you are highly encouraged to learn more about database theory and more about SQL.

Chapter 13

All In The Family: Subclasses

CONTENTS



13.1 Chapter Overview

Did you know that your code is kind of like a family? That may sound odd, but it’s true. Just as you have inherited certain physical characteristics or personality traits from your parents, pieces of your code can inherit properties and behaviors from their parents.

One big difference is that in software development, one doesn’t refer to children and parents. They are subclasses and superclasses.

In Chapter 8, you learned about classes. As you read there, a class is something that represents a real life object (something you can touch or point to) or an abstract idea (a concept or “intangible” idea). Of course, in the real world, outside of your code, things aren’t always that simple. Sometimes different kinds of objects have attributes that overlap.

In this chapter, you’ll learn how different classes can relate to each other, and you’ll create a custom control that you can reuse in other projects.

13.2 Classes and Subclasses

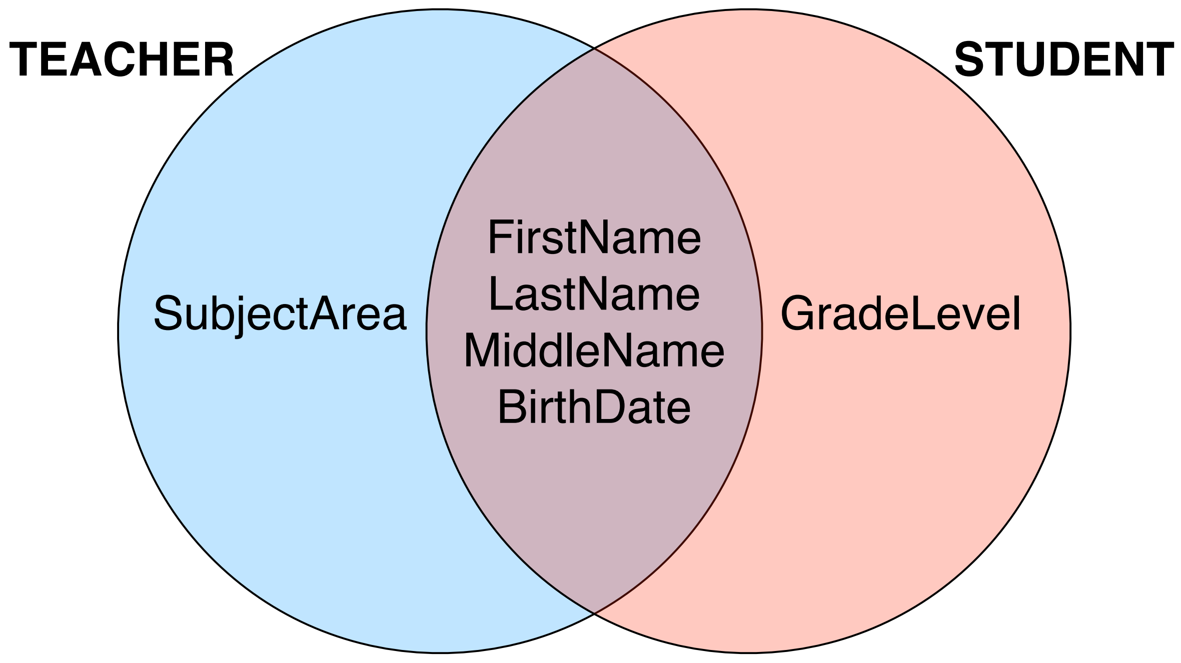
Think back to the Student class that you created in Chapter 8. You gave it the following properties:

Property	Data Type
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
GradeLevel	String

These properties are certainly relevant to a student, but some could easily apply to other things, even things that may need to be represented in your application, like teachers. In fact, a Teacher class might have these properties:

Property	Data Type
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime
SubjectArea	String

Here's another way to look at it:



As you can see, some of the properties are the same. In fact, in this example, most of them are.

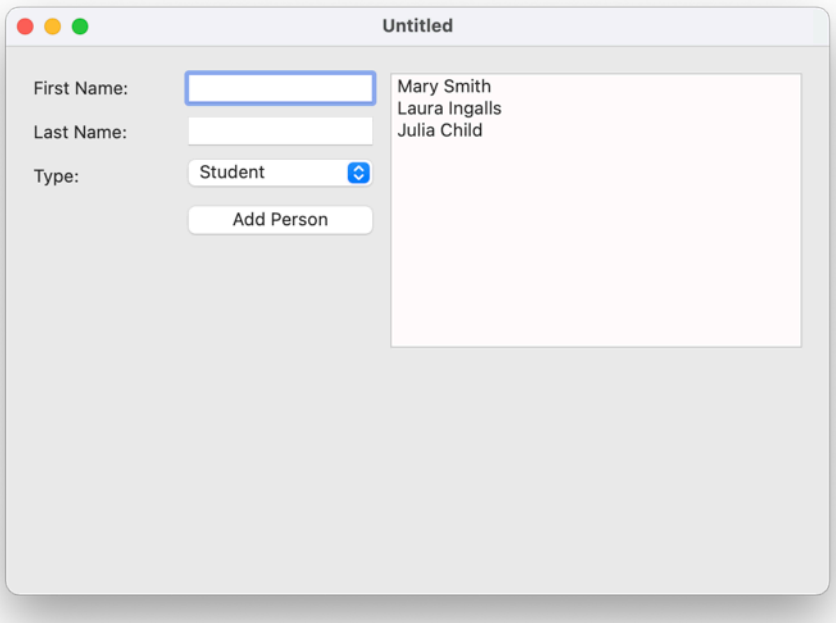
This is because a student and a teacher are both *people*, and most people have some attributes in common with one another. So what this situation calls for is a new class called Person. The Person class will have the common properties we saw above:

Property	Data Type
FirstName	String
LastName	String
MiddleName	String
Birthdate	DateTime

But you still need a way to store information and behaviors for students and teachers. This is where subclasses come into play. A subclass is a class that derives properties and behaviors from another class, called a superclass. This is similar to the parent/child relationship described above.

You don't need to do anything special to create a superclass. Every class you create (and in fact, many that are built into Xojo already) can potentially be a superclass. A class becomes a superclass as you create a subclass from it.

Create a new Xojo desktop project and save it as Subclasses. You will be building a small application that lets you add students and teachers to a common list, while still retaining information about them. Here is a preview of the interface (although yours may look different):



1) **Create a new class by going to the Insert Menu and choosing Class. Name the class "Person".**

2) **Give the Person class four public properties: FirstName As String, LastName As String, MiddleName As String, and BirthDate As DateTime.**

3) **Give the Person class a new method called "AnnounceName".**

```
    MessageBox("Hello, my name is " + FirstName + " " + LastName + "!")
```

4) Give the Person class another method called “SetName”.

```
FirstName = fName
LastName = lName
```

This method takes two parameters: `fName As String` and `lName As String`. It allows you to set both the FirstName and LastName properties with one line of code.

5) Create another new class, this one called “Student”.

When you create the class, set its Super property to Person. You may do this by typing “Person” into the Super field of the Inspector or by clicking the “edit” button and scrolling through the list provided. Student is now a subclass of Person. Because Student is a *subclass* of Person, it automatically inherits the properties and methods of Person. So Student has properties for FirstName, LastName, MiddleName, and BirthDate, and it has the methods AnnounceName and SetName. Note that even though it has these methods, they do not display within the Student class.

6) Give Student a new property: GradeLevel As String.

7) Repeat the process above to create another subclass of Person, called “Teacher”.

8) Give Teacher a new property: SubjectArea As String.

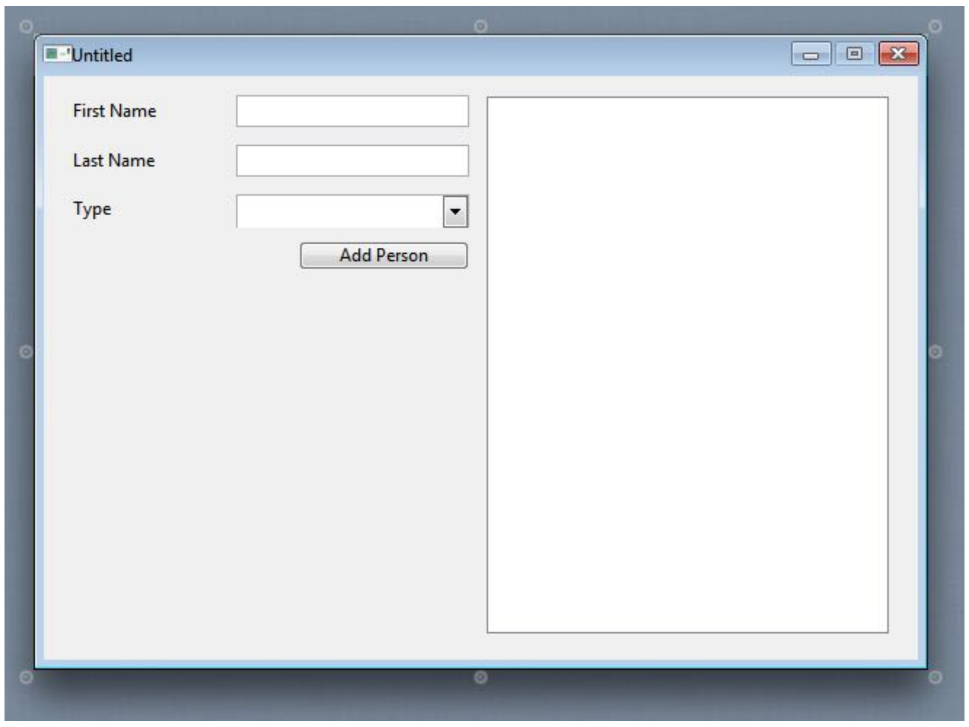
9) Go back to the Person class and add another method, called “AnnounceType”.

```
If Self IsA Student Then
  MessageBox("I'm a student!")
Else
  MessageBox("I'm a teacher!")
End If
```

The job of this method is to determine whether the active Person object is a Student or a Teacher, and then display the results in a message box. To determine this information, you will use IsA, an operator built into Xojo that will tell you whether one object is also another kind of object. In this example, you’ll use it to see if the active Person object is also a Student or a Teacher.

10) Lay out your application’s interface.

Again, it may look something like this, but feel free to use your own creativity when designing yours:



11) **Add these controls to the window.**

Control	Name
Text Field (w/Label)	FirstNameField
Text Field (w/Label)	LastNameField
Popup Menu (w/Label)	TypeMenu
Button	AddButton
List Box	PeopleList

12) **To populate TypeMenu with a list of options, add this code to its Open event:**

```
Me.AddRow("Student")
Me.AddRow("Teacher")
Me.SelectedRowIndex = 0
```

13) **Add this code to AddButton's Pressed event:**

```
Var t As Teacher
Var s As Student
Var newName As String
newName = FirstNameField.Text + " " + LastNameField.Text
If TypeMenu.SelectedRowValue = "Student" Then
  s = New Student
  s.SetName(FirstNameField.Text, LastNameField.Text)
  PeopleList.AddRow(newName)
  PeopleList.CellTagAt(PeopleList.LastAddedRowIndex, 0) = s
Else
  t = New Teacher
  t.SetName(FirstNameField.Text, LastNameField.Text)
  PeopleList.AddRow(newName)
  PeopleList.CellTagAt(PeopleList.LastAddedRowIndex, 0) = t
End If
```

```
    FirstNameField.Text = ""
    LastNameField.Text = ""
    FirstNameField.SetFocus
```

This code will need to create a new Person object (by creating either a Student or Teacher object) and then add that Person's name to the ListBox. It will also place the newly created object itself in one of the ListBox's CellTags. Before you create the object, you'll need to know whether to instantiate a Teacher or a Student, so you'll need to check what TypeMenu says. Finally, in a bit of cleanup, FirstNameField and LastNameField should be cleared and the focus should be set to FirstNameField.

14) Add this code to PeopleList's DoublePressed event:

```
Var p As Person
If Me.SelectedRowIndex <> -1 Then
  p = Me.CellTagAt(Me.SelectedRowIndex, 0)
  p.AnnounceName
  p.AnnounceType
End If
```

This code enabled PeopleList to display information about the object that it holds. Because you stored the Teacher or Student object in a CellTag, you can retrieve that object and access its properties and methods. Remember that a CellTag is a variant, so it can contain any object without displaying in the user interface.

15) Run your project.

16) Add some students and some teachers to your list.

Your students and teachers should show up in the List Box. Double-click on a few entries and see if they report their types correctly.

17) Quit your application.

While the above example is admittedly a bit contrived, it nonetheless illustrates some valuable information about subclasses. First, notice how the Student and Teacher subclasses had all of the properties and methods of their superclass. This is called inheritance. Second, and this is because of inheritance, subclassing can save you a lot of time: rather than adding identical methods and/or properties to various classes, you abstract them into a superclass and create subclasses as necessary (remember one of the cardinal rules of programming: *Don't Repeat Yourself!*).

13.3 Hands On With Subclasses

Speaking of not repeating oneself, subclassing is also a useful way to create custom controls, or controls that exhibit specific behavior.

For example, consider the Label control. As it stands, it's handy for indicating the purpose of a neighboring control, such as a Text Field or Popup Menu, but it doesn't really do anything interesting, at least by default. Suppose you wanted your end user to be able to click on a label to see more information about something, or to visit a URL that you could specify in your code. It would be a relatively simple exercise to implement this behavior.

In broad strokes, you would need to add a Label to a Window. You would need to implement its MouseDown event; remember that returning True in MouseDown causes MouseUp to be called. MouseUp is where you would use the ShowURL method to open the URL in question, which you would have to provide in your code. If you wanted to, you could also implement the Label's MouseEnter and MouseExit events (which are called when the mouse moves over the control and when it leaves the control, respectively) to change the color of the text in the label and even change the mouse cursor to the standard "pointing finger" cursor that some web browsers use to indicate links (this would be a very good visual indicator for your users that the item is clickable).

The steps outlined above would work well, but what if you needed the same functionality from a Label in a different Window? You would need to repeat all of the same steps above, possibly only changing the URL to be visited.

That's a lot of duplicate effort, which means a lot of wasted time and increases the possibility of bugs. This is a perfect situation for subclassing.

Create a new Xojo desktop project and save it as "CustomControl".

1) **Drag the Label from the Library on the right side of the screen to the Navigator on the left side of the screen.**

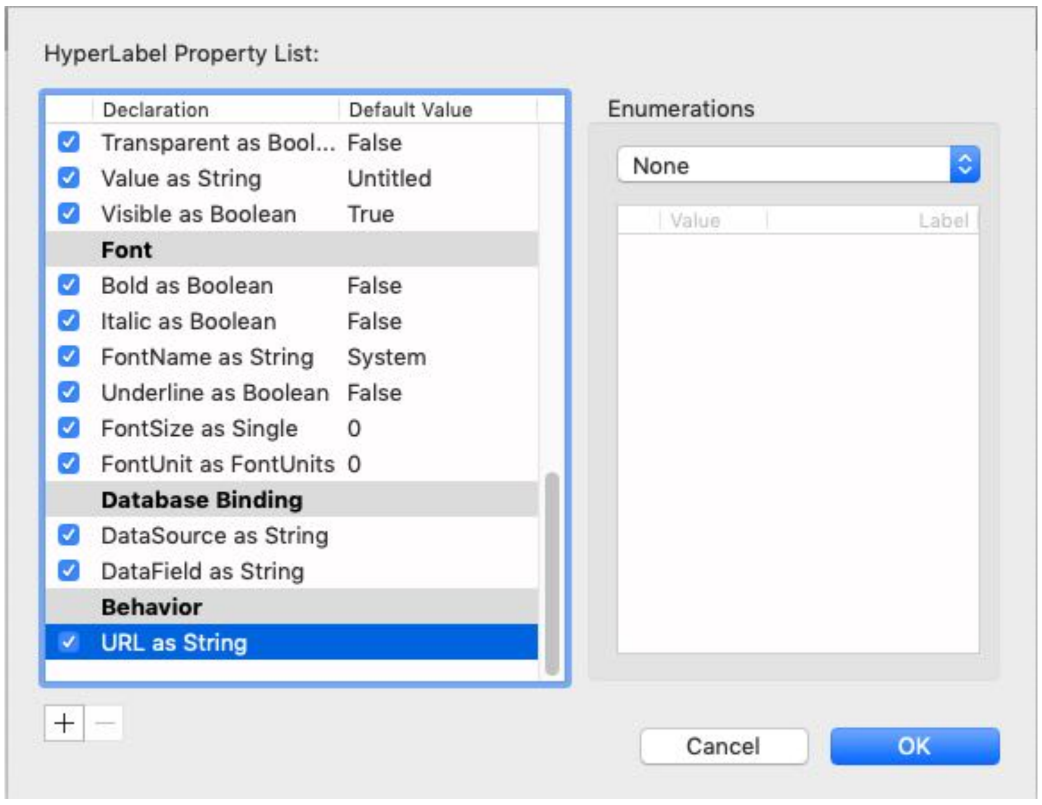
   A subclass with the name CustomLabel will automatically be created. Rename it "HyperLabel", since you'll be using it for hyperlinks.

2) **Add a new property to HyperLabel: URL As String.**

   Your HyperLabel will need a URL, and it would be ideal if you could set that in the Inspector rather than having to do it with code for each instance.

3) **Right click on HyperLabel in the Navigator and choose Inspector Behavior from the pop up menu that appears.**

You will see a list of HyperLabel's properties, including both the property you created and the properties that it inherits from its superclass:



HyperLabel Property List:

	Declaration	Default Value
✓	Transparent as Bool...	False
✓	Value as String	Untitled
✓	Visible as Boolean	True
	Font	
✓	Bold as Boolean	False
✓	Italic as Boolean	False
✓	FontName as String	System
✓	Underline as Boolean	False
✓	FontSize as Single	0
✓	FontUnit as FontUnits	0
	Database Binding	
✓	DataSource as String	
✓	DataField as String	
	Behavior	
✓	URL as String	

Enumerations: None

Make sure the Checkbox next to URL is checked (it will be listed at the end); that will cause the URL property to appear in the Inspector. If you'd like, you can also turn off any properties you won't need to see in the Inspector. Press OK when done.

4) **In HyperLabel's Opening event, add this code:**

```
Self.Underline = True
Self.TextColor = Color.RGB(0, 0, 255)
```

This makes the text blue and underlined, to make HyperLabel appear more like a hyperlink on a web page.

5) **Add this code to the MouseDown event:**

```
Return True
```

6) **And add this code to the MouseUp event:**

```
System.GotoURL(URL)
```

You need to implement its MouseDown and MouseUp events to make HyperLabel respond to clicks, as mentioned above.

7) **Add this code to the MouseEnter event:**

```
Self.TextColor = Color.RGB(0, 0, 150)
MouseCursor = System.Cursors.FingerPointer
```

To make HyperLabel behave more like a hyperlink on a web page, change its color slightly when the user points to it, and change the mouse cursor to the “pointing finger” cursor that you typically see in web pages.

8) Add this code to the MouseExit event:

```
Self.TextColor = Color.RGB(0, 0, 255)
MouseCursor = System.Cursors.StandardPointer
```

This code “resets” its appearance when the user stops pointing to it.

9) Select Window1. Look in the Library to find the new HyperLabel control that you just created and drag it onto Window1. Position it anywhere you like.

10) In the Inspector, set its text to “Xojo” and set its URL property to “https://www.xojo.com”.

11) Run your project.

12) Mouse over the HyperLabel and make sure your cursor changes and the text color changes. Try clicking the text.

13) Quit your application.

As mentioned earlier, now that you have created a custom control by subclassing, you can reuse this control without entering additional code on any Window in your project, or even multiple places on the same Window. In addition, you can add this control to other projects. Right click on HyperLabel in the Navigator and choose Export HyperLabel to save just that control as a file that can be imported into your other projects.

It is a common practice among developers to build your own “library” of custom controls that can be reused in any project where you need it. Your library is off to a good start with HyperLabel.

Chapter 14

Spit & Polish

CONTENTS

14.1 Chapter Overview

So far you've learned a lot about how applications work: variables, data types, logic, flow control, methods, functions, controls, classes, modules, files, databases, and graphics. These are essential things to know about when designing an application.

But another important topic, and one that too many developers ignore, is the user experience. Some people hear this, and they try to think of ways to add some magical "wow" factor to their applications that will grab their users' attention and set their applications apart from the crowd.

In reality, the "wow" factor isn't the important thing. The important thing is providing your users with a consistent and intuitive experience.

Another major component of a good user experience is responsiveness. Your application should never make the user feel like it's frozen. This ties in with keeping the user informed: the user should never wonder what's going on (or how long it will take).

Finally, your application should handle errors (whether generated by the users' actions or your own code) gracefully.

In this chapter, you will learn skills that you can apply to all three of these areas.

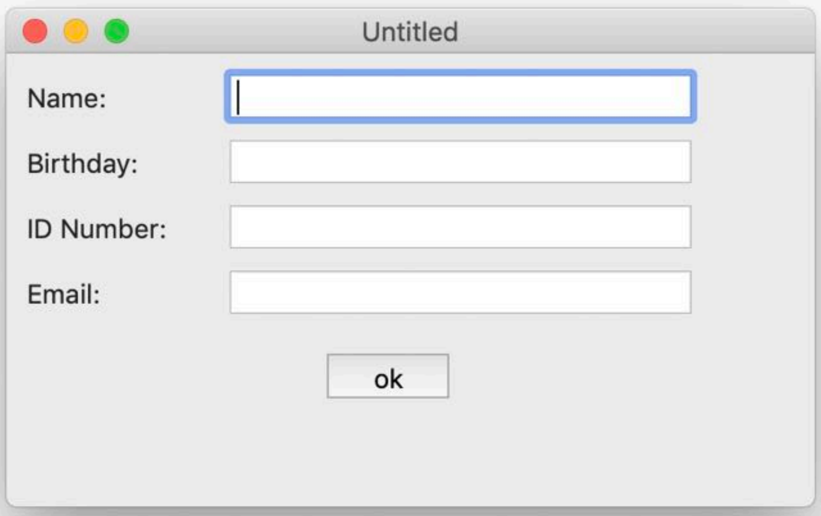
14.2 User Interface Guidelines

Whether you develop for macOS, Windows, Linux, iOS, Android, the web, or almost any other common operating system or platform, there are certain guidelines for the appearance and behavior of your application. These guidelines are typically authored and maintained by the companies that produce the operating systems in question. For example, Google maintains the Android User Interface Guidelines, while Microsoft takes care of the Windows User Experience Interaction Guidelines.

These documents are usually updated whenever a new version of the operating system or platform is released. So each time Apple releases a new version of macOS, the Apple macOS Human Interface Guidelines are updated to reflect the changes in user interface design. With each revision to the operating system, there are often major changes (such as when Apple introduced the Aqua interface or Microsoft added the Universal look to Windows), but even minor changes add up over time.

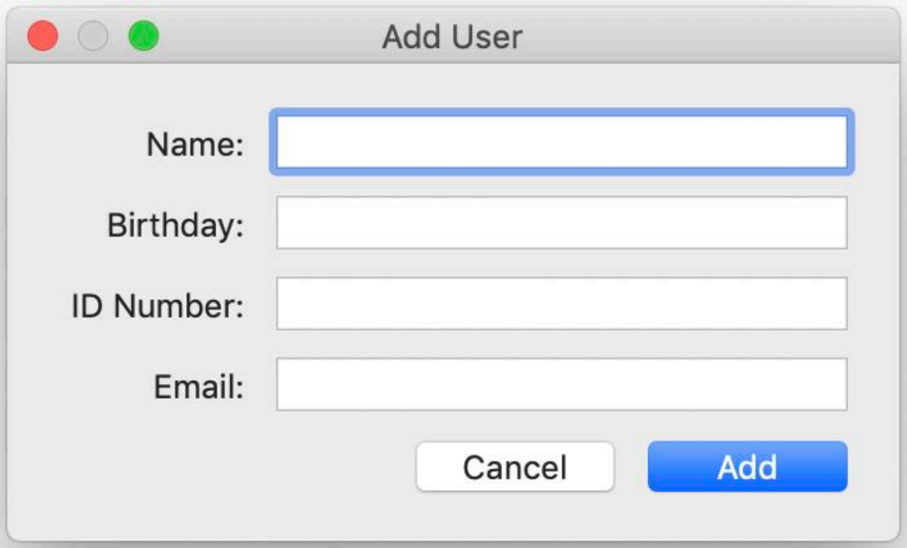
These guidelines can be quite specific, even going so far as to outline how many pixels a certain type of control should be from the top of a window or how many pixels should be between PushButtons. You should follow these guidelines to the best of your ability when it suits the purposes of your application.

Here is an example of an interface designed without consulting any User Interface Guidelines:



Note the inconsistent spacing, incorrect spelling of “OK”, and even a non-standard button being used.

Here is the same interface, tweaked to follow User Interface Guidelines:



As you can see, while both interfaces can accomplish the same task, one will be much more pleasant to use (and is also likely to be more consistent and stable).

Xojo helps you follow the guidelines by helping you position your controls correctly (notice the blue lines that appear when you drag a control near the edge of a window, for example).

However, don't be afraid to go against the guidelines, or "break the rules" when it serves your users better. Remember that the guidelines are only, in fact, guidelines, not hard and fast rules that must be obeyed. In general, though, unless you have a good reason to go against them, stick with what the guidelines say. The advantage in that approach is that your users will automatically be familiar with how certain aspects of your application work, simply by being exposed to similar interfaces in other applications.

A list of interface guidelines for various operating systems and platforms can be found here:

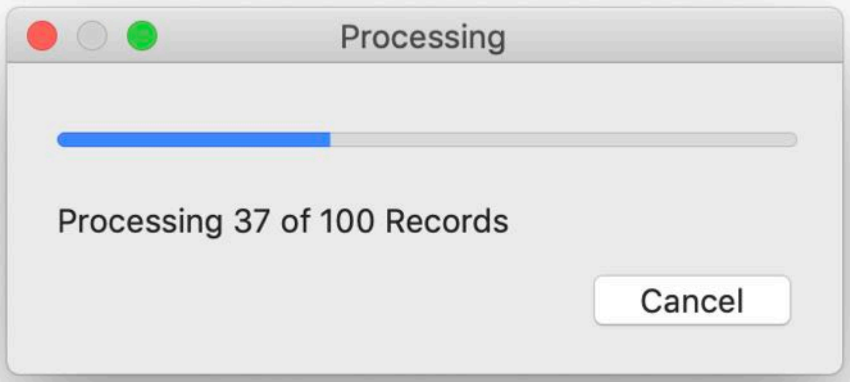
http://en.wikipedia.org/wiki/Human_interface_guidelines

Although they can certainly make for some dry reading material, it's worth keeping a copy of them somewhere you can access it.

14.3 Creating A Responsive Interface

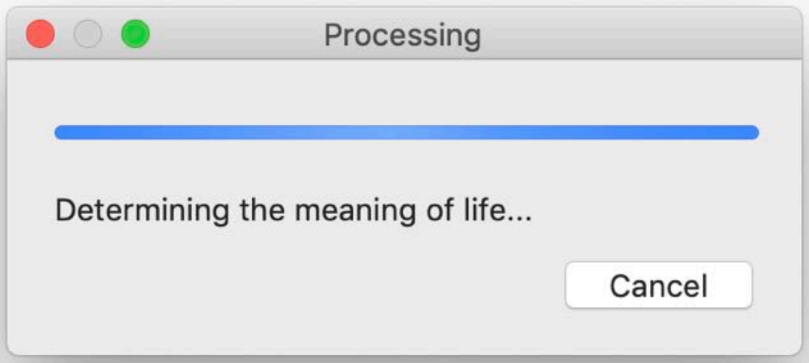
Have you ever been using an application and wondered what it was doing? Maybe you were trying to download a file, print a document, or process an image. Often, an application will provide you with some indicator of both its progress and how much time remains. The applications that don't provide any feedback often give the impression that they are frozen or hung; as a user, this can be very frustrating, as you must decide at which point you stop waiting and force the application to quit.

Think back to the controls you learned about in earlier chapters. Two controls that are excellent for providing feedback to the user are the ProgressWheel and ProgressBar. If you recall from earlier, the ProgressBar is best used when you can quantify what your application is doing. In other words, if you are processing a known number of records or can calculate how long a process will take, the ProgressBar is a great fit. This is because it gives the user an indication not just that something is happening, but how much has happened and how much remains to be done.



Processing 37 of 100 Records

A ProgressBar can also be set to indeterminate which sends a different message to the user. It essentially says, "I don't know how long this will take, but please be patient because I'm still working on it." This mode is better suited to situations where it is difficult or impossible to guess how long something will take, such as using a Socket to connect to a remote server or initiating a database connection.



Determining the meaning of life...

The following project illustrates how to use a ProgressBar to keep your user informed. The project will loop through 10,000,000 numbers and multiply each one by a random number. It's kind of a silly example, but it's a good approximation of the kind of data-intensive processing some applications need to do. You will create this project with and without a ProgressBar, so that you can see the difference.

Create a new Xojo desktop project and save it as "Progress".

1) **Add a Button to Window1.**

2) **Add the following code to the Button's Pressed event:**

```
Var j As Integer
For i As Integer = 0 To 10000000
  j = i * System.Random.InRange(1, 1000)
Next
MessageBox("Done Processing!")
```

This may be your first exposure to the Random class. The Random class, as implied by its name, is used to generate random numbers. In this example, you're using its InRange function, which takes two numbers as parameters and returns a number between those two numbers (in this case, between 1 and 1,000).

3) **Run your project.**

4) **Click the Button and wait for the message box.**

It will take a few seconds, and in the meantime, your application will be unusable and non-responsive. When the process is complete, quit your application.

5) **Add a ProgressBar to Window1. Position it so that it is as wide as possible. Also give Window1 a new property: Progress As Integer.**

6) **Add a Thread and a Timer to Window1.**

Remember the Thread and Timer position themselves in the the Shelf because they are not visual controls. You'll learn more about Timers in the next section.

7) **Change the code in the Button's Pressed event to this:**

```
ProgressBar1.MaximumValue = 10000000
ProgressBar1.Value = 0
Timer1.Period = 500
Timer1.RunMode = Timer.RunModes.Multiple
Thread1.Start
```

This will set the ProgressBar's MaximumValue and current value, set the timer to run every half second, and tell the Thread to start running.

8) Add this code to the Thread's Run event:

```
Var j As Integer
For i As Integer = 0 To 10000000
  j = i * System.Random.InRange(1, 1000)
  Progress = i
Next
```

This is a slightly modified version of the code that was previously in the Button's Pressed event. For each number that the app processes, the Window's Progress property will be increased by one. Because this is being done inside a Thread, the user interface will remain responsive throughout the operation.

9) Add this code to the Timer's Action event:

```
ProgressBar1.Value = Progress
If ProgressBar1.Value >= ProgressBar1.MaximumValue Then
  MessageBox("Done processing!")
  Me.RunMode = Timer.RunModes.Off
End If
```

This will set the ProgressBar's Value to be the value of the Progress property that is set by the running Thread. The Thread cannot directly set the ProgressBar's Value because (due to operating system restrictions) Threads cannot modify (or access) the user interface. The code displays a message and stops the Timer when it has reached the maximum.

10) Run your project.

11) Again, click the Button and wait for the message box.

It still takes a few seconds, but this time, the ProgressBar lets you know that something is happening and that the application isn't stuck, frozen, or crashed.

12) Quit your application.

14.4 Making Things Happen On A Schedule

There will be occasions when you need a certain method or function to run at a certain time or at a certain interval, such as every ten seconds. For situations such as this, use the Timer control.

To see a Timer in action, you are going to create an application that keeps a clock running in order to keep the user informed of the current time.

Create a new desktop project and save it as "Timers".

1) Add a Label to Window1. Set its name to "ClockLabel".

Feel free to position it anywhere you like, but make sure that its Width property is at least 100.

2) Add a new method to Window1 called "UpdateClock".

```
Var today As DateTime = DateTime.Now
ClockLabel.Text = today.SQLDateTime
```

This method has no parameters. Its job is to determine the current time (using a DateTime object) and display it in ClockLabel.

3) Add a Button to Window1. In its Pressed event, call the UpdateClock method:

```
UpdateClock
```

4) Run your project.

5) Click the Button to update the time display.

6) Quit your application.

So far, this is a pretty bad user experience. If the user needs to see the current time, he or she needs to click the Button. It's time to improve this project.

1) Add a Timer to Window1.

Note how the Timer positions itself below the Window and all other controls (the Shelf). This is because the Timer is not a visual control; it has no visual interface for the user to see.

2) **In the Inspector, make sure the Timer's RunMode is set to Multiple and its Period is set to 1000.**

The RunMode can be Off, Single, or Multiple. When the RunMode is set to Off, the Timer is essentially dormant and won't do anything. When the RunMode is set to Single, the Timer's Action event will fire one time, and that's it. When the RunMode is set to Multiple, the Timer's Action event will fire repeatedly, depending on the value of the Period property. The Period is set in milliseconds, or thousandths of a second, so our value of 1000 will cause the Timer's Action event to be called about once a second.

Note that Timers are not precise. The Period indicates the how often you want the Timer to be called but depending on what your application (or the OS) is doing, it may be called less frequently. For example, this Timer with its 1000ms Period may instead get called after 1100ms has passed in some cases.

3) **Add this code to the Timer's Action event:**

```
UpdateClock
```

4) **Run your project.**

Notice that the clock now updates itself about every second, whether the Button is used or not.

5) **Quit your application.**

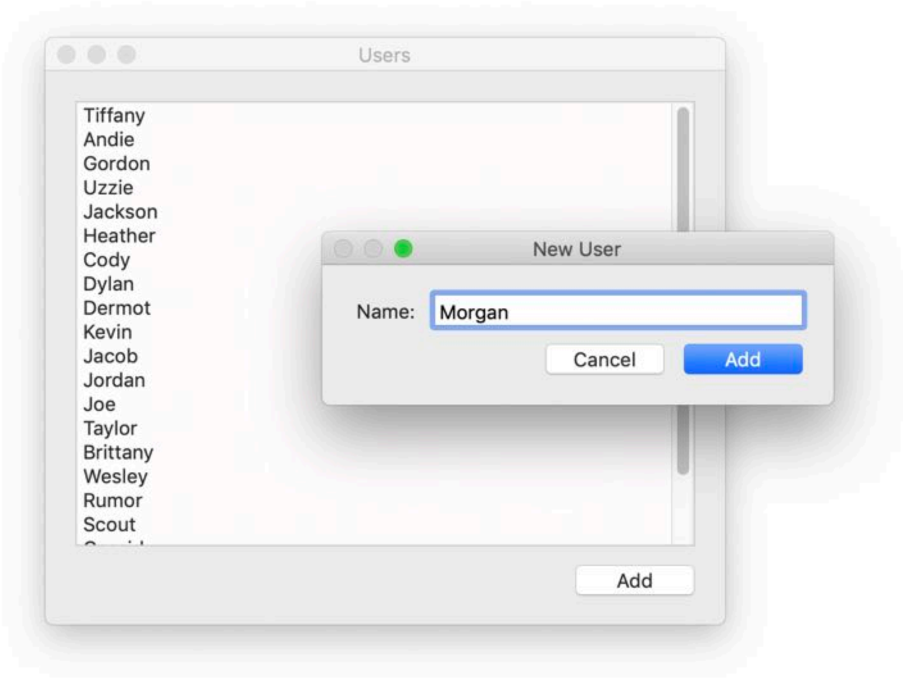
14.5 Managing Windows

So far, all of the projects and applications you have built have one thing in common: they are all single-window applications. This in itself is not necessarily a bad thing, especially since modern user interfaces are trending more and more toward a single-window philosophy (this is particularly true in the case of smartphone and tablet interfaces). However, there will still be times when you need to support multiple windows in an application. Fortunately, each window you add to your project is really a class. This makes managing them very easy, since you already know how to work with classes.

Applications with multiple windows are quite common. For example, any time you use an application with a tool palette, you're using multiple windows. Many email clients have multiple windows as well: one for the list of messages and one for composing a new message. In addition, some applications will allow you to open two windows with different views of the same data!

As you work with various applications, see if you can figure out how many windows each one has defined.

Create a new desktop project and save it as "MultiWindows". This project will allow you to use one window to add names to a list, which is displayed in another window. It may look something like this:



1) **Rename Window1 to “ListWindow”.**

2) **Add another Window to your project by choosing Window from the Insert menu. Name it “DetailWindow”.**

3) **Add a List Box and a Button to ListWindow.**

4) **Set the Button’s Caption to “Add”.**

5) **Add a new property to ListWindow: Names(-1) As String.**

This is an array that will store a list of names.

6) **Add a method called “PopulateNames” (with no parameters) to ListWindow.**

```
Names.Sort
ListBox1.RemoveAllRows
For Each name As String In Names
  Listbox1.AddRow(name)
Next
```

This method’s job is to sort the names in the array and then add each one to the ListBox, using a For loop. Because it will be run multiple times, it will need to remove all existing rows from the ListBox first:

7) **Add a public method to ListWindow called “AddName”. This method takes one parameter: name As String.**

```
Names.Add(Name)
PopulateNames
```

This method’s job is to add a new value to the Names array and then run the PopulateNames method.

8) **In ListWindow’s Opening event, run the PopulateNames method:**

```
PopulateNames
```

9) **Add this code to the Button’s Pressed event:**

```
Var w As New DetailWindow
```

With this code, the Button creates a new instance of DetailWindow which also displays it.

10) **In DetailWindow, add two Buttons (named “OKButton” and “CancelButton”), a Text Field (named “NameField”), and a Label (with a Value of “Name:”).**

11) Set OKButton’s Default property to ON and set CancelButton’s Cancel property to ON.

This will cause OKButton’s Pressed event to be called when the enter key is pressed and CancelButton’s Pressed event to be called when the escape key is pressed (in both cases, the buttons still respond to mouse clicks, of course).

12) Add this code to CancelButton’s Pressed event:

```
Self.Close
```

CancelButton, when pressed, should close its containing Window. This code makes that happen.

13) Add this code to OKButton’s Pressed event:

```
ListWindow.AddName(NameField.Text)
Self.Close
```

OKButton has more work to do. It needs to take the text in NameField and add it to ListWindow’s Names array, then close its containing window. Since it needs to refer back to ListWindow, you might assume that you need to create a variable for it, as you did with DetailWindow above. While that would work, it would also create another instance of ListWindow when the New keyword is used. Windows in Xojo feature something called implicit instantiation, which is a fancy way of saying that if a window is already open, and the application only needs one copy of that window, you can access it by name at any time.

14) Run your project.

15) Click the Add button on ListWindow to bring up the DetailWindow.

From there, you can enter names, which will be sorted and added to the List Box on ListWindow.

16) Quit your application.

You are encouraged to experiment with different kinds of Windows by changing the Type property under Frame in the Inspector. See how this sample project behaves differently with different types of Windows.

14.6 What To Do When Things Go Wrong

Another important aspect of the user experience is error handling. The first step in error handling is to accept that your application will have errors, whether caused by incorrect assumptions or incorrect code. Once you've accepted that errors will occur, you need to learn how to defend your application and your user against these errors.

Many errors in Xojo are actually called exceptions. An exception is just that: it means that something exceptional has happened, something that shouldn't ever happen. An unhandled exception in a Xojo application will almost always cause that application to shut down.

One of the most common exceptions in Xojo applications is the NilObjectException. A NilObjectException occurs when your code tries to access an object that doesn't exist. Take the following code for example:

```
Var d As DateTime
MessageBox(d.ToString(DateTime.FormatStyles.Short))
```

If you run that code, you will see a NilObjectException, because d, the DateTime object, hasn't been instantiated. The variable exists, but it points to an object that doesn't yet exist. This is, of course, easily fixed:

```
Var d As DateTime = DateTime.Now
MessageBox(d.ToString(DateTime.FormatStyles.Short))
```

Other NilObjectExceptions can be trickier to pin down. Imagine you had defined a class called Student, and that this class pulls information from a database. So given a student's ID number, you could query the database and create a new Student object with the appropriate properties filled in, such as first name, last name, and birthdate.

Your code might look something like this (assuming GetStudent is a function that returns a Student object):

```
Var s As Student
s = GetStudent(12345)
MessageBox(s.BirthDate.ToString(DateTime.FormatStyles.Short))
```

This would probably work well as long as you have a valid ID number. What could cause an invalid ID number? There are several possibilities. First, a user could enter an incorrect value when trying to look up a record. Second, it's possible that between the time you start looking for a certain record and the time that it's displayed to you, someone else has deleted the record from the database. Other possibilities exist as well.

With that in mind, your code should look more like this:

```
Var s As Student
s = GetStudent(12345)
If s <> Nil Then
  MessageBox(s.BirthDate.ToString)
End If
```

That If statement, checking to see if s is Nil, can protect you and your users from a lot of pain. Even better would be to provide an error message if s actually is Nil:

```
Var s As Student
s = GetStudent(12345)
If s <> Nil Then
  MessageBox(s.BirthDate.ToString)
Else
  MessageBox("The student could not be found.")
End If
```

You might even include a message to contact tech support or other appropriate people.

The example above, however, only protects you if s is Nil. What if s is a valid, existing Student, but the birthdate hasn't been defined for some reason? In that case, using s.BirthDate.ToString will also result in a NilObjectException. This is an example of something that should be handled in the Student class itself, perhaps by providing a default date or dealing with the NilObjectException there.

```
Try
  Var s As Student
  s = GetStudent(12345)
  MessageBox(s.BirthDate.ToString)
Catch err As NilObjectException
  MessageBox("The student could not be found.")
End Try
```

Everything in the top portion, under Try, is attempted. If a NilObjectException occurs, the code in the lower portion is executed.

Another common exception is the OutOfBoundsException. This occurs when your code has tried to access an element of a list (array, ListBox, etc.) with an index outside of the list. For instance, an

OutOfBoundsException would occur if you try to access the tenth row of a ListBox with only five rows.

Most OutOfBoundsExceptions can be prevented with careful coding. For example, when looping through an array, consider using a For Each loop rather than a loop with a counter.

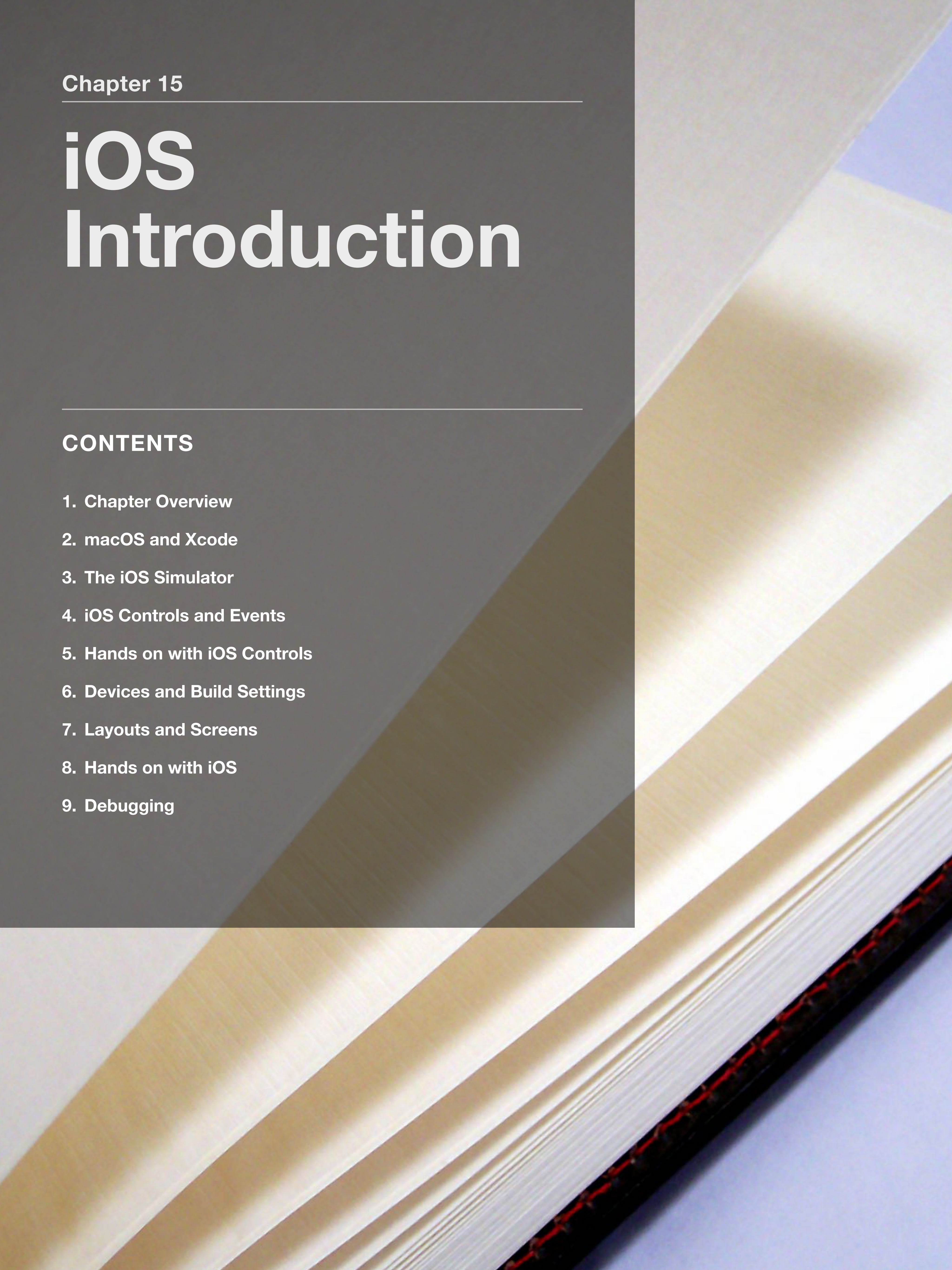
Of course, there are some exceptions that you can’t do much about. One example is the OutOfMemoryException, which occurs when the computer is basically maxed out and can’t spare the memory resources required for the task at hand. In such a case, the best you can do is MessageBox.

Learning to code defensively like this is the best way to protect you and your users from unexpected errors.

Chapter 15

iOS Introduction

CONTENTS



15.1 Chapter Overview

For a long time, there were only two ways to build and deploy apps for iPhone and iPad. You could use Apple's Xcode, which is a powerful tool, but requires you to learn one of Apple's programming languages. Or you could build a web app in HTML and JavaScript, which works, but results in an app that can't be distributed through the App Store and that generally runs much more slowly.

This chapter will provide you with an introduction to creating iPhone and iPad apps with Xojo, which means you can leverage your existing knowledge and skills in Xojo to build and deploy apps for one of today's biggest mobile platforms.

Right off the bat, there are a few issues that should be mentioned. First, Xojo currently only supports iOS, so Xojo apps for Android and other mobile platforms aren't possible at the moment. Second, because of the way that Apple's Developer Tools work, Xojo iOS apps can only be developed on the Mac. You'll learn more about that in a bit.

This is meant to be an introduction to iOS development with Xojo. You'll learn how to create an iOS app and test it in Apple's iOS Simulator. Additional topics like distributing your app on the App Store are beyond the scope of this introduction, but there are some links at the end to help you with those advanced topics.

This introduction assumes that you have worked through Introduction to Programming with Xojo and are familiar with the basics of using Xojo.

15.2 macOS and Xcode

As mentioned earlier, developing for iOS requires a Mac. You should be running the latest version of macOS. You will also need to install Apple's Xcode, which is a free download from the Mac App Store.

Apple's Developer Program does have paid memberships, but you do not need to pay to test your iOS apps. If you decide to move forward and deploy an app on the App Store, you'll need to investigate a paid membership at that point.

Xcode is Apple's recommended development environment. It's a very powerful tool, but it has a much steeper learning curve than Xojo. You will mostly only need Xcode because it supplies you with the iOS Simulator. You can download Xcode for free from the App Store.

Note that the first time you launch Xcode (whether you launch it manually or Xojo launches it for you), you'll need to agree to Apple's terms and also allow it to install any additional tools and system components.

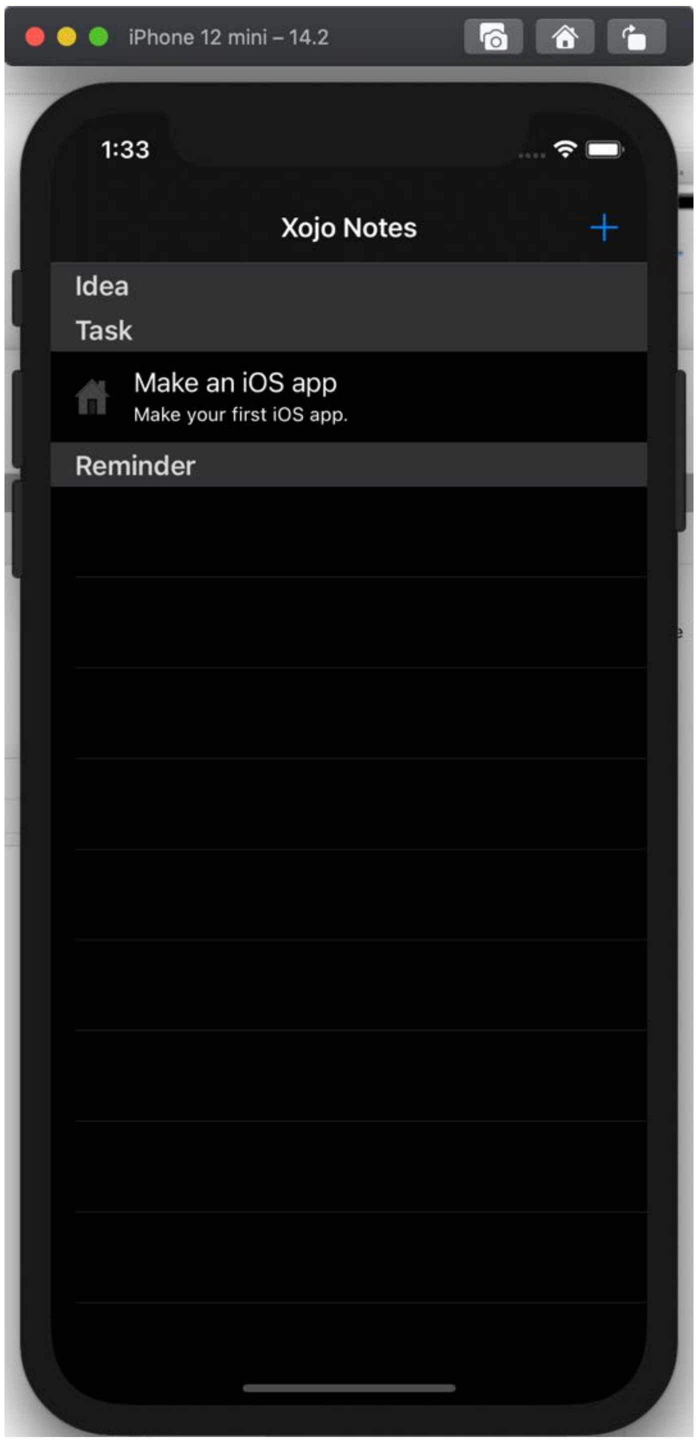
Other Xcode install tips are located here: https://documentation.xojo.com/topics/application_deployment/apple_requirements/installing_xcode_and_apple_certificates.html

15.3 The iOS Simulator

The iOS Simulator is a Mac app that runs a stripped down version of iOS, which you can use to test your apps. If you think back to working with Xojo desktop apps, you could click the Run button and launch a temporary version of your app. For iOS apps, that temporary version runs inside the iOS Simulator.

In the early days of iOS, a developer could rely on every iPhone having the same screen size and same resolution. That started to change in 2010 with the advent of retina displays, and in recent years, Apple has released an even wider variety of screen sizes and resolutions.

Because of this variety, you'll want to test your app on different devices. The iOS Simulator allows you to set up multiple virtual iOS devices (both iPhones and iPads of various sizes), which you'll learn about in the next section.



15.4 iOS Controls and Events

Before you jump into a quick and easy sample project, you need to know a few things about controls on iOS. They differ from desktop (and web) controls in a few significant ways.

First, because there is no mouse pointer on iOS, controls have fewer events. There is no MouseEnter event, or anything mouse-related. Most controls have events for Opening, Closing, and some have Pressed.

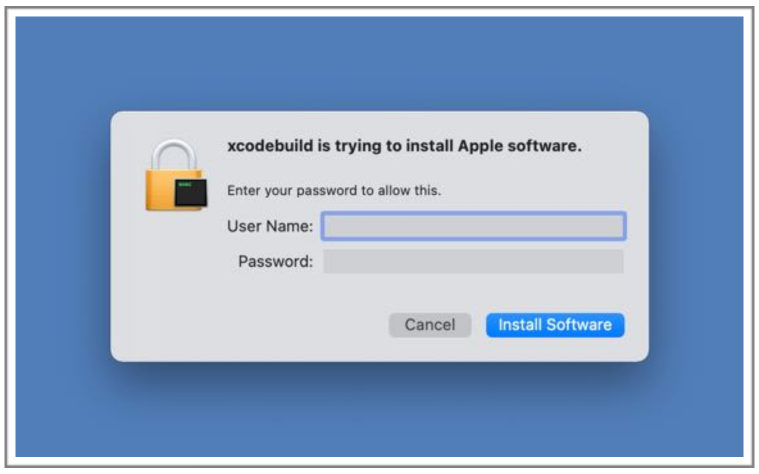
Second, while Xojo on the desktop keeps track of your mouse pointer's coordinates for you, iOS doesn't have that capability unless the user's finger is touching the screen.

Last, you'll notice that the list of available controls for iOS is a bit smaller than for the desktop and web. Many of those controls are not relevant on touch screen devices.

15.5 Hands on with iOS Controls

o create an iOS app, begin by opening the Xojo application (download from https://www.xojo.com/download). You will be prompted to choose a template for a new project. Select “iOS” and fill in the Application Name, Company Name, and Application Identifier.

The Application Name and the Company Name can be whatever you like. The Application Identifier, however, should be in Apple’s “reverse DNS” format. For example, if a company called FooBar created an app called DreamCatcher, the Application Identifier might be com.foobar.dreamcatcher (as you can see, it almost looks like a backwards website address, which is why it’s sometimes called reverse DNS format).



xcodebuild is trying to install Apple software.

Enter your password to allow this.

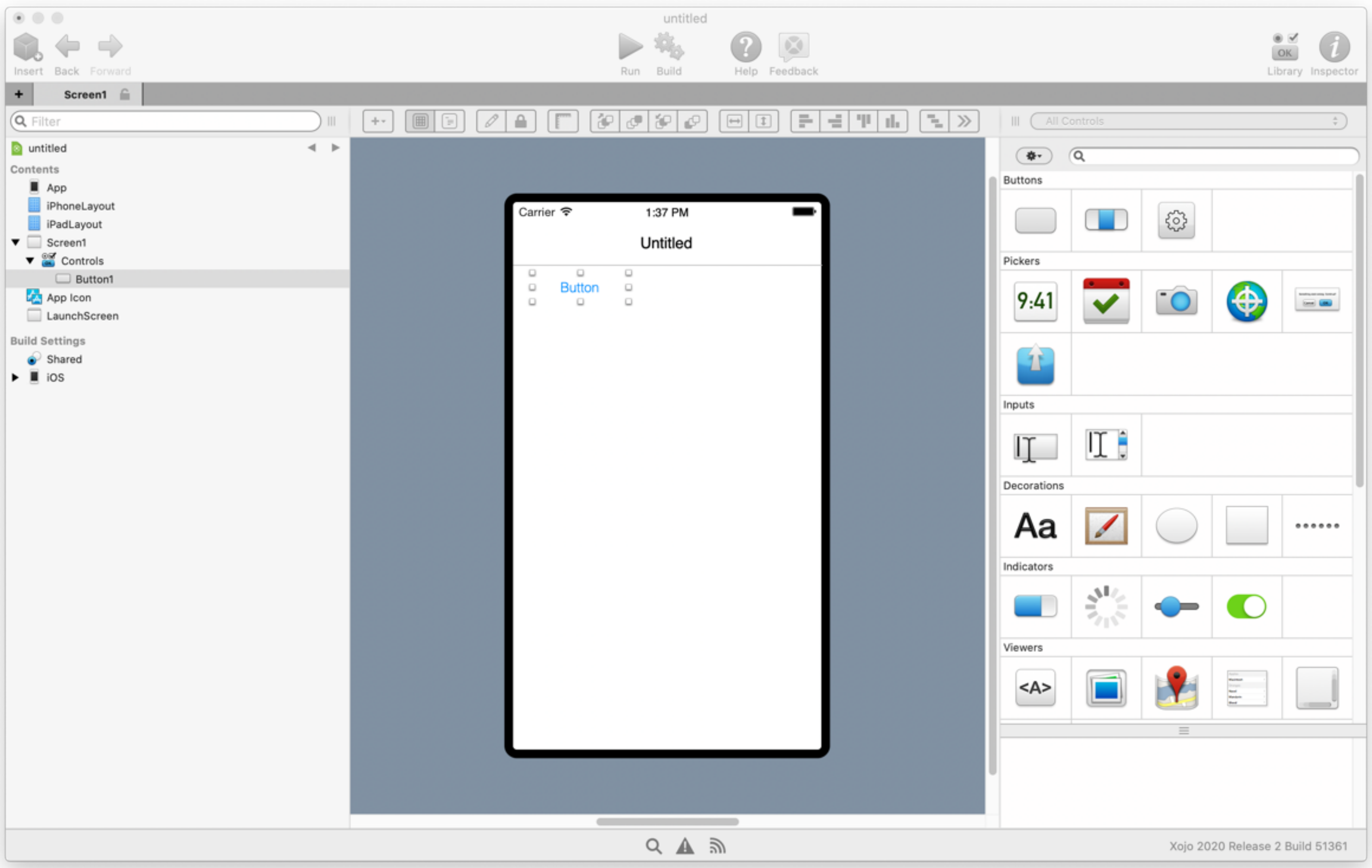
User Name:

Password:

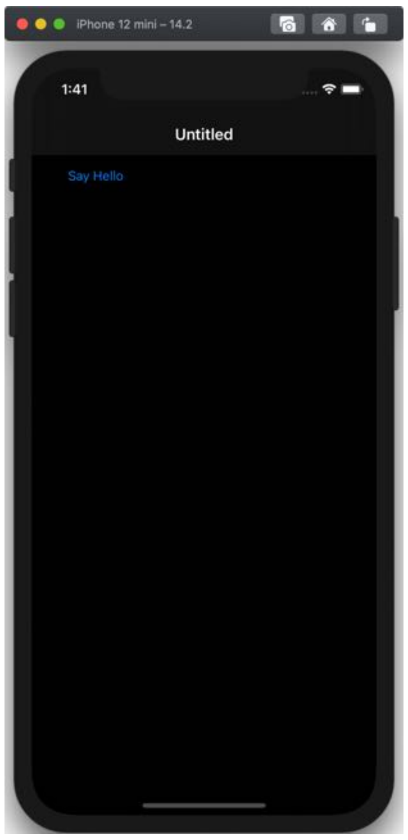
Once you’ve filled in all three fields, press the OK button. You’ll be greeted with a screen that looks largely familiar, but different in some subtle ways.

Note that if Xcode has not been installed or properly configured, you may get a dialog that prompts you to install additional Apple development components.

Now the Xojo workspace appears, where you’ll see an iPhone screen (instead of the usual desktop window or web page). You’ll also notice that the list of available controls is not as lengthy and varied as the desktop selection, as mentioned above.



The steps below walk you through creating a simple iOS "Hello, World" app.



1) Find the Button in the controls list and drag it onto Screen1.

Use the guides to center the Button on the Screen.

2) Find the MessageBox in the controls list and drag it onto Screen1.

The MessageBox control will automatically place itself at the bottom of the Layout Editor (this area is called the Shelf), because it is a non-visual control that only appears when needed.

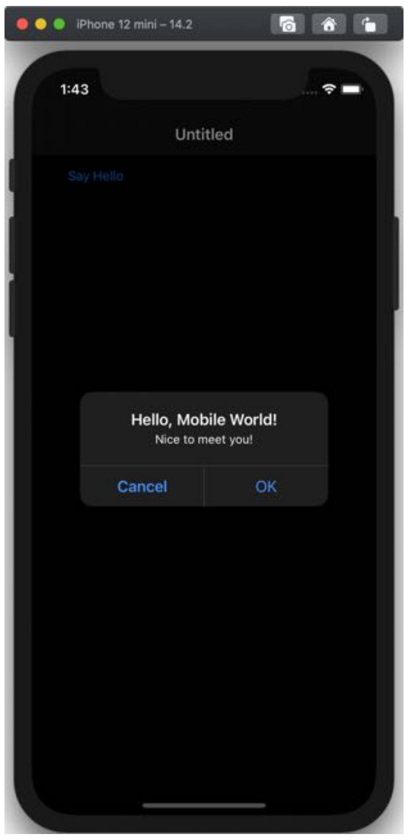
3) Using the Inspector, change your Button's caption to "Say Hello."

4) Using the Inspector, change the following properties of your MessageBox:

Message: "Nice to meet you!"
Title: "Hello, Mobile World!"
Left Button: "Cancel"
Right Button: "OK"

5) Add the following code to your Button's Pressed event:

```
MessageBox1.Show
```

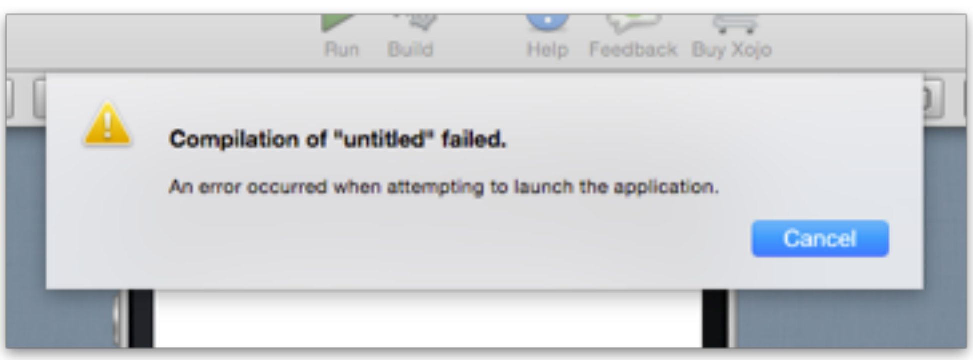


6) Run your project by choosing Project->Run On from the menu and selecting an installed Simulator.

If your computer is set up correctly, the iOS Simulator will appear and launch your app. Click on the button to see the "Hello, Mobile World" MessageBox appear. It should look something like the images below.

7) Troubleshoot

If your computer is not set up correctly, you may see an error message, such as the following message indicates that either Xcode is has never been run (which also means that the license hasn't been agreed to).



Compilation of "untitled" failed.

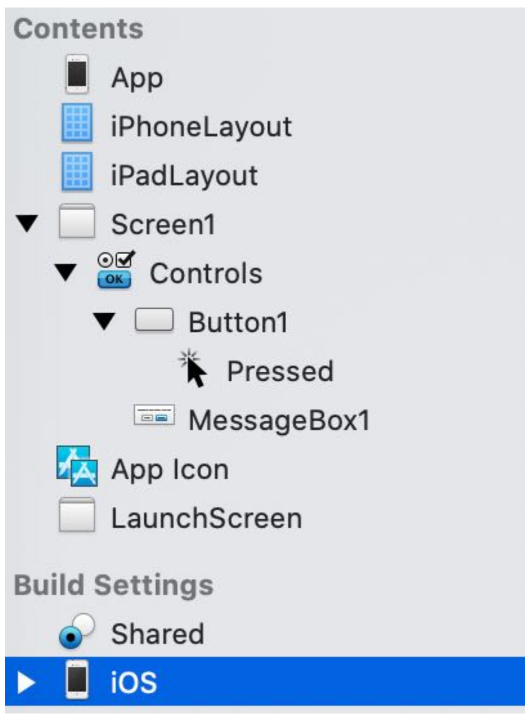
An error occurred when attempting to launch the application.

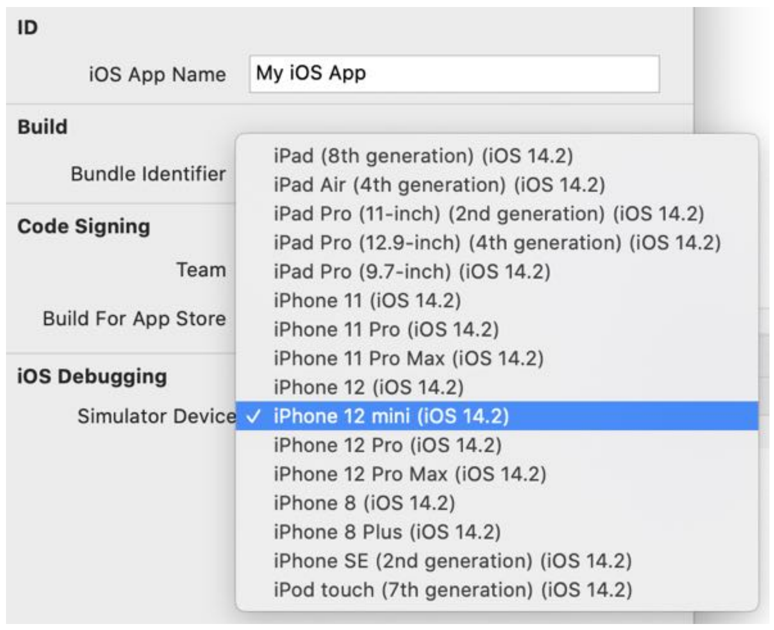
15.6 Devices and Build Settings

As mentioned earlier, the iOS Simulator can be used to test your app on a variety of virtual devices. By default, Xcode installs several devices for you to use.

If you want to add more devices or devices with different versions of iOS, you can do so using Xcode's Preferences window.

Back in Xojo, you can choose which device to use for debugging by going to the iOS Build Settings in the Navigator:





ID

iOS App Name: My iOS App

Build

Bundle Identifier

Code Signing

Team

Build For App Store

iOS Debugging

Simulator Device:

- iPad (8th generation) (iOS 14.2)
- iPad Air (4th generation) (iOS 14.2)
- iPad Pro (11-inch) (2nd generation) (iOS 14.2)
- iPad Pro (12.9-inch) (4th generation) (iOS 14.2)
- iPad Pro (9.7-inch) (iOS 14.2)
- iPhone 11 (iOS 14.2)
- iPhone 11 Pro (iOS 14.2)
- iPhone 11 Pro Max (iOS 14.2)
- iPhone 12 (iOS 14.2)
- ✓ iPhone 12 mini (iOS 14.2)
- iPhone 12 Pro (iOS 14.2)
- iPhone 12 Pro Max (iOS 14.2)
- iPhone 8 (iOS 14.2)
- iPhone 8 Plus (iOS 14.2)
- iPhone SE (2nd generation) (iOS 14.2)
- iPod touch (7th generation) (iOS 14.2)

Whichever Simulator Device you choose here will be launched when you run your iOS app in the Debugger, and your app will be scaled to that device accordingly.

15.7 Layouts and Screens

One of the biggest ways that iOS development is different from desktop development is in how information is displayed to the user. On the desktop, your user sees a window, which is just a portion of the screen. Web projects use web pages.

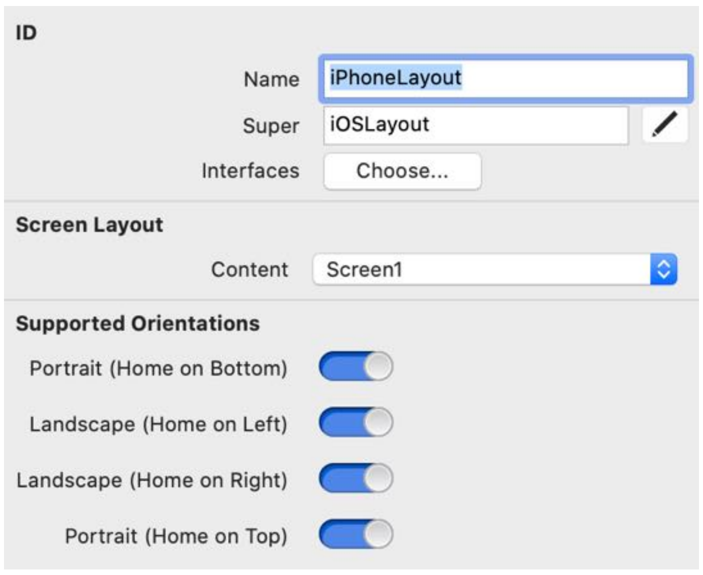
On iOS, there are no windows or web pages. The user is unable to resize the interface, and his or her entire screen is always active (on phones, at least). Since iOS can run on a variety of devices and screen sizes, your app needs to be able to adapt to different layouts and resolutions. This is accomplished using Layouts and Screens.

It’s helpful to think of a Screen as the iOS version of a window or web page. They're not exactly the same, but a Screen is where you design your interface and define how your app responds to the user, much like a desktop Window.

A Layout is a bit trickier, because there’s no real desktop or web feature that compares to it. In your iOS app, you define a Layout by device and orientation. An iOS app in Xojo will have two Layouts by default: an iPhone Layout and iPad Layout.

If you’re building an iPhone-only app, it’s safe to delete the iPad Layout, and vice-versa.

Recall that in a desktop app, you select the default window seen by the user. In an iOS app, you select the default Screen for the user, but you must do so for each Layout:



ID

Name: iPhoneLayout
Super: iOSLayout
Interfaces: Choose...

Screen Layout

Content: Screen1

Supported Orientations

Portrait (Home on Bottom): On
Landscape (Home on Left): On
Landscape (Home on Right): On
Portrait (Home on Top): On

Notice that by combining your Layout and Screen with the different orientations available, you can create a very customized interface for your iOS app. For example, someone using your app on an iPad in landscape orientation could see an entirely different interface from someone using your app on an iPad in portrait orientation. And you can even further refine the interface based on whether the user has the iOS device’s Home button at the top, at the bottom, on the right, or on the left.

But switching from one Screen to another on iOS is not as simple as showing a new Window on the desktop. Because the current Screen will fill up the user interface, you must manage which Screen is in front. This is best demonstrated with a quick sample project.

1) **Create a new iOS project.**

   Since this is a quick demonstration project, you do not need to worry about the Application name and other related details. A Screen called Screen1 will already exist by default.

2) **Create a new Screen.**

   Go to the Insert menu and choose Screen. The new Screen will be called Screen2 by default.

3) **Add an Button to Screen1.**

   Drag a Button onto Screen1 and position it in the center of the Screen. Change its Caption to “Show Screen2.” Give the Button a Pressed event handler with the following code:

```
Var s2 As New Screen2
s2.Show
```

4) **Set Screen1’s BackButtonCaption property.**

   Every Screen has a property called “BackButtonCaption.” This property determines the caption of the button that will return the user to the Screen. Set Screen’s BackButtonCaption to “Back.” This will be displayed on the system-level toolbar for Screen2 as a Back button. Remember that the Back button’s caption on Screen2 is set by Screen1’s BackButtonCaption property.

5) **Add a Label to Screen2.**

   Drag a Label onto Screen2 and position it in the center of the Screen. Change its Text property to “This is Screen2.”

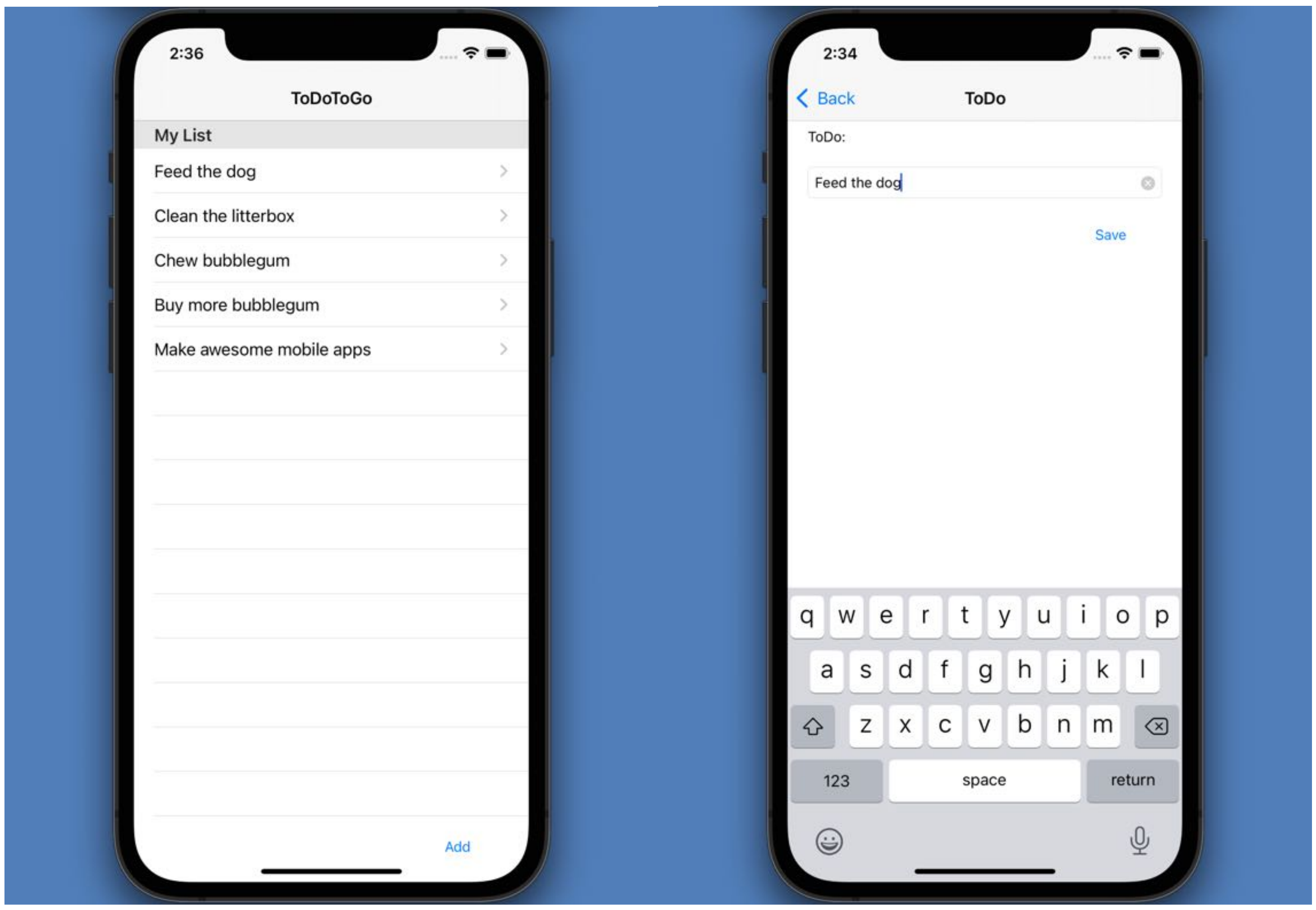
6) **Run your project.**

   Experiment with navigating back and forth between Screen1 and Screen2. Notice in particular how Screen2 “slides” on top of Screen1 and then “slides” back out of it way.

Managing these interactions between Screens is a critical part of developing a great iOS app because they give user a sense of “place” within the interface. The Show method in particular, causes a Screen to “slide in” or “on top of” another Screen, creating a sense for the user that these Screens are stacked and that their positions are relative to one another.

15.8 Hands on with iOS

Your sample project for iOS is a variation on the To Do List Manager that you created in Chapter 5. You will be able to add items to a to do list, edit those items, and delete them. The finished app will something like this:



1) **If you haven't already done so, launch Xojo and create a new iOS Application. Save it as "ToDoToGo".**

   To start, you'll build the interface, then add your code later. First, rename your default Screen1 to "ListScreen". Make sure ListScreen's BackButtonCaption is set to "Back" and that its HasNavigationBar property is set to True. Set ListScreen's title to "ToDoToGo." From the Library, add two controls to ListScreen: a Table named "ToDoTable" and a Button named "AddButton."

2) **Add a new module to your project by going to the Insert menu and choosing Module.**

   The module should be named "ToDoList." Give it a new property: ToDo() As String. The parentheses after the property name make this property an empty array. This is where your app will store the user's to do items. Make sure the property's Scope is set to Global.

3) **Add a new method to ListScreen.**

   Your app will need a way to display the current list of items to the user, so you will need to add a method that loops through the array of to do items and places each one in the iOSTable. Add a new method to ListScreen called "PopulateTable."

4) Add this code to the PopulateTable method:

```
Var cell As MobileTableCellData
ToDoTable.RemoveAllRows
ToDoTable.AddSection("My List")

For i As Integer = 0 To ToDo.LastIndex
  cell = ToDoTable.CreateCell
  cell.Text = ToDo(i)
  cell.AccessoryType = ¬
  MobileTableCellData.AccessoryTypes.Disclosure
  ToDoTable.AddRow(0, cell)
Next
```

Note that working with an iOSMobileTable is not quite the same as working with a ListBox in a desktop app. One major difference is that a Table on iOS has sections, which are discrete areas of the table, each with its own title. For example, if you wanted to break your to do items into categories, you could have a section of the Table for each category. To keep things simple in this example project, the above code adds one section called “My List” to the Table.

Adding cell data to a Table is also quite different. Remember that on the desktop, you only need to use the AddRow method and feed the ListBox a string value. On iOS, you will create a new instance of the class MobileTableCellData (by calling CreateCell) and pass that to the iOSMobileTable’s AddRow method. That method also takes an integer indicating which section the cell should belong to. In this example, since there is only one section, just use zero as the first parameter.

The MobileTableCellData data type has one property that should always have a value: Text. This is the data that will be displayed on the list that the user sees. The above method also assigns an AccessoryType to the MobileTableCellData. The AccessoryType provides a graphical cue to the cell’s purpose. In this case, the Disclosure type adds a small arrow at the end of the cell indicating that the user can touch that cell to trigger a new interaction.

Finally, note that before you fill the Table with data, you must clear out any existing data, to avoid presenting the user with multiple copies of the same data and potentially causing confusion.

5) Give ListScreen an Event Handler for the Activated event.

Whenever the user sees ListScreen, the data should be refreshed and up to date, so the Activated event handler should include this line of code:

```
PopulateTable
```

6) Add a new Screen called “ToDoScreen.”

This is the Screen where the user will enter and edit to do items. Make sure its HasNavigationBar property is set to True and its Title property is set to “ToDo”. It will need four controls, which you may position as you see fit (again, using the screenshot above as a guide): a Label, a TextField named “ToDoField,” a Button named “SaveButton,” and a Button named “DeleteButton.” SaveButton and DeleteButton’s captions should be “Save” and “Delete,” respectively. Set DeleteButton's Visible property to False; it will only be needed when the user is editing a to do item, so your app should only show it when needed. Finally, give ToDoScreen a new property: ToDoItem As String.

7) Add a new method to ToDoScreen called "Populate."

The method should take one parameter: item As String. The code below will take that value and assign to the Screen's ToDoItem property.

When the user edits a to do item, your app will need to show the text of that item in ToDoField. The Populate method will take care of that. Here is the code for the method:

```
ToDoItem = Item
ToDoField.Text = ToDoItem
DeleteButton.Visible = True
```

This code sets ToDoScreen's ToDoItem property to the text of the item that the user wishes to edit. It also places that value into ToDoField's text property. Finally, it makes DeleteButton visible to allow the user to remove the to do item from his or her list.

8) Add the following code to the Pressed event of AddButton on ListScreen.

Give AddButton on ListScreen a handler for its Pressed event and add this code:

```
Var todo As New ToDoScreen
todo.Show
```

This code should look familiar from the navigation example earlier in this chapter. It displays ToDoScreen to allow the user to create a new to do item.

9) Add the following code to the Pressed event of ToDoTable on ListScreen.

Give ToDoTable on ListScreen a handler for its SelectionChanged event and add this code:

```
Var item As String
Var todo As New ToDoScreen
item = Me.RowCellData(section, row).Text
todo.Populate(item)
todo.Show
```

This code is called when the user taps a row on ToDoTable. It determines which to do item the user has tapped and passes that to the Populate method of ToDoScreen. Note that rather than simply accessing the string value of the selected row as you would on the desktop, you must access it via the RowCellData method.

10) Add the following code to the Pressed event of SaveButton on ToDoScreen.

Give SaveButton on ToDoScreen a handler for its Pressed event and add this code:

```
If ToDoField.Text <> "" Then
  If ToDoItem <> "" Then
    For i As Integer = 0 To ToDo.LastIndex
      If ToDo(i) = ToDoItem Then
        ToDo(i) = ToDoField.Text
      End If
    Next
  Else
    ToDo.Add(ToDoField.Text)
  End If
End If
Self.Close
```

When the user taps SaveButton, your app will need to either create a new to do item or update an existing one. To determine which action to take, this code checks to see if ToDoScreen’s ToDoItem property contains any text. If it does not, it is safe to assume that the user intends to create a new to do item. This is done by adding the text property of ToDoField to the global ToDo array.

If, however, ToDoScreen's TodoItem does contain text, then the existing item must be updated. To do so, this code loops through the global array of ToDo items, looking for one that matches the one that was given to ToDoScreen via the Populate method. If it finds a match, it updates that item in the array.

When either task is complete, the method closes ToDoScreen and returns the user to ListScreen.

11) Add the following code to the Pressed event of DeleteButton on ToDoScreen.

Give DeleteButton on ToDoScreen a handler for its Pressed event and add this code:

```
If ToDoItem <> "" Then
  For i As Integer = 0 To ToDo.LastIndex
    If ToDo(i) = ToDoItem Then
      ToDo.RemoveAt(i)
    End If
  Next
End If
Self.Close
```

When the user taps DeleteButton, the app will loop through the global ToDo array, searching for an item that matches the one that was given to ToDoScreen via the Populate method. If a match is located, it is removed from the array, ToDoScreen is closed and the user is returned to ListScreen.

12) Run your project.

The iOS Simulator should appear and your iOS app should launch. Try adding, editing, and deleting some to do items.

If your project fails to run, make sure that you have selected a Simulator Device in your project's iOS Build Settings and that Xcode is properly installed.

15.9 Debugging

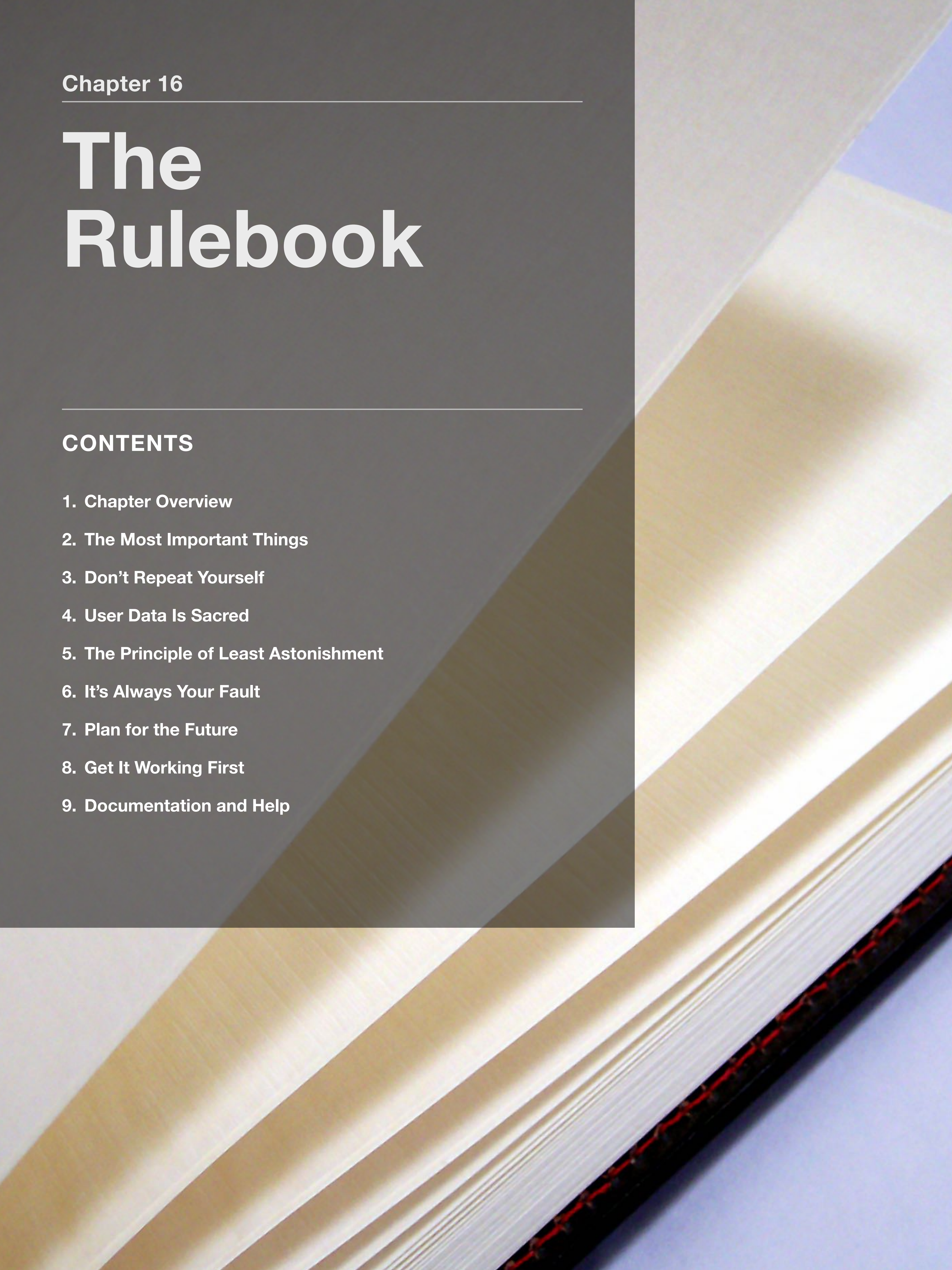
Even though your iOS project seems like it’s bouncing through various applications before you see it running, debugging it is almost exactly the same as debugging a desktop app. You can still set breakpoints and step through code.

For more information on debugging, see previous sections 1.5 and 2.6.

Chapter 16

The Rulebook

CONTENTS



16.1 Chapter Overview

If you’ve completed the other chapters in this textbook, you have a good start on the basics of computer programming, especially using Xojo.

However, whether you choose to continue using Xojo or try something like Swift, PHP, Objective-C, JavaScript, Ruby, Java, C/C++, C#, or any other language, you still have much to learn about the rules and principles of computer programming. While some of these “rules” are specific to computer programming and software development, many of them are also directly related to good project management and communication skills.

This chapter will be different from the rest. Rather than providing you with instructions for a sample project, this chapter will pose some questions that are intended to be discussed.

16.2 The Most Important Things

Many computer developers get caught in what can be called the Programmers' Trap. After spending months or years learning to write good code, and then spending more months or years writing applications, many developers fall into the trap of thinking that their job is writing code, but it is not.

Even if your job title is Computer Programmer, Software Developer, or Chief Software Architect, your primary job is not to write code. That's simply an aspect of your job. You have two main jobs. The first is solving problems and the second is adding value. You will, of course, often write code to help you solve problems and add value. In this field, that's to be expected.

The most difficult part of solving problems is rarely coming up with a workable solution. More often than not, the hardest part is defining the problem in the first place. Quite often, the users or clients who will be asking you for an app will also have a hard time defining the problem. But that's why they came to you.

The best way to define the problem is to ask questions. When you get answers, keep asking questions. Then repeat the questions. Keep going until you thoroughly understand what the client or user wants. Then write it down and develop a flowchart of what the client or user has described and go over it with them. Before you write a single line of code, make sure you understand the problem that you're trying to solve.

Just as important as the questions you ask are the people you talk to. It's always a good idea to talk to whoever is signing the check, so to speak, but it's also critical to get "into the trenches" and talk with the people who will be using your application, perhaps on a daily basis.

But be aware that your end users won't always be able to specify what they want. After all, as Henry Ford pointed out, if he had asked people what they wanted, they would have asked for a faster horse. Before Apple released the Macintosh and Microsoft released Windows, people wanted bigger, faster DOS machines.

Very often, the problem that needs to be solved will not be the one specified by the client or user. In such cases, you will need to dig deeper. A classic example tells of a man who said he needed a hammer. Digging deeper into his story, it turns out that while he did need a hammer, that wasn't his real problem. He actually needed to drive a nail into a wall. But that wasn't his real need either. He actually needed the nail in the wall so he could hang a picture. And he wanted to hang the picture because he wanted to beautify his surroundings. When asked, the man said he needed a hammer, but what he really wanted was art and beauty around him. In the same way, you need to ask questions and dig deeper.

This leads into the developer's second main job: adding value. Your work and your code should add value to the user's experience. This can take many forms, such as providing art as mentioned above. It might be taking a paper process and moving it online to make it faster and more efficient. Whatever the task is, your job is to make your users' lives better, even if in very small ways.

FOR DISCUSSION:

1) **When trying to determine the problem that needs to be solved, what kinds of questions would you ask the user or client?**

2) **How would you handle someone who came to you with a detailed solution already mapped out without specifying the actual problem?**

3) **In what other ways can you add value?**

16.3 Don't Repeat Yourself

As mentioned earlier in this book, one of the cardinal rules of programming is Don't Repeat Yourself, sometimes referred to as the DRY principle. The DRY principle means that if you have the same code in more than one place in your application, you're doing it wrong.

Whenever you have duplicate code, you are inviting problems, and bugs, into your application. Inevitably, something will change down the road and you will need to change your code. Will you remember to change it in all of the relevant places? No matter how smart or diligent you are, history suggests that you will eventually forget to make an important change, and your code will end up out of sync.

The solution to not repeating yourself is to abstract as much code as possible into methods and functions. As a general rule of thumb, if a portion of code is more than two or three lines and you are using that portion of code in more than two places (this is known as the Rule of Three), you would be well served by moving that code into a method or function that you can call as needed.

Also remember that if your code isn't DRY, it's WET. WET stands for Write Everything Twice.

FOR DISCUSSION:

1) **Why is the DRY principle more important than other programming principles?**

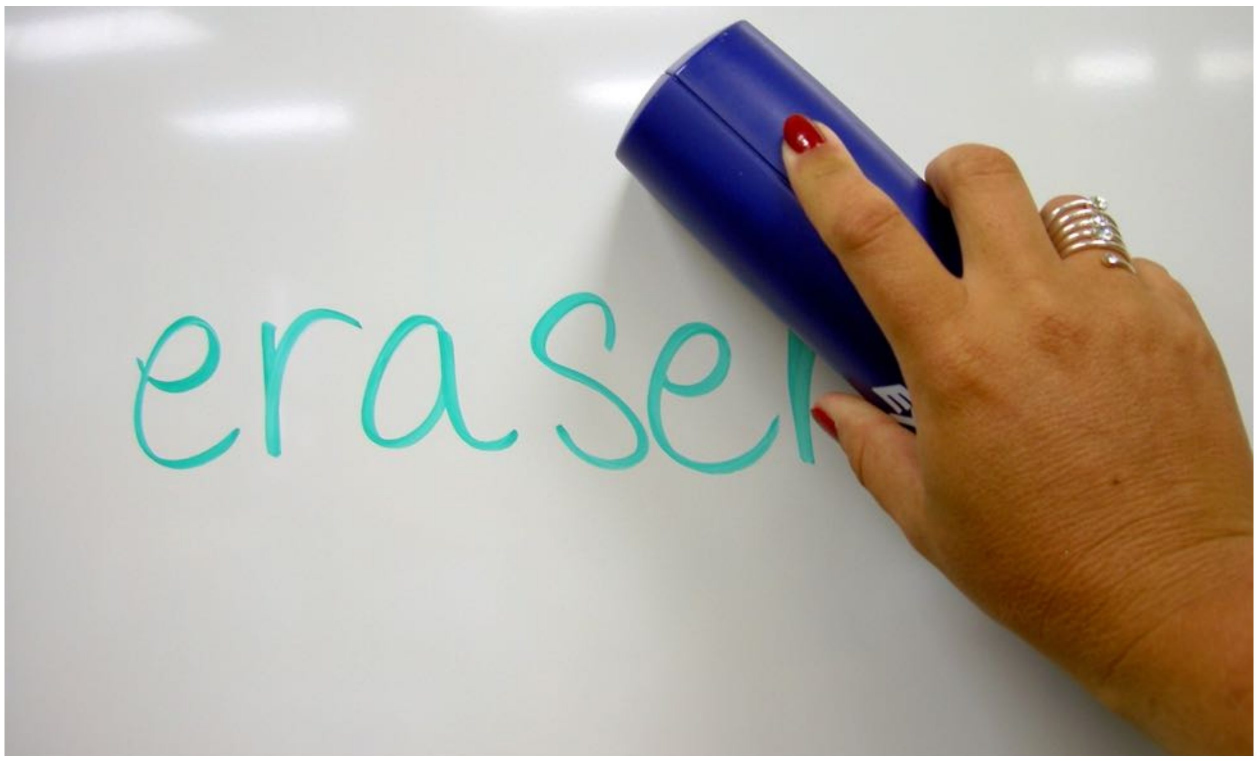
2) **What are the disadvantages to a WET approach?**

16.4 User Data Is Sacred

You likely use an online email service like Gmail or Yahoo Mail. Imagine logging into your account one day to discover that all of your archived messages were gone. How would you react?

Or imagine that you logged into Facebook only to find that all of your photos had been deleted.

Odds are good that you would be pretty upset in either case. Your users would be upset, too.



Another rule of computer programming is this: User Data Is Sacred. Never delete your user's data without three things: a very good reason, your user's permission, and a chance for your user to cancel.

It's important to note that this applies only to user data. If your code generates some temporary files that no longer store important data, those are always safe to delete. In fact, as in the rest of life, it's a good idea to clean up after yourself. But data that the user has created or saved must be kept safe.

As stated above, before you delete user data, you need a very good reason. The most obvious reason is the user deciding to delete something. Another reason might be that the data has expired. There may be others as well. An example of a bad reason to delete user data is a programming error. If your application deletes data for no good reason, you can rest assured that very few people will use it!

You also need the user's permission. If the user has initiated the deletion process, then you are probably clear on this one. If your application started the process, then you need to make sure, through a dialog box or other mechanism, that your user is granting you permission to continue.

Finally, you need to provide an opportunity for the user to cancel the deletion. This usually takes the form of an "Are you sure?" dialog box. Granted, these can become cumbersome for a user, but that inconvenience is a good trade off for more secure data.

As a bonus, you might provide a method for the user to undo deleting the data. This is not required, although it is very common. If the deletion cannot be undone, it's a good idea to warn your user of that fact when confirming the deletion.

FOR DISCUSSION:

1) What are some other valid reasons for deleting data?

2) How do you think you might start implementing an undo feature?

16.5 The Principle Of Least Astonishment

When you click on a PushButton labelled "Print", what do you expect to happen? If you're like most people, you'd probably expect something to come out of your printer (likely something related to what's on the screen). What if you clicked a Print button and the app quit? Or what if you clicked on a checkbox and the application sent an email?

Users have come to expect certain interface elements to do certain things (see the previous chapter for more examples). That's part of the Principle of Least Astonishment: interface elements should do what they normally do. That means a radio button shouldn't close a window, and a PushButton shouldn't pop up a menu.

Now, this is not to say that your application should never surprise your users. In fact, sometimes surprises are excellent. But, a surprise should always leave the user delighted rather than astonished. Always, always strive to delight your users; there are few better ways to keep people using your software. But conversely, try never to surprise your user in a negative way.

This ties into a related rule known as KISS, which stands for, if you'll pardon the expression, Keep It Simple, Stupid!

Many developers fall into the trap of making too many aspects of their applications configurable, which often leads to a mess of a user interface.

For an excellent example of a simple, minimal user interface that blows away its competition, look no further than the Google home page. Their simple search box redefined internet search as Google's competitors' home pages were growing ever more complex and more difficult to navigate. Google’s simple and direct user interface gave users more confidence in the product, and now Google essentially owns internet searching.

The general rule is to design your application to please 80% percent of people. While that may sound like aiming too low, remember that you’ll never be able to please everyone.

Bear in mind, however, that simple for the user does not necessarily translate into simple for the developer. In fact, quite often, the simpler the interface is, the more complex the code behind it is.

FOR DISCUSSION:

1) In what ways are simple interfaces superior to complex interfaces?
2) When is a complex interface appropriate?
3) What are some ways that your application could delight rather than astonish your users?

16.6 It's Always Your Fault

Apps crash.

Data gets corrupted.

Files get lost.

What do all of these have in common? If they involve your application, they're all your fault.

That may sound harsh, but the reality is that you need to predict everything that could possibly go wrong and defend your user against it. This usually also involves defending your user against themselves, because many times, the user is their own worst enemy.

Design your application so that it minimizes the damage a user can do. Some of it goes back to the "User Data Is Sacred" section a few pages back. But you should be even more proactive than that.

For example, if your application has a TextField in which the user is only supposed to enter numbers, don't let them enter any letters (to see how, check out the Asc function and the KeyDown event).

If the user is supposed to enter a date, don't just hope that they enter it in the correct format. Some will use slashes and some will use hyphens. Some will use the US format and some will use the UK format. Some will use the business standard and some will use the academic standard. Some will just go ahead and spell out the month. If possible, use a third party date picker control or provide some popupmenus to guide the user in entering the date.

The key point is this: if you allow your user to enter invalid data, whatever happens after that is your fault. Allowing the user to do something sends the message that it's okay to do it, so allowing them to enter arbitrary text into a numeric field or a date field without warning them tells them that you're going to parse the data correctly. If you need the data in a certain way, make it easy for the user to enter it that way and extremely difficult (if not impossible) to enter it the wrong way.

You will be amazed at the things that users try to do with your applications. Some of these things will be great ideas that you can implement, but many of them will be things that quite honestly make you scratch your head and wonder.

In the last chapter, you learned about catching errors and exceptions. This is where errors and exceptions really come into play in the real world. It may sound cynical, but you really do need to prepare for your users to do destructive, seemingly random things. And if your code allows it, then it's your fault.

Related to this, if your application doesn't solve the problem it was intended to solve, that means that you didn't ask the right questions or enough of them early on.

FOR DISCUSSION:

1) **Aside from numeric and date inputs, what other formats might need special consideration?**

2) **In what ways can you make sure that your application is solving the right problems in the right ways?**

16.7 Plan For The Future

Most of the software you use isn't on the first version. Apple has been working on its Mac operating systems since before 1984, and they've been working on iOS since before 2007. Microsoft has versions of DOS and Windows stretching back decades as well. Word, Photoshop, Firefox, Google Chrome, and Xojo have all been expanded, enhanced, and upgraded over the years.

Your code will likely need to be updated as well. Your application might not be used for decades like some of these examples, but the chances that it will be perfect and complete at version 1.0 are very close to zero.

Because of this, you need to plan for the future. The best way to do this is to write your code to be read. This means that you should use logical and consistent method, function, and variable names. You should also stick to the DRY principle. And you should comment your code extensively. Someday down the line, someone will need to update your code, and that person will need to be able to figure out how the code works. And there's an excellent chance that person will be you. Remember that what’s fresh in your head now will likely be very stale a year from now.

So do your future self a favor, and write readable code now. And if it's not you who has to maintain it, then you will be making some other developer very happy.

Another way to future-proof your code is not to make too many assumptions. Write your code to be flexible. For example, you may be writing an application that currently only has to handle three files, but someday it might be more, and your application should be ready for that with minimal or no code changes.

FOR DISCUSSION:

1) What are some other practical ways to future-proof your code?

2) Imagine that you need to build a small application that communicates with a web-based social network. What are some future features that may need to be added in the future, and how could you prepare your code for them now?

16.8 Get It Working First

Another trap that many developers fall into is the optimization trap. When this happens, a developer will repeatedly delay shipping the application while continuing to work on small, often inconsequential speed improvements. But as Steve Jobs said, "Real artists ship." In other words, an application that no one but you ever uses may not count for much.

When it comes to optimization, take a lesson from Donald Knuth. You have probably never heard of Donald Knuth, but you owe him some thanks simply because you're using a computer. Born in 1938, Knuth is a pioneer in computer science and computer programming. One of his best known quotes about software development relates directly to optimization: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Perhaps Knuth was overstating things a bit for the sake of making his point, but it's a point worth making: don't get so caught up in making your application faster that you never get around to shipping it.

Knuth certainly wasn't alone in this opinion. His contemporary Michael Anthony Jackson (not to be confused with another Michael Jackson, the late King of Pop) said this: "The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet."

The danger in "premature optimization" is that you can become so caught up in the speed of your application that you allow it to compromise your design, and end up making some short-sighted decisions in the process.

This is related to another trap called feature creep, which occurs when a developer repeatedly tries to add "one more feature" to an application before shipping it. This happens quite often, and it is easy to fall into. But it can be avoided by drawing up two very important documents: the spec and the roadmap.

The spec, or specification, details exactly which features are required for shipping the application. The roadmap, on the other hand, lays out when, and sometimes how, new features will be added. For instance, the spec for your new email client should include items such as sending email, receiving email, contact management, and even spell check. These are the necessities. The roadmap is where you might find features like Facebook integration, a real time Twitter stream, and interactive Google Maps. Cool features, to be sure, but not necessary for the first version of your application.

FOR DISCUSSION:

1) **What other items might you find in the spec for a basic email client? And in the roadmap?**
2) **What do you think of the claim that "Real artists ship?"**

16.9 Documentation and Help

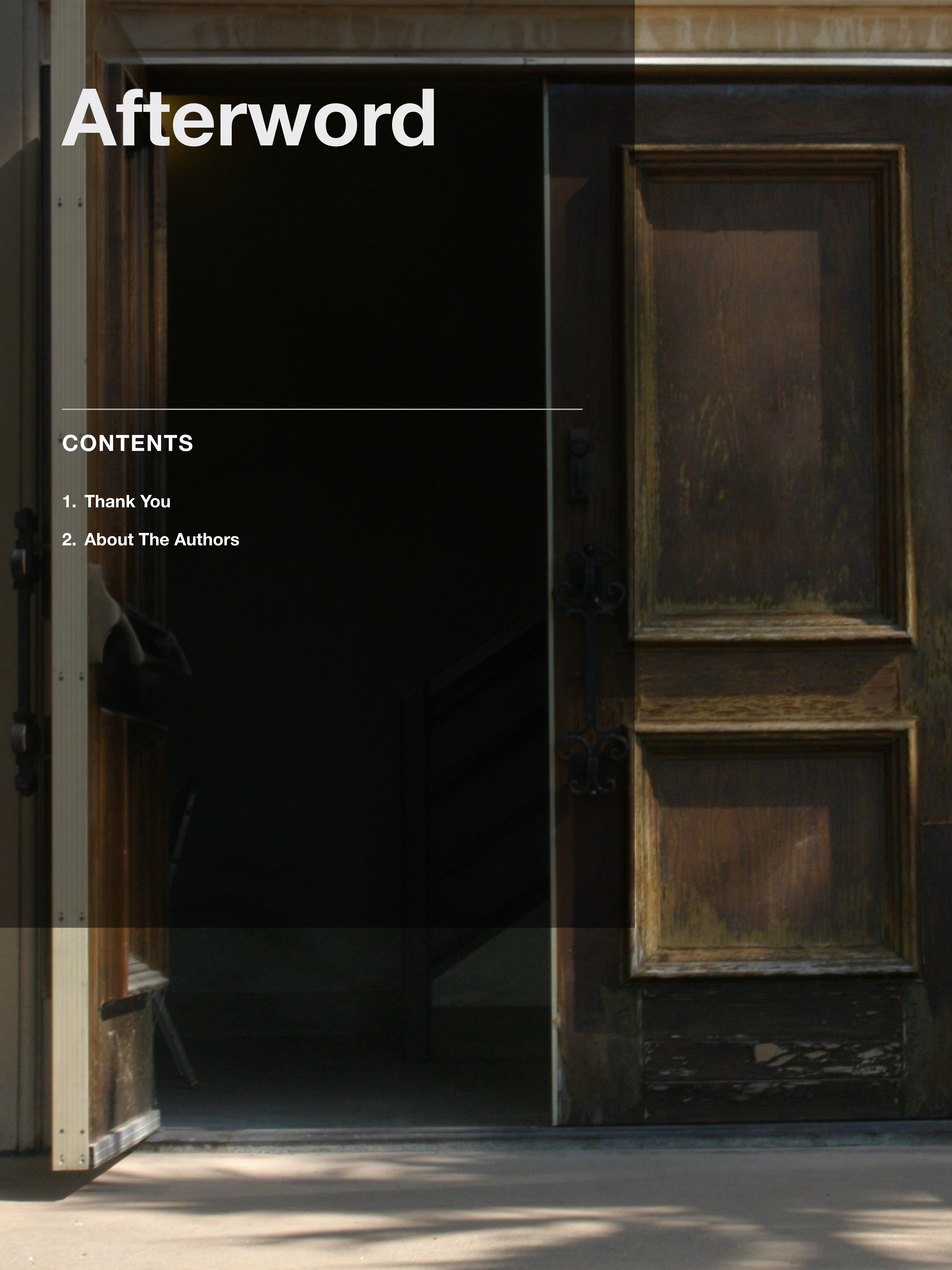
One of the most vital pieces of any application has nothing to do with your code: the documentation. It's a virtual guarantee that when you deliver the final build of an application, the users will want documentation. Whether this is in the form of a printed manual, a series of short tip sheets, screencasts, or an online help system, documentation will give your users a sense of security and confidence in your absence.

Some organizations develop their documentation first. In this case, the documentation essentially acts as the spec and guides any developers and designers along the way as the development of the application progresses.

This may sound like a contradiction, but at the same time, you should also strive to develop applications that are so simple to use that very few users need the documentation.

FOR DISCUSSION:

1) **What are some simple and practical ways you could include documentation with a Xojo application?**
2) **When was the last time you used an app's documentation and why?**



Afterword

CONTENTS

Thank You

Thanks for taking the time to read *Introduction to Programming with Xojo*. Whether you went through this book alone or as part of a class, I hope that it has provided you with some of the fundamentals of computer programming. You may not be destined for a career as a dedicated app developer, but it's highly likely that some programming skills will come in handy in our increasingly technological world.

If you have any questions, comments, or suggestions about anything in this book, please feel free to send an email to docs@xojo.com.

About The Authors

BRAD RHINE

Brad is a self-professed computer geek who has worked as a Computer Programmer, Web Developer, Technical Writer, Database Administrator, Assistant Director of Technology, and, briefly, Christmas Tree Salesman.

He is also a former columnist for XDev Magazine and has presented at the Xojo Developer Conference on many different topics.

Brad has spent most of his professional career working in the public school system.

When he's not writing code or writing about code, you'll find Brad playing his guitar, hanging out with his family, or running.

He lives in rural Pennsylvania with his wife and their two children, as well as a dog and two maladjusted cats.

PAUL LEFEBVRE

Paul is a Xojo Software Engineer who has also contributed greatly to the documentation, examples and more. He has been working with computers since first using an Atari 400 back in 1983.

Copyright & License