Chapter 1. Getting Started

This book teaches the Swift 3 programming language by exploring the development of three applications for Apple platforms: macOS, iOS, and watchOS. This book's approach might differ from what you're used to, because our philosophy is that the best way to learn Swift is to build apps using it! The vast majority of the code in this book will be part of the apps we're building—a full note-taking app for macOS, iOS, and watchOS—rather than individual pieces of sample code. You can see the final product in Figure 1-1.

Figure 1-1. Our finished app, for macOS, iOS, and watchOS

Our app is fully functional, but we do make some deliberate design and feature decisions along the way to constrain the scope a little (the book is almost 500 pages!). As we mentioned in the Preface, we assume that you're a reasonably capable programmer, but we don't assume you've ever developed for iOS or macOS, or used Swift or Objective-C before. We also assume that you're fairly comfortable navigating macOS and iOS as a user.

TIP

We recommend that you work through this book front to back, building the macOS app, then the iOS app, then the watchOS app, even if you're only interested in one of the platforms. By approaching the book this way, you'll get the best understanding of what building a real app with Swift requires.

Programming with Swift, and using the Cocoa and Cocoa Touch frameworks to develop macOS and iOS apps, respectively, involves using a set of tools developed by Apple. In this chapter, you'll learn about these tools, where to get them, how to use them, how they work together, and what they can do. At the end of this chapter, you'll make a very simple Swift application for iOS. Then we dive into the details of the Swift language and Apple's frameworks in the following two chapters.

TIP

The Apple development tools have a long and storied history. Originally a set of standalone application tools for the NeXTSTEP OS, they were eventually adopted by Apple for use as the official macOS tools. Later, Apple largely consolidated them into one application, known as Xcode, though some of the applications (such as Instruments and the iOS simulator) remain somewhat separate, owing to their relatively peripheral role in the development process. You'll notice the prefix `NS` on many of the classes you use for Cocoa and Cocoa Touch development with Swift. This prefix comes from the NeXTSTEP heritage of many of Apple's frameworks.

In addition to the development tools, Apple offers developers a paid membership in its Developer Program, which provides resources and support. The program allows access to online developer forums and specialized technical support for those interested in talking to the framework engineers. If you are just interested in learning Swift and exploring the development tools, you can do so for free. You will need a paid membership, however, if you wish to use developer services like iCloud in your apps or to distribute anything you build through either the iOS or macOS App Store.

NOTE

Swift is open source, but this doesn't really mean much when it comes to using it to develop apps for macOS, iOS, and watchOS. There's an excellent community of people working on the language that you can find at the Swift website.

With the introduction of Apple's curated App Stores for macOS, iOS, and watchOS, as well as emerging Apple platforms like tvOS, the Developer Program has become the official way for developers to provide their credentials when submitting applications to Apple—in essence, it is your ticket to selling apps through Apple. In this chapter, you'll learn how to sign up for the Apple Developer Program, as well as how to use Xcode, the development tool used to build apps in Swift.

The Apple Developer Program

The paid Apple Developer Program provides access to beta development tools, beta operating system releases, and distribution ability through Apple's app store. It also allows you to use some of the cloud-dependent features of the platforms, such as iCloud, CloudKit, In-App Purchase, and App Groups.

WARNING

We will be using a lot of cloud-dependent features, including iCloud, in the apps we build throughout this book. You will not be able to run these apps if you do not have a paid membership.

It isn't necessary to be a member of the Apple Developer Program if you don't intend to submit apps to the app stores, or don't need the cloud-dependent features. We strongly recommend joining, though, if you intend to build apps for any of Apple's platforms, as the other benefits are substantial:

- Access to the Apple Developer Forums, which are frequented by Apple engineers and designed to allow you to ask questions of your fellow developers and the people who wrote the OS.
- Access to beta versions of the OS before they are released to the public, which enables you to test your applications on the next version of the macOS, iOS, watchOS, and tvOS platforms, and make necessary changes ahead of time. You also receive beta versions of the development tools.
- A digital signing certificate (one for each platform) used to identify you to the App Stores. Without this, you cannot submit apps to the App Store, making a membership mandatory for anyone who wants to release software either for free or for sale via an App Store.

That said, registering for the Developer Program isn't necessary to view the documentation or to download the current version of the developer tools, so you can play around with writing apps without opening your wallet.

Registering for the Apple Developer Program

To register for the Developer Program, you'll first need an Apple ID. It's quite likely that you already have one, as the majority of Apple's online services require one to identify you. If you've ever used iCloud, the iTunes store (for music or apps), or Apple's support and repair service, you already have an ID. You might even have more than one (one of this book's authors has four). If you don't yet have an ID, you'll create one as part of the registration process. When you register for the Developer Program, the membership gets added to your Apple ID.

TIP

If you don't want to register for the paid developer program, you can skip to "Downloading Xcode" for instructions on installing Xcode, the developer tools.

Once again, keep in mind that you won't be able to build the apps that we teach in this book if you don't have a paid membership, as we use cloud-dependent features such as iCloud.

There are alternatives to many of Apple's tools—such as the Google Maps SDK for iOS, or cloud-storage services from Amazon and Microsoft. However, you'll still need a paid membership through Apple to put apps in the iTunes App Store.

Once you're on the Apple Developer Program website, simply click Enroll, and follow the steps to enroll.

You can choose to register as an individual or as a company. If you register as an individual, your apps will be sold under your name. If you register as a company, your apps will be sold under your company's legal name. Choose carefully, as it's very difficult to convince Apple to change your program's type.

If you're registering as an individual, you'll just need your credit card. If you're registering as a company, you'll need your credit card as well as documentation that proves you have authority to bind your company to Apple's terms and conditions.

NOTE

For information on code signing and using Xcode to test and run your apps on your own physical devices, see Apple's App Distribution Guide. We don't cover this in the book, as it's a process that changes often.

Apple usually takes about 24 hours to activate an account for individuals, and longer for companies. Once you've received confirmation from Apple, you'll be emailed a link to activate your account; when that's done, you're a full-fledged developer!

Downloading Xcode

To develop apps for either platform, you'll use Xcode, Apple's integrated development environment. Xcode combines a source code editor, debugger, compiler, profiler, iOS simulator, Apple Watch simulator, and more into one package. It's where you'll spend the majority of your time when developing applications.

NOTE

At the time of writing, Xcode is only available for Mac, but who knows what the future holds for the iPad Pro?

You can get Xcode from the Mac App Store. Simply open the App Store application and search for "Xcode," and it'll pop up. It's a free download, though it's rather large (several gigabytes at the time of writing).

Once you've downloaded Xcode, it's straightforward enough to install it. The Mac App Store gives you an application that on first launch sets up everything you need to use Xcode. Just launch the downloaded app, and follow the prompts, and you'll be up and running in no time.

NOTE

This book covers Swift 3, which is available only if you're using Xcode 8 or later. Make sure you're using the latest version of Xcode from the Mac App Store. It's good practice to use the latest Xcode at all times.

Creating Your First Project with Xcode

Xcode is designed around a single window. Each of your projects will have one window, which adapts to show what you're working on.

To start exploring Xcode, you'll first need to create a project by following these steps:

1. Launch Xcode. You can find it by opening Spotlight (by pressing ⌘-space bar) and typing `Xcode`. You can also find it by opening the Finder, going to your hard drive, and opening the *Applications* directory. If you had any projects open previously, Xcode will open them for you. Otherwise, the "Welcome to Xcode" screen appears (see Figure 1-2).

   Figure 1-2. The "Welcome to Xcode" screen

2. Create a new project by clicking "Create a new Xcode project" or go to File→New→Project.

   You'll be asked what kind of application to create. The template selector is divided into two areas. On the lefthand side, you'll find a collection of application categories. You can choose to create an iOS, watchOS, or macOS application from the project templates, which will set up a project directory to get you started. Because we're just poking around Xcode at the moment, it doesn't really matter what we select, so choose Application under the iOS header and select Single View Application. This creates an empty iOS application and displays the project settings window shown in Figure 1-3.

   Figure 1-3. The project settings window

3. Name the application. Enter `HelloSwift` in the Product Name section.

4. Enter information about the project. Depending on the kind of project template you select, you'll be asked to provide different information about how the new project should be configured.

   At a minimum, you'll be asked for the following information, no matter which platform and template you choose:
   The product's name

   This is the name of the project and is visible to the user. You can change this later.

Your organization's name

This is the name of your company or group. It's not directly used by Xcode, but new source code files that you create will mention it.

Your organization identifier

This is used to generate a *bundle ID*, a string that looks like a reverse domain name (e.g., if O'Reilly made an application named MyUsefulApplication, the bundle ID would be *com.oreilly.MyUsefulApplication*).

NOTE

Bundle IDs are the unique identifier for an application, and are used to identify that app to the system and to the App Store. Because each bundle ID must be unique, the same ID can't be used for more than one application in either of the iOS or Mac App Stores. That's why the format is based on domain names—if you own the site *usefulsoftware.com*, all of your bundle IDs would begin with *com.usefulsoftware*, and you won't accidentally use a bundle ID that someone else is using or wants to use because nobody else owns the same domain name.

If you don't have a domain name, enter anything you like, as long as it looks like a backward domain name (e.g., *com.mycompany* will work).

NOTE

If you plan on releasing your app, either to the App Store or elsewhere, it's very important to use a company identifier that matches a domain name you own. The App Store requires it, and the fact that the operating system uses the bundle ID that it generates from the company identifier means that using a domain name that you own eliminates the possibility of accidentally creating a bundle ID that conflicts with someone else's.

If you're writing an application for the Mac App Store, you'll also be prompted for the App Store category (whether it's a game, an educational app, a social networking app, or something else).

Depending on the template, you may also be asked for other information (e.g., the file extension for your documents if you are

creating a document-aware application, such as a Mac app). You'll also be asked which language you want to use; because this book is about Swift, you should probably choose Swift! The additional information needed for this project is covered in the following steps.

5. Make the application run on the iPhone by choosing iPhone from the Devices drop-down list.

NOTE

iOS applications can run on the iPad, iPhone, or both. Applications that run on both are called "universal" applications and run the same binary but have different user interfaces. For this exercise, just choose iPhone. You should be building universal iOS apps in general, and we'll be doing that when we properly start on iOS in Part III.

6. Leave the rest of the settings as shown in Figure 1-4. Click Next to create the project.

Figure 1-4. The project settings

7. Choose where to save the project. Select a location that suits you. We recommend putting all your work related to this book (and other Swift programming learning you might do) in one folder. You might notice a little checkbox for Source Control; this creates a source code control repository for your code, giving you a place where you can save and manage different versions of your code as you create them. While in general this is a good idea to use, for this example project, make sure this is *unchecked*.

Once you've done this, Xcode will open the project, and you can now start using the entire Xcode interface, as shown in Figure 1-5.

Figure 1-5. The entire Xcode interface

The Xcode Interface

As mentioned, Xcode shows your entire project in a single window, which is divided into a number of sections. You can open and close each section at will, depending on what you want to see.

Let’s take a look at each of these sections and examine what they do.

THE EDITOR

The Xcode editor (Figure 1-6) is where you’ll be spending most of your time. All source code editing, interface design, and project configuration take place in this section of the application, which changes depending on which file you have open.

If you’re editing source code, the editor is a text editor, with code completion, syntax highlighting, and all the usual features that developers have come to expect from an integrated development environment. If you’re modifying a user interface, the editor becomes a visual editor, allowing you to drag around the components of your interface. Other kinds of files have their own specialized editors as well.

When you first create a project, the editor will start by showing the project settings, as seen in Figure 1-6.

Figure 1-6. Xcode’s editor, showing the project settings

The editor can also be split into a *main editor* and an *assistant editor* through the *editor selector*. The assistant shows files that are related to the file open in the main editor. It will continue to show files that have a relationship to whatever is open, even if you open different files.

For example, if you open an interface file and then open the assistant, the assistant will, by default, show related code for the interface you’re editing. If you open another interface file, the assistant will show the code for the newly opened files.

At the top of the editor, you’ll find the *jump bar*. The jump bar lets you quickly jump from the content that you’re editing to another piece of related content, such as a file in the same folder. The jump bar is a fast way to navigate your project.

THE TOOLBAR

The Xcode toolbar (Figure 1-7) acts as mission control for the entire interface. It’s the only part of Xcode that doesn’t significantly change as

you develop your applications, and it serves as the place where you can control what your code is doing.

Figure 1-7. Xcode's toolbar

From left to right, after the macOS window controls, the toolbar features the following items:

Run button (Figure 1-8)

Clicking this button instructs Xcode to compile and run the application.

Figure 1-8. The Run button

Depending on the kind of application you're running and your currently selected settings, this button will have different effects:

- If you're creating a Mac application, the new app will appear in the Dock and will run on your machine.
- If you're creating an iOS application, the new app will launch in either the iOS simulator or on a connected iOS device, such as an iPhone or iPad.
  Additionally, if you click and hold this button, you can change it from Run to another action, such as Test, Profile, or Analyze. The Test action runs any unit tests that you have set up; the Profile action runs the application Instruments (we cover this much later, in Chapter 16); and the Analyze action checks your code and points out potential problems and bugs.

Stop button (Figure 1-9)

Clicking this button stops any task that Xcode is currently doing—if it's building your application, it stops; and if your application is running in the debugger, it quits it.

Figure 1-9. The Stop button

Scheme selector (Figure 1-10)

Schemes are what Xcode calls build configurations—that is, what's being built, how, and where it will run (i.e., on your computer or on a connected device).

Figure 1-10. The scheme selector

Projects can have multiple apps inside them. When you use the scheme selector, you choose which app, or *target*, to build.

To select a target, click the lefthand side of the scheme selector.

You can also choose where the application will run. If you are building a Mac application, you will almost always want to run the application on your Mac. If you're building an iOS application, however, you have the option of running the application on an iPhone simulator or an iPad simulator. (These are in fact the same application; it simply changes shape depending on the scheme that you've selected.) You can also choose to run the application on a connected iOS device if it has been set up for development.

Status display (Figure 1-11)

The status display shows what Xcode is doing—building your application, downloading documentation, installing an application on an iOS device, and so on.

Figure 1-11. The status display

If there is more than one task in progress, a small button will appear on the lefthand side, which cycles through the current tasks when clicked.

Editor selector (Figure 1-12)

The editor selector determines how the editor is laid out. You can choose to display either a single editor, the editor with the assistant, or the versions editor, which allows you to compare different versions of a file if you're using a revision control system like Git or Subversion.

Figure 1-12. The editor selector

NOTE

We don’t have anywhere near the space needed to talk about using version control in your projects in this book, but it’s an important topic. We recommend Jon Loeliger and Matthew McCullough’s *Version Control with Git, 2nd Edition* (O’Reilly).

View selector (Figure 1-13)
: The view selector controls whether the navigator, debug, and utility panes appear on screen. If you’re pressed for screen space or simply want less clutter, you can quickly summon and dismiss these parts of the screen by clicking each of the elements.

Figure 1-13. The view selector

THE NAVIGATOR

The lefthand side of the Xcode window is the *navigator*, which presents information about your project (Figure 1-14).

Figure 1-14. The navigator pane has eight tabs at the top

The navigator is divided into eight tabs, from left to right:

Project navigator
: Lists all the files that make up your project. This is the most commonly used navigator, as it determines what is shown in the editor. Whatever is selected in the project navigator is opened in the editor.

Symbol navigator
: Lists all the classes and functions that exist in your project. If you’re looking for a quick summary of a class or want to jump directly to a method in that class, the symbol navigator is a handy tool.

Search navigator
: Allows you to perform searches across your project if you’re looking for specific text. (The shortcut is ⌘-Shift-F. Press ⌘-F to search the current open document.)

Issue navigator

Lists all the problems that Xcode has noticed in your code. This includes warnings, compilation errors, and issues that the built-in code analyzer has spotted.

Test navigator

Shows all the unit tests associated with your project. Unit tests used to be an optional component of Xcode but are now built into Xcode directly. Unit tests are discussed much later, in "Unit Testing".

Debug navigator

Activated when you're debugging a program, and it allows you to examine the state of the various threads that make up your program.

Breakpoint navigator

Lists all of the breakpoints that you've set for use while debugging.

Report navigator

Lists all the activity that Xcode has done with your project (such as building, debugging, and analyzing). You can go back and view previous build reports from earlier in your Xcode session, too.

UTILITIES

The utilities pane (Figure 1-15) shows additional information related to what you're doing in the editor. If you're editing a Swift source file, for example, the utilities pane allows you to view and modify settings for that file.

The utilities pane is split into two sections: the *inspector*, which shows extra details and settings for the selected item; and the *library*, which is a collection of items that you can add to your project. The inspector and the library are most heavily used when you're building user interfaces; however, the library also contains a number of useful items, such as file templates and code snippets, which you can drag and drop into place.

Figure 1-15. The utilities pane, showing information for a source file

THE DEBUG AREA

The debug area (Figure 1-16) shows information reported by the debugger when the program is running. Whenever you want to see what the application is reporting while running, you can view it in the debug area. By default the debug area is not shown unless there is a program running. You can bring up the debug area by using the Xcode Toolbar View selector middle button.

Figure 1-16. The debug area

The area is split into two sections: the lefthand side shows the values of local variables when the application is paused; the righthand side shows the ongoing log from the debugger, which includes any logging that comes from the debugged application.

You can show or hide the debug area by clicking the view selector, at the top right of the window (see Figure 1-17).

Figure 1-17. The central button in the view selector, which hides and shows the debug area

Developing a Simple Swift Application

Through the bulk of this book, we'll be developing a complex, full-fledged Swift application, spanning three of Apple's platforms: macOS, iOS, and watchOS. But for now, before we even explore how and why Swift itself works, we're going to get a brief taste by building a very, very simple application for iOS.

NOTE

If you're more interested in Mac development, don't worry! Exactly the same techniques apply, and we'll be exploring Mac apps in detail later on, in Part II.

This simple application is extremely cutting-edge: it will display a single button that, when tapped, will pop up an alert and change the button's

label to “Test!” We’re going to build on the project we created earlier in “Creating Your First Project with Xcode”, so make sure you have that project open.

It’s generally good practice to create the interface first and then add code. This means that your code is written with an understanding of how it maps to what the user sees.

To that end, we’ll start by designing the interface for the application.

Designing the Interface

When building an application’s interface using Cocoa and Cocoa Touch, you have two options. You can either design your application’s screens in a *storyboard*, which shows how all the screens link together, or you can design each screen in isolation. As a general rule, storyboards are a better way to create your interfaces even if you only have a single view, as in the case of this first application we are building. The reason is that if you later want to give your application more than one view, it will be easier to do that in a storyboard.

Start by opening the interface file and adding a button. These are the steps you’ll need to follow:

1. First, we’ll need to open the main storyboard. Because newly created projects use storyboards by default, your app’s interface is stored in the *Main.storyboard* file.
   Open it by selecting *Main.storyboard* in the project navigator. The editor will change to show the application’s single, blank frame. You may need to pan or zoom the view around to fit it on your monitor.
2. Next, we need to drag in a button. We’re going to add a single button to the frame. All user interface controls are kept in the *Object library*, which is at the bottom of the utilities pane on the righthand side of the screen.
   To find the button, you can either scroll through the list until you find Button, or type `button` in the search field at the bottom of the library.

   Once you’ve located it, drag it into the frame.

3. At this point, we need to configure the button. Every item that you add to an interface can be configured. For now, we'll change only the text on the button.

   Select the new button by clicking it, and select the Attributes Inspector, which is the fourth tab from the right at the top of the utilities pane. You can also reach it by pressing ⌘-Option-4.

   There are many attributes on the button; look for the one labeled Title. The Title attribute has two different components inside it: a drop-down box and a text field containing "Button." In the text field, change the button's Title to "Hello!"

   TIP

   You can also change the button's title by double-clicking it in the interface.

Our simple interface is now complete (Figure 1-18). The only thing left to do is to connect it to code.

Figure 1-18. Our completed simple interface

Connecting the Code

Applications aren't just interfaces—as a developer, you also need to write code. To work with the interface you've designed, you need to create connections between your code and your interface.

There are two kinds of connections that you can make:

Outlets
: Variables that refer to objects in the interface. Using outlets, you can instruct a button to change color or size, or to hide itself. There are also *outlet collections*, which allow you to create an array of outlets and choose which objects it contains in the interface builder.

Actions
: Methods in your code that are run in response to the user interacting with an object. These interactions include the user touching a finger to an object, dragging a finger, and so on.

To make the application behave as we've just described—tapping the button displays a label and changes the button's text—we'll need to use both an outlet and an action. The action will run when the button is tapped and will use the outlet connection to the button to modify its label.

To create actions and outlets, you need to have both the interface builder and its corresponding code open. Then hold down the Control key and drag from an object in the interface builder to your code (or to another object in the interface builder, if you want to make a connection between two objects in your interface).

We'll now create the necessary connections:

1. First, open the assistant by selecting the second button in the editor selector in the toolbar. The symbol is two interlocking circles.
   The assistant should open and show the corresponding code for the interface *ViewController.swift*. If it doesn't, click the intertwining circles icon (which represents the assistant) inside the jump bar and navigate to Automatic→*ViewController.swift*. Make sure you don't select the assistant symbol in the toolbar, as that will close the assistant editor.
2. Create the button's outlet. Hold down the Control key and drag from the button into the space below the first `{` in the code.
   A pop-up window will appear. Leave everything as the default, but change the Name to **`helloButton`**. Click Connect.

   A new line of code will appear: Xcode has created the connection for you, which appears in your code as a property in your class:
   ```
   @IBOutlet weak var helloButton : UIButton!
   ```
3. Create the button's action. Hold down the Control key, and again drag from the button into the space below the line of code we just created. A pop-up window will again appear.
   This time, change the Connection from Outlet to Action, set the Name to `showAlert`, and click Connect.
   More code will appear. Xcode has created the connection, which is a method inside the `ViewController` class:
   ```
   @IBAction func showAlert(sender: Any) {
   }
   ```
4. In the `showAlert` method you just created, add in the new code:
   ```
   5. let alert = UIAlertController(title: "Hello!", message: "Hello,
   world!",
   ```

```
6.         preferredStyle: UIAlertControllerStyle.alert)
7. alert.addAction(UIAlertAction(title: "Close",
8.         style: UIAlertActionStyle.default, handler: nil))
9. self.present(alert, animated: true, completion: nil)
   self.helloButton.setTitle("Test!", forState: UIControlState.normal)
```

This code does the following things:

It creates a `UIAlertController`, which displays a message to the user in a pop-up window. It prepares it by setting its title to “Hello!” and the text inside the window to “Hello, world!”

Finally, an action that dismisses the alert is added, with the text “Close”.

The alert is then shown to the user.

Finally, it sets the title of the button to “Test!”

The application is now ready to run. Click the Run button in the upper-left corner. The application will launch in the iPhone simulator. Don’t worry if the app takes a while to launch the first time; the simulator can take a fair amount of time on first launch.

NOTE

If you happen to have an iPhone or iPad connected to your computer, Xcode will try to launch the application on the device rather than in the simulator. To make Xcode use the simulator, go to the Scheme menu in the upper-left corner of the window and change the selected scheme to the simulator.

When the app finishes launching in the simulator, tap the button. An alert will appear; when you close it, you’ll notice that the button’s text has changed.

Using the iOS Simulator

The iOS simulator (Figure 1-19) allows you to test out iOS applications without having to use actual devices. It's a useful tool, but keep in mind that the simulator behaves very differently compared to a real device.

Figure 1-19. The iOS simulator

For one thing, the simulator is a lot faster than a real device and has a lot more memory. That's because the simulator makes use of your computer's resources—if your Mac has 8 GB of RAM, so will the simulator, and if you're building a processor-intensive application, it will run much more smoothly on the simulator than on a real device.

The iOS simulator can simulate many different kinds of devices: everything from the iPad 2 to the latest iPad Pro, and from the Retina display 3.5- and 4-inch iPhone-sized devices to the latest 4.7-inch and 5.5-inch iPhones. It can also test on variable-size devices.

To change the device, open the Hardware menu, choose Device, and select the device you want to simulate. You can also change which simulator to use via the scheme selector in Xcode. Each simulator device is unique, and they do not share information. So if you want to test your application on different simulators, you will need to build it again through Xcode.

WARNING

If you change hardware in the simulator while running an app, it will crash and Xcode will alert you. Be wary of changing the hardware in the simulator while testing applications unless you really like crashes.

You can also simulate hardware events, such as the Home button being pressed or the iPhone being locked. To simulate pressing the Home button, you can either choose Hardware→Home, or press ⌘-Shift-H. To lock the device, press ⌘-L or choose Hardware→Lock.

There are a number of additional features in the simulator, which we'll examine more closely as they become relevant to the various parts of iOS we'll be discussing.

Conclusion

In this chapter, we've looked at the basics of the Apple Developer Program, as well as the tools used for building apps. We've also made a really quick and simple iOS app, just to give you a taste of the process. In the next two chapters, we'll look at the Swift programming language, using a feature of Xcode and Swift called Playgrounds to work with Swift code outside the application context.