Learn Swift, Apple's new language for native app development

Learn Swift on the Mac

For OS X and iOS

Waqar Malik

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Whenever developers come to a new platform, they are faced with the task of getting to know unfamiliar development tools, design patterns, the standard frameworks available in the new environment, and perhaps even a new programming language.

Most of the time, this is all done while trying to deliver an application as soon as possible. In such situations, developers tend to fall back on the patterns and approaches they are familiar with from previous environments, which too often results in code that doesn't fit the new environment, or in duplicate code that might already be provided by the built-in frameworks. This can cause problems down the road or delays in delivery.

It would be great to have colleagues already familiar with the platform who could offer guidance to get you going in the right direction. Well, it's not always possible to have mentors to help you, and that's where this books steps in—to be your mentor.

The author of this book is a veteran of Apple's Developer Technical Services organization, and has answered countless questions from software engineers who are new to Apple technology. That experience results in a book that anticipates the most common misunderstandings and takes care to explain not only the how, but also the why of Apple's development platform.

For example, the conceptual basis provided in Chapter 4, "Introduction to Object-Oriented Programming," gives you the means to place the material that follows into a coherent picture, instead of just tossing you into a flurry of unfamiliar classes, methods, and techniques and hoping you'll somehow sort it all out with practice.

Learn Swift on the Mac provides a step-by-step guide that will help you acquire the skills you need to develop applications for iOS and OS X.

Chapter 1

Hello Swift

Swift is a new language designed by Apple for developing iOS and OS X applications. It takes the best parts of C and Objective-C and adapts them with modern features and patterns. Swift-compiled programs will run on iOS7 or newer and OS X 10.9 (Mavericks) or newer.

The two main goals for the language are compatibility with the Cocoa and Cocoa Touch frameworks, and safety, as you'll see in the upcoming chapters. If you've been using Objecive-C, especially the modern syntax, Swift will feel familiar.

But Swift's syntax is actually a major departure from Objective-C. It takes lots of cues from programming languages like Haskell, C#, Ruby, and Python.

Some of the technologies I'll cover in this book are:

- Automatic reference counting
- Closures (blocks)
- Collection literals
- Modules
- Frameworks
- Objective-C runtime
- Generics
- Operator overloading
- Tuples
- Namespaces

Improvements over Objective-C

Let's take a quick look atsome of the features that make Swift better than Objective-C. I'll cover these in detail in later chapters.

Type Inference

In Swift, there is usually no need to specify the type of variables (though you can always specify them); the types of the variables can be inferred by the value being set.

Type Safety

Conversion between types is done explicitly. The compiler knows more about types in method calls and can use table look-up for methods for dispatch instead of the dynamic dispatch that Objective-C uses. Static dispatch via table look-up enables more checks at compile time, even in the playground. As soon as you enter an expression in the playground, the compiler evaluates it and lets you know of any possible issues with the statement; you can't run your program until you fix those issues. Here are some features that enhance safety:

- Variables and constants are always initialized
- Array bounds are always checked.
- Raw C pointers are not readly available.
- Assignments do not return values.
- Overflows are trapped as runtime errors.

Control Flow

The switch statement has undergone a major overhaul. Now it can select based not only integers, but also on strings, floats, ranges of items, expressions, enums, and so forth. Moreover, there's no implicit fall-through between case statements.

Optionals

Variables can now have optional values. What does that mean? It means a variable will either be nil or it will have a valid value. The nil value is distinct from any valid value. Optionals can also be chained together to protect against errors and exceptions.

Strings

Strings in Swift are much easier to work with, with a clear, simple syntax. You can concatenate strings using the += operator. The mutability of the strings is defined by the language, not the `String` object. You declare a string as either mutable or nonmutable with the same `String` object, by using either the `let` or `var` keywords.

Unicode

Unicode is supported at the core: You can define variables names and function names using full Unicode. The `String` and `Character` types are also fully Unicode-compliant and support various encodings, such as UTF-8, UTF-16, and 21-bit Unicode scalers.

Other Improvements

- Header files are no longer required.
- Functions are full-fledged objects; they can be passed as arguments and returned from other fuctions. Functions can be scoped similarly to instance variables.
- Comments can be nested.
- There are separate operators for assignment (=) and comparison (==), and there's even an identity operator (===) for checking whether two data elements refer to same object.
- There is no defined entry point for programs such as main.

If all this sounds good to you (it does to me), let's go get the tools to start playing with Swift.

Requirements

Before you can begin playing with Swift, you need to download and install Xcode, the IDE that's used to build applications for iOS and OS X. You'll need Xcode 6.1 or later.

It's really easy to download and install Xcode. Here are the basic requirements:

- Intel-based Macintosh computer
- OS X 10.10 Yosemite (or later)
- Free disk space
- An Internet connection
- An iOS device running iOS 7 (or later)

> **Note** As a rule, the later the version of the OS, the better. The examples in the book are developed using Xcode 6.1 running on OS X 10.10 Yosemite and for iOS 8 running on iPhone 5S.

Getting Xcode

Launch the App Store application and use the search bar on the top right to search for Xcode. . You can then get more information by selecting Xcode, as shown in Figure 1-1, or install it by selecting the Install button.

Figure 1-1. Xcode on App Store

When you launch Xcode for first time, it will download and install other required items in order to complete the installation. If you have mutltiple versions of Xcode installed, be sure to select Xcode version 6.1 or later for the command-line tools. You can do this by selecting **Xcode ➤ Preferences**, then choosing the **Locations** tab as shown in Figure 1-2.

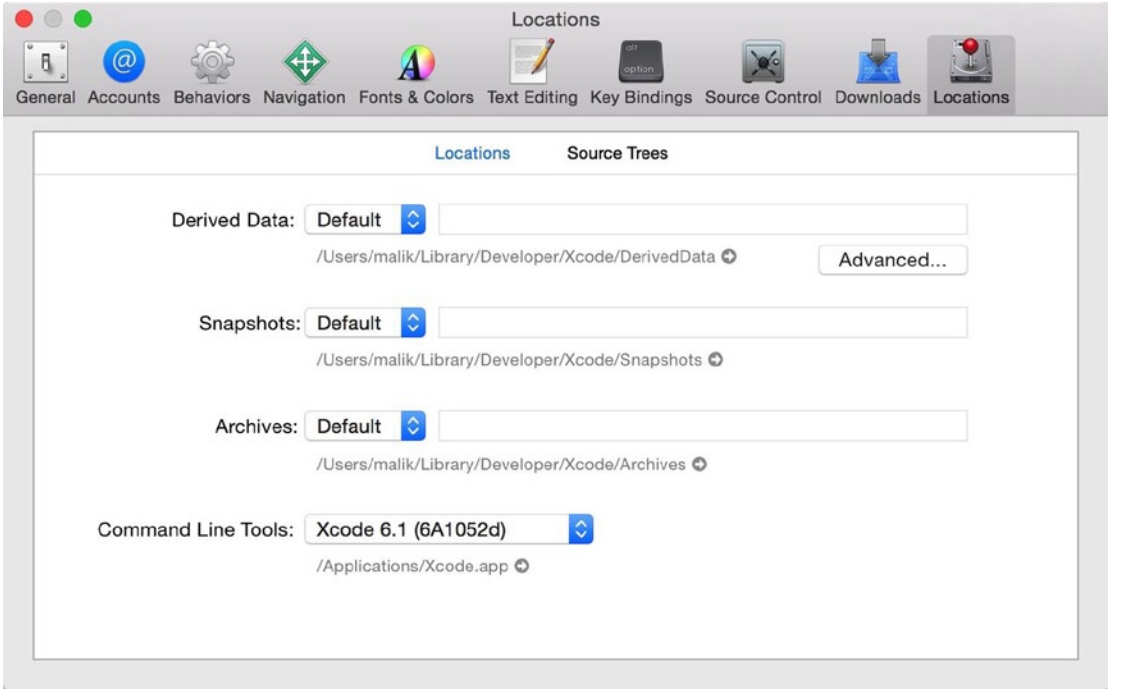

Figure 1-2. Selecting the command-line tools

Quick Tour of Xcode

If you launch Xcode without opening a project, you'll see the screen shown in Figure 1-3. You can create or open existing projects or a playground.

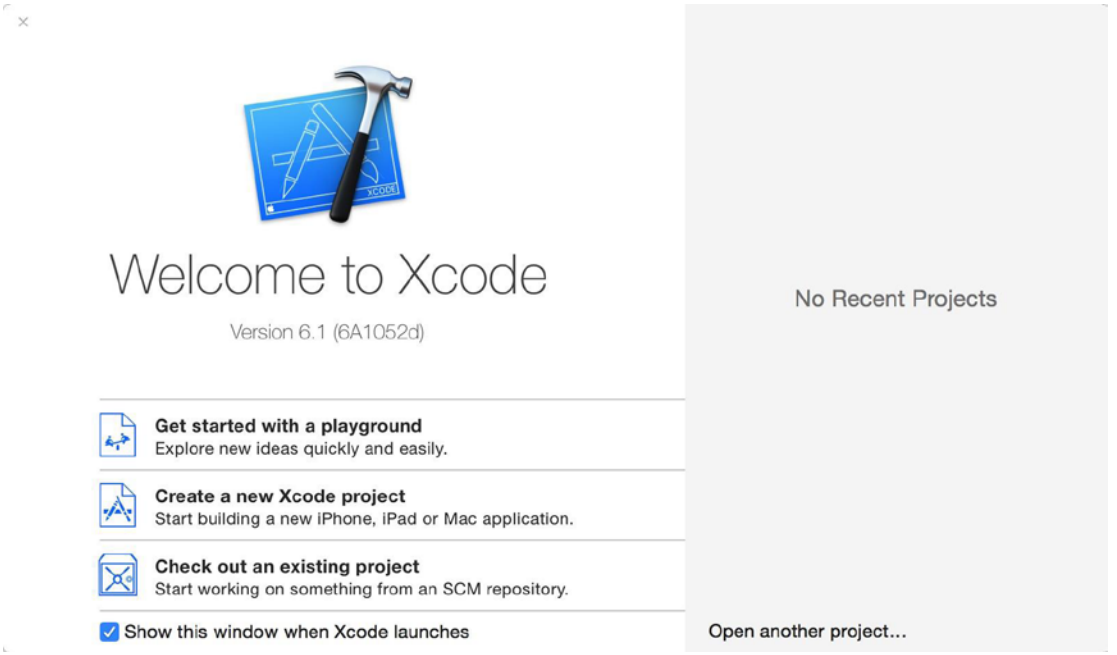

Figure 1-3. Xcode Welcome Screen

Let's start by creating a new playground, using the **Get started with a playground** option. As Figure 1-4 shows, the next screen asks you to name your playground and pick the operating system framework you'd like the playground to use. For now, just pick the default iOS and name your playground `Learn Swift`, then select Next to save your playground on your computer.

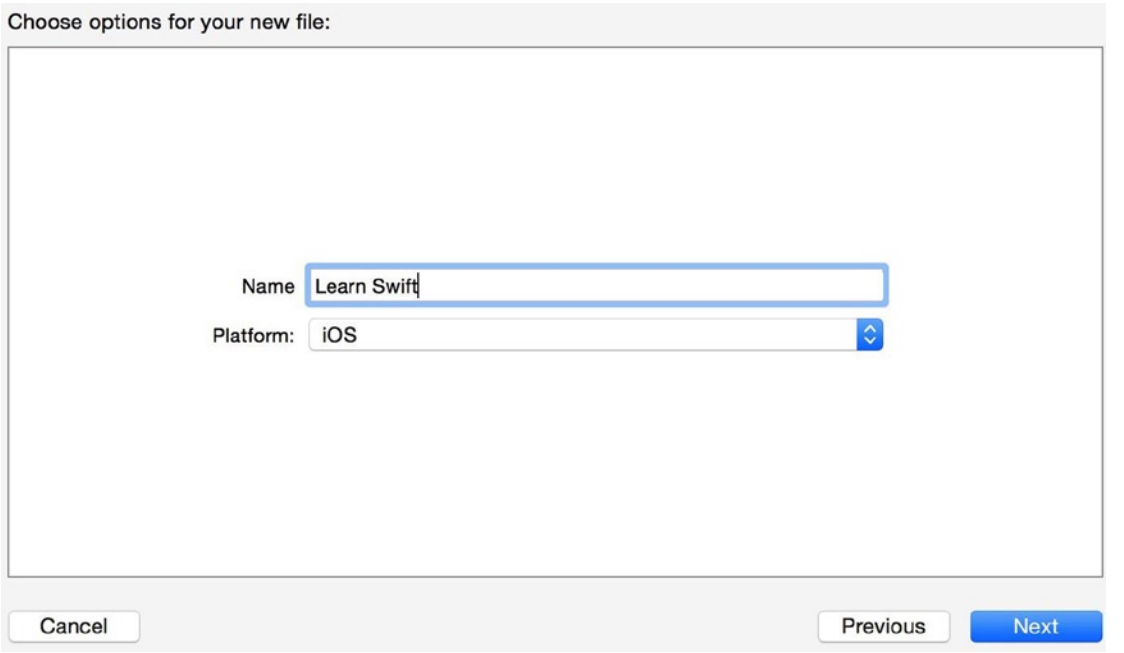

***Figure 1-4.** Naming a playground*

And now you're ready to play with Swift. As you can see in Figure 1-5, line numbers are not on by default. To turn them on—and set all of your text editing preferences—select **Xcode ➤ Preferences** again and then select the Text Editing tab. The first option you'll see lets you enable line numbers, so you'll be able to easily find a line as I discuss it.

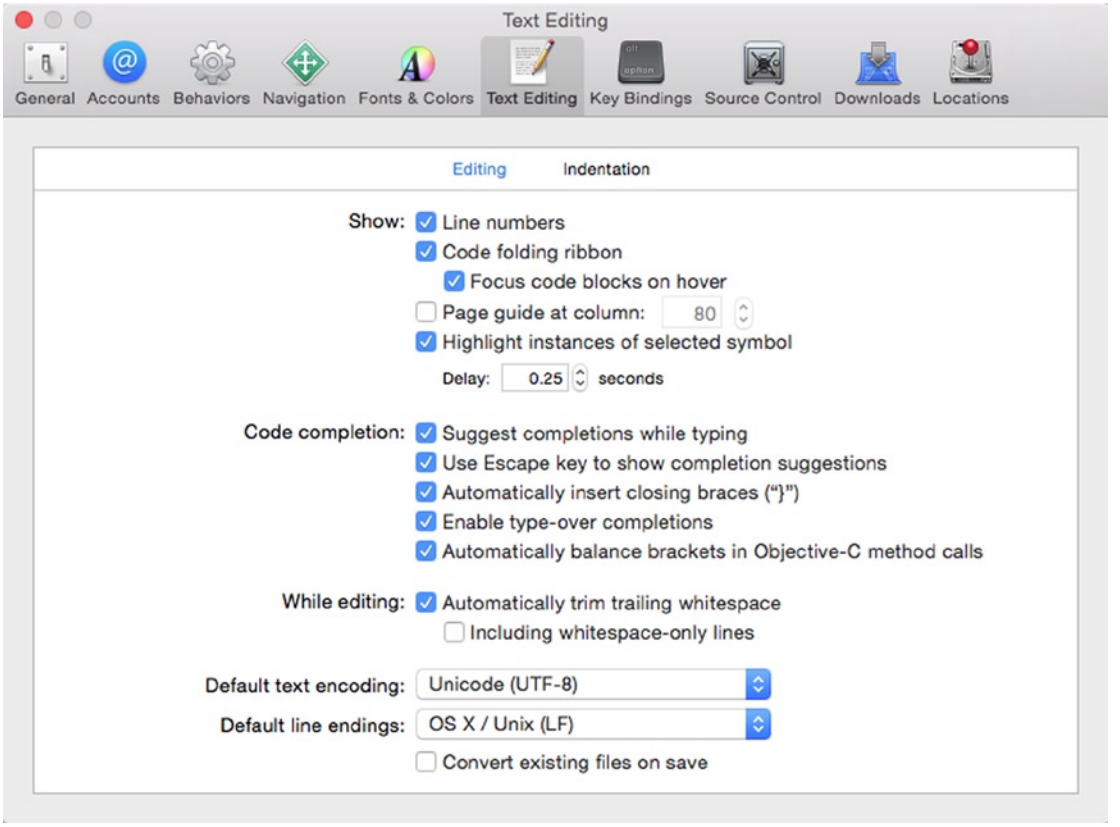

Figure 1-5. *Setting text-editing Preferences*

Once you have created the playground and updated the preferences. You will be greeted by the playground window as shown in Figure 1-6.

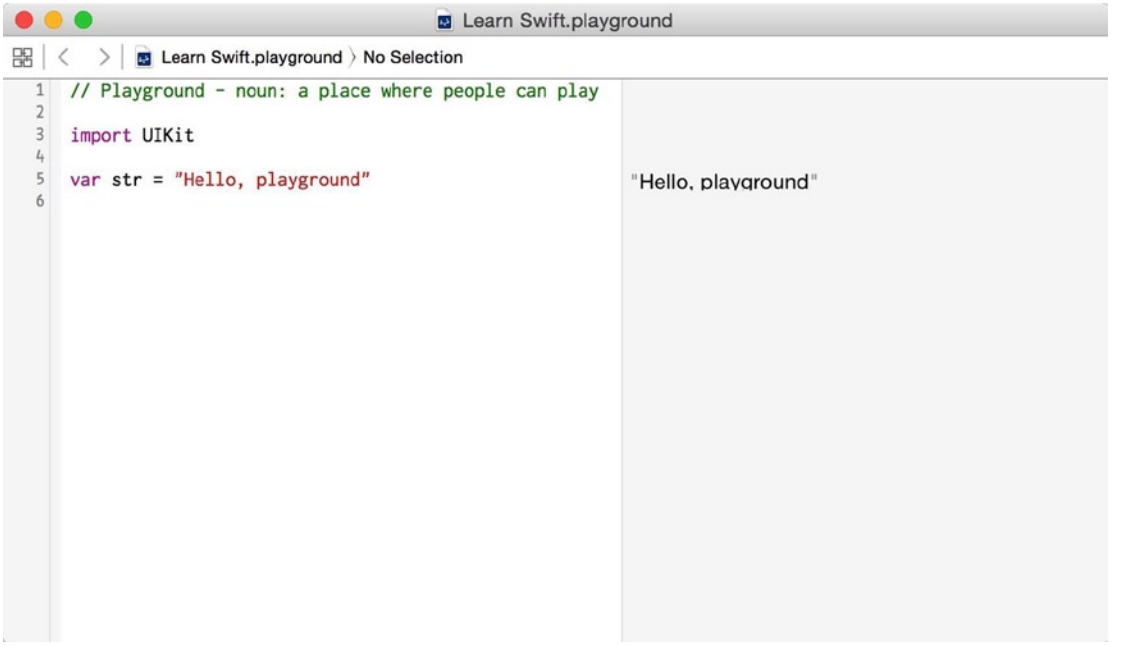

Figure 1-6. *Interactive playground window*

There are two parts to the playground. On the left, is the editor where you write code, and on the right is a sidebar that shows what happens when the code runs. As you can see, there is already some code in the file.

Line 1 shows a single-line comment, which starts with `//` and ends with new line. You can have as many as you like, but each must start on a new line. You can also use multiline comments, which start with `/*` and end with `*/` and can span multiple lines, like this:

```
/* this the first line of comment,
   it continues on the second line */
```

Line 3 tells the compiler to import the iOS Cocoa Touch API so I can use it if I want.

> **Note** Cocoa is the framework that defines the API for developing OS X applications. Cocoa Touch is the equivalent for iOS. Sometimes Cocoa is used to mean both OS X and iOS and, in that case, the desktop version is referred to as AppKit.

Notice that there's no semicolon (;) at the end of the `import` statement, In Swift, semicolons are optional; they are required only if you put more that one statements on a line, such as `var a = 4; var b = 8.`

Line 5 is a variable declaration, which shows the keyword `var` and the name of the variable, and assigns the initial value to the variable. As you'd expect, the keyword `var` tells the compiler I'm declaring a variable. Then I give it a name; the default name is str but it could be any valid string (as I'll discuss in a later chapter).

Quick Tour of Swift

Whenever you learn a new language, there's a long-standing tradition that your first program is one that displays "Hello, World". Let's stick with that tradition. In Swift, this takes just one line:

```
println("Hello, Swift!")
```

This is a complete Swift program, so you don't need to import a separate library to use the function. And you don't need a special entry point, such as the `main` function in Objective-C.

Basic Types

To create values, you use either `let` or `var`. The first keyword, `let`, creates a constant value, which can be assigned only once; `var` creates a variable whose value can change during the execution.

```
var myVariable = 11
myVariable = 33
let someOtherVariable = 22
```

Notice that I didn't give explicit types to these variables. They are implicitly inferred from the type of value they were assigned, integers in this case.

If the type information can't be derived from the initial value, then it must be specified. You do this by adding the type specifier after the variable, separated by a colon (:)

```
var implicitDoubleValue = 1.0
var explicitDoubleValue : Double = 22
var a, b, c : Float
```

Values are never implicitly converted from one type to another type. Every type that needs to convert to anther type must provide a conversion function. Look at the following code:

```
var myString = "The answer is "
let answer = 42
let myAnswer = myString + answer
```

Swift would give an error here. You have to use one of the `String` functions that converts an integer to a string:

```
let myAnswer = myString + String(answer) + "."
```

What this does is create a new string from answer, which is then appended to myString and provides the final answer:

```
println(myAnswer)
```

Another way to insert values into strings is to use the \() expression conversion function To do this, you can write the expression:

```
let myAnswer = "The Answer is \(answer)."
```

The basic types are `String`, `Character`, `Int`, `UInt`, `Float`, and `Double`.

Aggregate Types

You can define arrays and dictionaries using bracket syntax.

```
var myArray : [String]()
var myDictionary : [String : String]()
```

The types within the brackets are the types of values aggregates can hold. Here I define the array to hold only string type values, and, for the dictionary, both the key and the value are of type `string`. But these don't have to be of type `string`; they can be `Int` or other aggragate types.

```
var myFavoriteFruits =["Oranges", "Bananas", "Grapes", "Mangos"]
myFavoriteFruits[2] = "Guavas"
var favorites = ["myFavorites" : myFavoriteFruits]
favorites["MishalsFavorite"] = ["Oranges", "Watermelon", "Grapes"]
favorites["AdamsFavorite"] = ["Apples", "Pears"]
```

Control Flow

You can choose `if` or `switch` for conditionals, and `for-in`, `for`, `while`, and `do-while` for loops. The paretheses around the contidional and loop variables are optional:

```
if a == b or if (a == b)
switch foo or switch (foo)
while a < b or while (a < b)
```

But the braces around the body are required:

```
If a == b
{
println(“they are equal”)
}
```

Functions

The syntax for a function is:

```
func functionName(arguments) -> returnType
{
}
```

or

```
func functionName(arguments)
{
}
```

In the second example, the function doesn't return a value.

> **Note** You also use the keyword func when defining methods for classes.

Functions in Swift are full-fledged types. You can pass them as arguments and return them from functions. You can have a function that takes a function and returns a function. There's a special kind of function called a closure. Closures are unnamed functions that can be passed as data.You write the code for closures between {}:

```
{ (arguments) -> Int in /* body */ }
```

> **Note** In Objective-C the concept equivalent to closures is blocks. When interfacing with Objective-C from Swift, blocks are imported as closures.

Objects

Use the keyword `class` to define class objects, similar to functions:

```
class MyClass {
}
```

Classes in Swift don't require parent classes.

```
class myClass : ParentClass, Protocol, AnotherProtocol
{
}
```

Use enum to create enumeration types

```
Enum : Int
{
  case One
  case Two
  case Three, Four, Five
}
```

The big difference in Swift for enums is they can include methods that operate on the cases of the enum.

Use the `struct` keyword to define structs:

```
struct MyStruct
{
}
```

Structs support most of what classes can do. But the big difference between classes and structs is that when passing structs around the code, they are always copies, while classes are passed by reference.

Generics

Generic types are used when you design a class that can operate on different types of objects, which allows maximum reusability of the code. You can have a linked list of integers or characters or strings. In a language like Objective-C, you'd end up using id or NSObject to hold different types of objects. In Swift, you define your object with a gerenic type in angle brackets <>. Then, anytime you have to define a variable with a method or somewhere in your class, you use the type that was given in angle brackets. Typically, developers use `T` for type, but when you instantiate the class you have to give a proper type, such as `Int` or `Double` or `String`.

```
class Node<T>
{
  var value : T
}

var myNode : Node<Int>
```

In this example `T` is replace with `Int`, and now Node can hold only `Int` type values.

Getting the Sample Code

Xcode is a large application and will take some time to download and install. While you're waiting, you can download the sample code for this book from the Apress site. Go to http://www.apress.com/book/view/9781484203774. In the middle of the page below the book description, you'll see a tab that says `Source Code/Downloads`, where you'll find the download link. Click that link to download the source code to your preferred folder.

Summary

You should have every thing you need now to start playing with Swift or developing your app. Don't forget to download the development tools and set up your development environment.

You've gotten just a quick overview of the Swift language so far. Next, we are going to jump right in and start to play with Swift itself. By the end of this book, you'l be writing programs yourself.

Chapter 2

The Swift Playground in Xcode 6

The Swift playground is a new interactive environment in which developers can view and manipulate their code live, instead of having to continually go throught the complete compile-run-test cycle. You type your code, it's evaluated, and you get feedback right away. You can see immediately whether your code is behaving as expected. Think of it as a mini project with one file and an SDK to compile against.

This chapter will walk you through creating your first playground and show you how to interact with the playground. You'll also create your first simple program in Swift. You'll be using playgrounds throughout this book, whenever you need to try out some standalone code.

I'll also delve into the different parts of the playground and discuss the functionality they provide, and show how to become really good at using playgounds.

Getting Started with a Playgorund

When you launch Xcode, you'll be greeted with its welcome screen, where you can create a new playground. If that window isn't visible, you can create a new playground by selecting **File ➤ New ➤ Playround**... or use **Option-Shift-Command-N** to get the dialog shown in Figure 2-1.

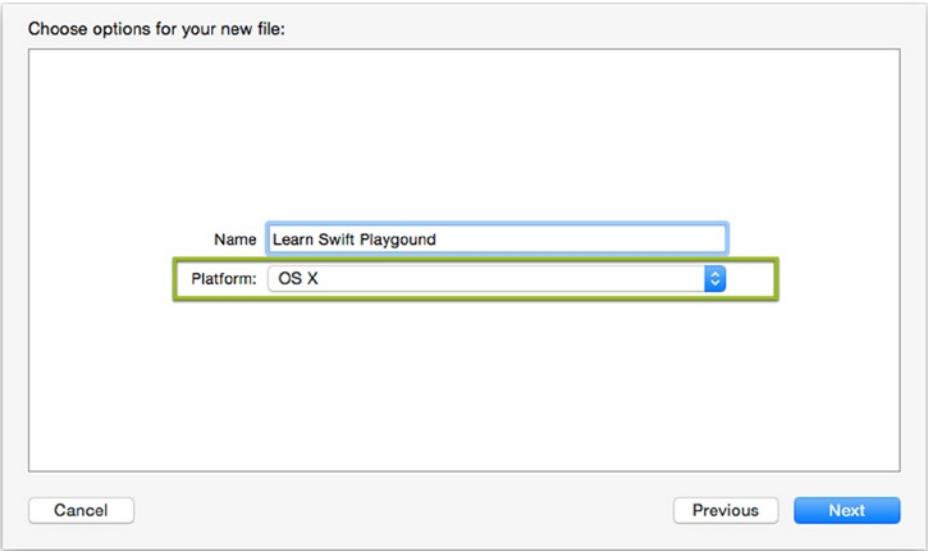

Figure 2-1. *Naming your playground*

Start by creating a new OS X playground, then name your playground and press Next. You'll be greeted by the window shown in Figure 2-2.

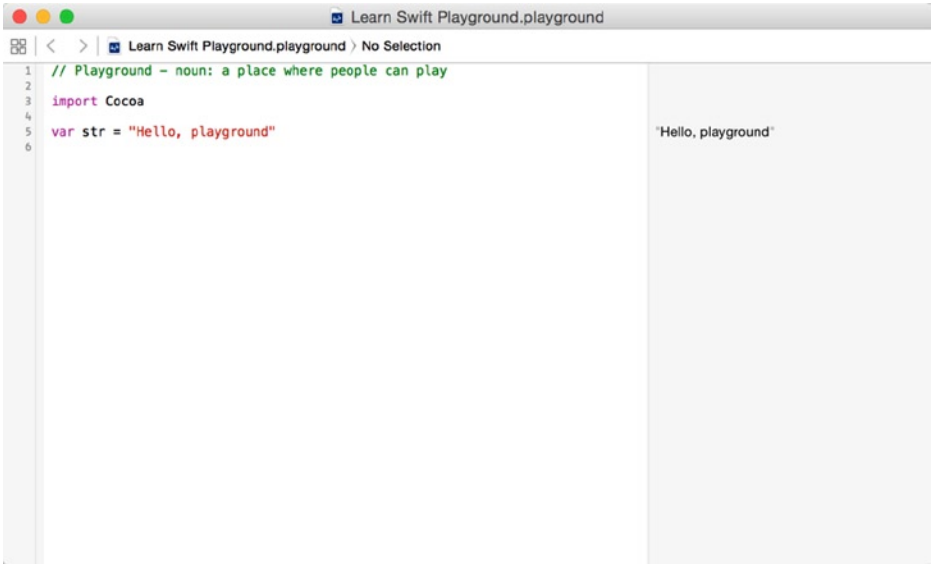

Figure 2-2. *Playgound interaction window*

There are two parts to the playground, the editor on the left and what's called the sidebar on the right. The editor area is where you enter code into the playground for Swift to interpret. The sidebar is where Swift will show the results and respond to your command if it has something to let you know. As you can see, there's already some code in the sidebar.

Line 1 is a single-line comment, as discussed in Chapter 1. Line 3 tells the compiler to import the entire desktop Cocoa API so it will be available for use.

Notice there's no semicolon (;) at the end of the `import` statement. This is because, in Swift, using semicolons to terminate a statement is optional if you only have one statement on a line. If you put more than one statement on a line, you must separate each statement with semicolons, except for the last statement. You can do this: `var a = 4; var b = 8`.

Line 5 defines a variable and then assigns a value to it. The default name is `str`, but it could be any valid string, (as I'll discuss in a later chapter).

You'll note that `I` didn't give the type of variable, such as `Int` or `Float`. That's because of type inference—Swift infers the type of the variable from the initial value assigned to it. Once the initial type is set for a variable, the type can't be changed. So, if you try to set an integer value to `str`, the compiler will give you an error, as line 6 in Figure 2-3 shows.

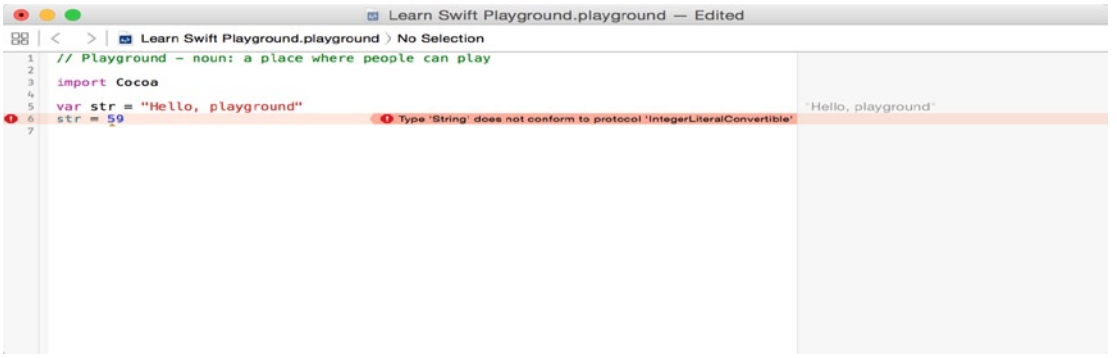

Figure 2-3. *Error when the wrong type is assigned*

This error says that the `String` type doesn't have a method that can take an integer argument and return a string when assigning to `str`.

Let's create another variable called `value` without assigning a value. The compiler is not happy, Because Swift is strongly typed, the language enforces the types for variables at compile time. If you make an error as you're writing code, you'll see the stop icon to the left of the line numbers. If you need to see the error message, click on the icon to display the error, as shown in Figure 2-4.

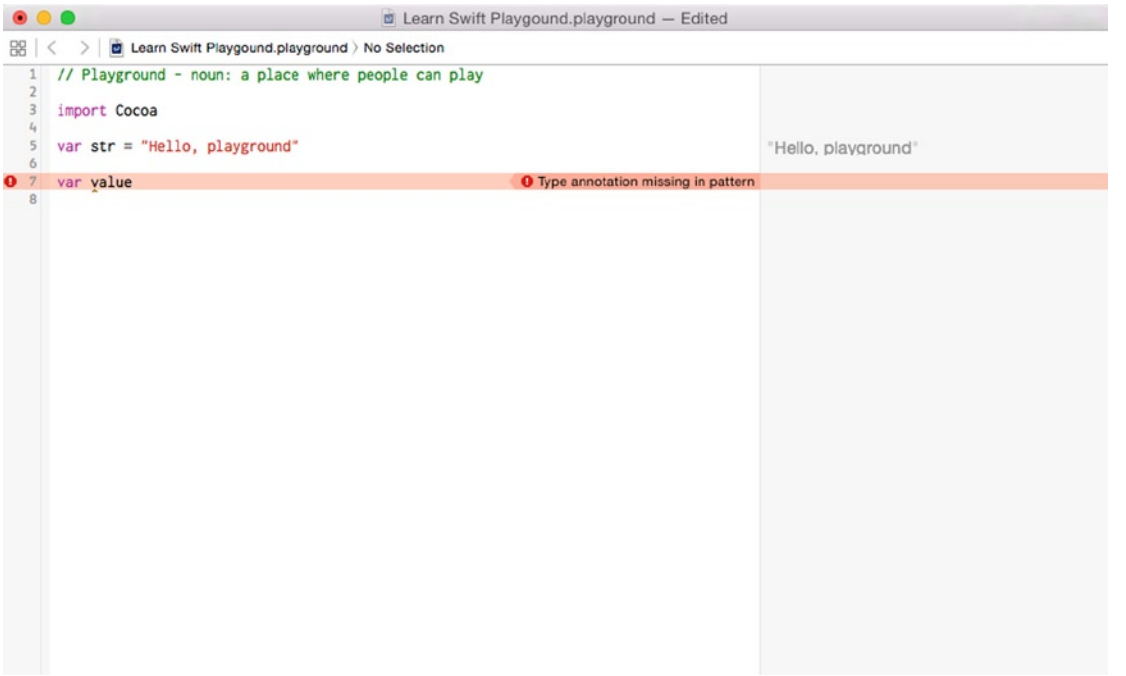

Figure 2-4. A missing-type error

To give a type to a variable in Swift, you add the type after the variable name, separated by a colon. If you write `var value : String`, the compiler is happy. However, if you try to use this variable in an expression, the compiler will again complain, because the variable has not yet been initialized, as Figure 2-5 shows. This is one of the Swift's safety requirements.

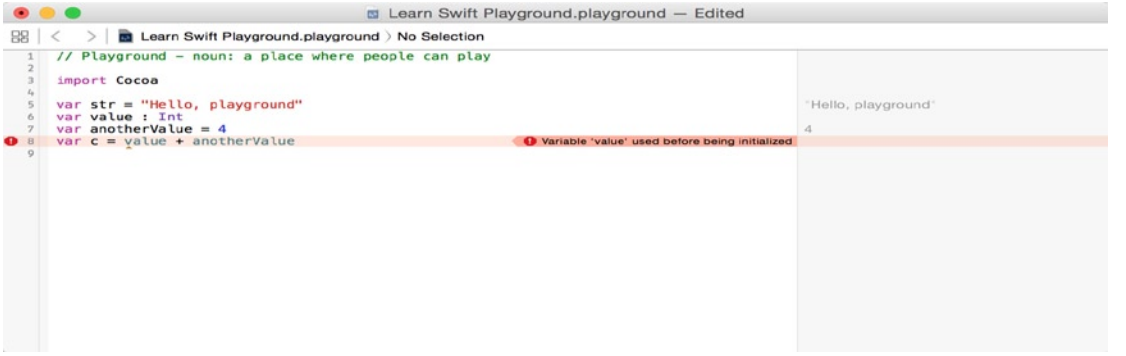

Figure 2-5. Uninitialized variable error

If you look at the right-hand side of the window—the sidebar—you'll see the result of the statement on line 5: the value of the variable `str` was set to the string "Hello, playground".

If you hover over the value on line 5, you'll see two icons on the right side of the line. The first, which looks like an eye, is QuickLook. Clicking it pops up a view of the object. The built-in types are supported and you can customize it for your own types. This is handy if the value is too long or complex to be displayed fully on the right side. The icon that looks like a circle brings up the value history display in the Assistant Editor.

To give this a try, type the following in the editor:

```
for i in 1...10 {
    i * i
}
```

Then select the value history and, as Figure 2-6 shows, the Assistant Editor will display a visual representation of the loop.

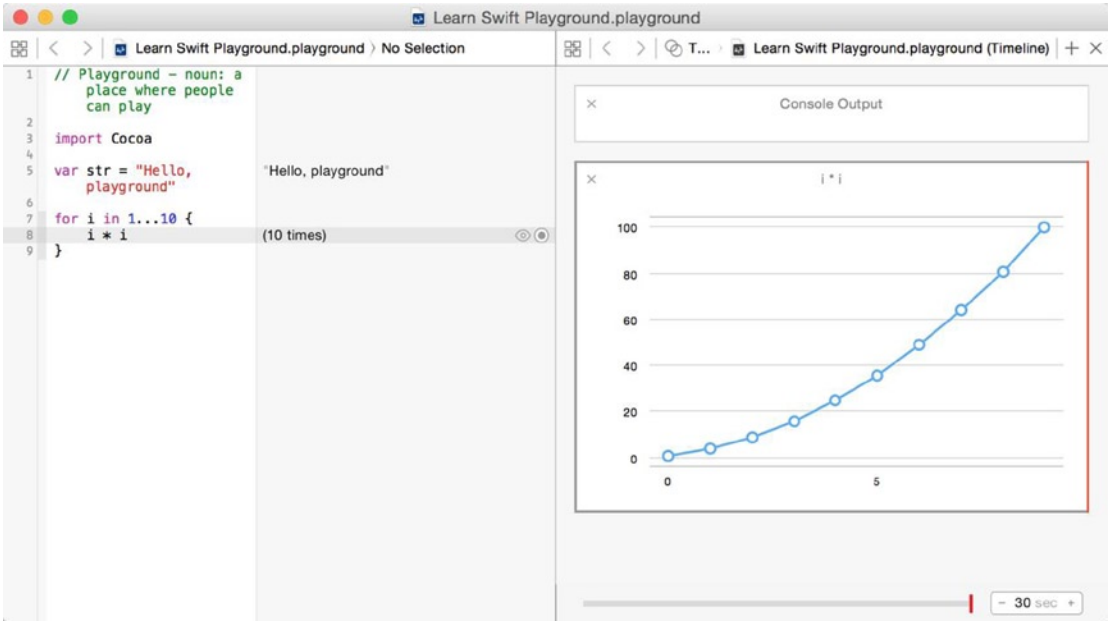

Figure 2-6. Assistant history view

The Assistant Editor uses QuickLook to visualize the output. The supported types are:

- Objects (Structs and Classes)
- Strings
- Images
- Colors
- Views
- Array and Dictionaries
- Points, Rects and Sizes
- Bezier Paths
- URLs (using WebView)

You can use use the `debugQuickLookObject` function to display objects that are derived from NSObject, which encompasses pretty much any existing Cocoa framework. Examples include UIImage, NSImage, UIColor, NSColor, UIView, NSView, and more.

Custom QuickLook Plugins

Because you're not yet familiar with the language, you might find the following examples a bit terse. Still, they will give you an idea how powerful playgrounds can be.

To develop a custom plugin, you use the XCPlayground module, which has three functions that let you display custom values and views in the Assistant Editor.

> **Note** XCPlayground only works for NSView- and UIView-based subviews.

XCShowView

If you're developing a custom view and want to see how it's going, you can call this method to display what the view looks like. `XCShowView` takes an identifier that's displayed at the top of the view so you know which view is being displayed.

```
XCShowView(identifier : String, view : NSView)
XCShowView(identifier : String, view : UIView)
```

XCCaptureValue

If you're developing your program and want to display some values, you can use the `XCCaptureValue` function to display the values:

```
XCCaptureValue<T>(identifier : String, value : T)
```

XCPSetExecutionShouldContinueIndefinitely

Client/server communication is common in mobile computing, but the network is inherently unreliable. As a result, most communication is therefore asynchronus, where the client makes a call to server and when the server returns the response, the client then acts on it. But the call gets executed quickly and runs on the background thread. If the tasks on the main thread finish too quickly, the program could exit without giving the client a chance to process the response from the server. To prevent this, use the `XCPSetExecutionShouldContinueIndefinitely` function to keep the main program from terminating.

This function allows you to execute long-running asynchronous tasks. Here's how you could use it to download some JSON data from the network:

```
XCPSetExecutionShouldContinueIndefinitely(continueIndefinitely: Bool = default)
```

Custom Modules for Playground

These functions are all good, but suppose you have your own code and don't want to copy and paste it into the playground, you just want to use your own classes in the playground. This is possible, but there are some prerequisites:

- The code must be in a framework.
- The classes you plan to use in the playground must have public access.
- The playground must be in the same workspace as the project with the framework.
- The framework must already be built.
- For iOS, the framework must be built for a 64-bit runtime.
- There must be a scheme that builds a target.
- Build location preference must be set to Legacy.
- The playground name must not be same as the build target.

Importing Your Code

Once you've fulfilled these conditions, you can simply use `import ModuleName` to import your code into the playground.

Now let's start create a framework. Launch Xcode and select the **Create a new Xcode project** option, as shown in Figure 2-7.

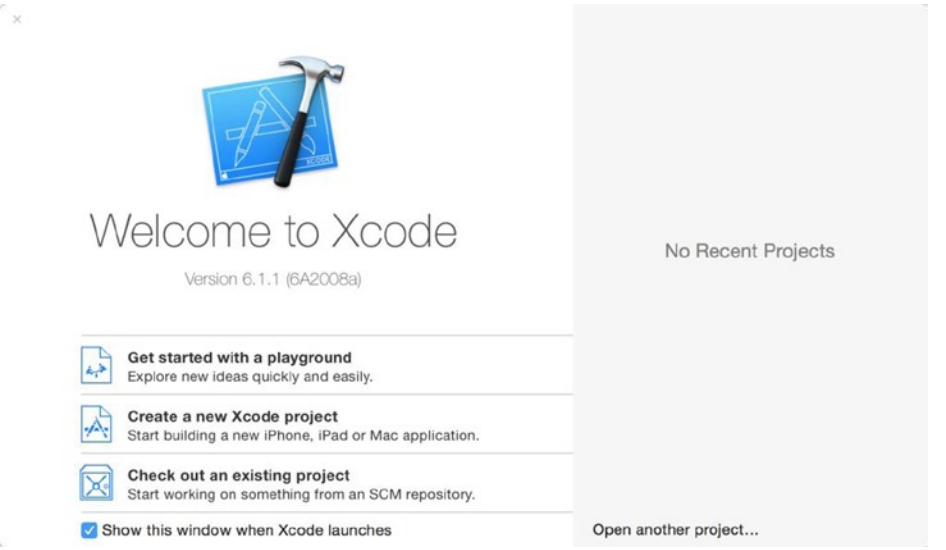

Figure 2-7. Xcode Welcome window

On the next screen you'll be asked to select the template for the project, as shown in Figure 2-8. Under iOS, select **Framework & Library**, and then choose **Cocoa Touch Framework**.