SWIFT for the Really Impatient

Matt Henderson
Dave Wood

Foreword by Jeff LaMarche

Swift for the Really Impatient

This page intentionally left blank

Swift for the Really Impatient

Matt Henderson
Dave Wood

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2014952492

ISBN-13: 978-0-13-396012-9
ISBN-10: 0-13-396012-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing: December 2014

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Tom Cirtin

Managing Editor
Kristy Hart

Senior Project Editor
Betsy Gratner

Copy Editor
Kitty Wilson

Indexer
Tim Wright

Proofreader
Leslie Joseph

Technical Reviewers
Rene Cacheaux
Ash Furrow

Editorial Assistant
Olivia Basegio

Cover Designer
Alan Clements

Senior Compositor
Gloria Schurick

To my mother and my father.

And to Clare.

—Matt Henderson

To my wife, Sabrina, who puts up with the crazy hours I put in and always makes sure I at least take a break for dinner.

—Dave Wood

This page intentionally left blank

Contents

Foreword

In March, 2008, Apple released an SDK for its wildly successful iPhone. That SDK, and the App Store that would go live several months later, attracted a lot of attention. A lot of people coming to Apple's platform for the first time were surprised to find that the entire toolset was based around an obscure 25-year-old programming language called Objective-C rather than a more widely used language like C++ or Java.

Mac developers and those who followed Apple weren't particularly surprised. Objective-C had been the primary language used to develop Mac apps since shortly after the NeXT acquisition in 1996. Unlike C++ or Java, which are general-purpose languages used on numerous platforms and for a wide variety of programming tasks, Apple chose to build first OS X and later iOS around a language that it could control. The language grew slowly and with a very singular focus on the development of GUI apps for Apple's operating systems.

In many ways, Objective-C and the NeXT frameworks used to build applications were years ahead of their time. Though never particularly successful as a commercial venture, NeXT's application-building tools were incredibly popular with those who used it because it enabled developers to build applications much faster than other tools on the market at the time.

But Objective-C is more than 30 years old now, and that's a very long time in terms of technology. While Objective-C has changed and evolved some, it really hasn't kept pace. Programming languages and compilers have evolved a lot in the last few decades, and many people have been clamoring for a more "modern" programming language for developing iOS and OS X apps.

At WWDC 2014, Apple surprised nearly everyone by announcing that it had a new modern language called Swift.

Developed in secret over the course of four years, Swift is a very different language. It has many interesting features that Objective-C lacks but uses the same run time as Objective-C and is able to use all the existing frameworks and libraries that make up the iOS and OS X SDKs.

But Swift really is different. It looks different. It feels different. You kind of even have to "think different" to use it well.

Fortunately, you've picked up just the right book to help you think different. Matt and Dave have done a great job taking you through this interesting but maybe just a little scary new language. They walk you through the how and the why and will help you steer clear of the many gotchas waiting for you as you get up to speed.

—Jeff LaMarche, author, *Beginning iPhone Development: Exploring the iPhone SDK* (Apress); managing partner and founder, MartianCraft

Preface

This book gives a concise introduction to Apple's new programming language, Swift. We've created this book for developers who are currently writing apps using Objective-C and for developers who are looking to start writing apps and are curious about what Swift offers.

This book is written in the "impatient" style and tries to mimic Cay Horstmann's style of presenting topics in a quick and clear manner that gives developers enough information to be immediately productive. The code in this book is primarily presented in short chunks designed to illustrate concepts. These code snippets are designed to act as a quick reference and do not provide a cookbook of complete examples that can be used directly.

Swift provides many exciting language features that aren't present in Objective-C, the current primary language for developing for Apple's platforms. These features are designed to make developers more productive and to make their code less prone to errors. With Swift, developers are able to make use of a strong type system and modern syntax to easily create powerful applications. Chapter 1, "Introducing Swift," and Chapter 2, "Diving Deeper into Swift's Syntax," give a rapid introduction to Swift's syntax, basic types, and features that might be new concepts for Objective-C developers. Later chapters focus more specifically on these features.

Like Objective-C, Swift is primarily an object-oriented language. Chapter 3, "Objects and Classes," introduces the major object types that are used heavily in any Swift app. Using classes in Swift will be familiar to developers who come from different object-oriented languages. In addition, Swift's type system helps prevent errors that are common with other languages.

Starting with Chapter 4, "Optionals," this book goes into slightly more depth on topics that might be less familiar to developers coming from Objective-C. Chapter 4 focuses on optional types, a new fundamental concept that forces objects to declare at the type level whether they might contain a missing or nil value. This chapter begins to explore Swift's focus on code safety. Chapter 5, "Generics," explores the concept of generics, which let developers abstract functionality for reusable code while still maintaining type safety. In Swift, functions are a first class type, which allows them to be passed around as parameters and returned as values. Chapter 6, “Functions and Closures,” explores the functional programming aspects of Swift.

The main reason to learn Swift is to be able to craft apps for iOS and Mac OS X. Chapter 7, “Working with Objective-C,” focuses on how to use Swift with Objective-C and Apple’s existing frameworks. Swift was designed to easily integrate with existing Objective-C projects and the frameworks Apple provides for making apps, but you need to know a few important things before you try to combine Swift with C and Objective-C, and that’s what Chapter 7 covers.

By the time you reach Chapter 8, “Common Patterns,” you’ll have a firm understanding of how to make apps using Swift and the advantages of leveraging the new features of Swift, so Chapter 8 focuses on practical use cases. Once you finish Chapter 8, you’ll know how to deal with several situations that commonly come up while developing apps.

At the end of each chapter, we’ve included exercises that will help you develop your newly acquired skills. We encourage you to do the exercises to reinforce what you’ve read. If you get stuck on an exercise, please visit our website at http://SwiftForTheReallyImpatient.com, where we post solutions. You’ll also find book errata and other interesting content at this site.

NOTE

Throughout this book where a line of code is too long for the printed page, a code-continuation arrow (➥) has been used to mark the continued line of code.

Acknowledgments

Our thanks go to our editors, Trina MacDonald and Betsy Gratner, for guiding two unpublished authors through all the steps required to make this book happen. Thanks to our reviewers, Rene Cacheaux, Tom Cirtin, Ash Furrow, Kitty Wilson, and Chris Zahn, who made all the chapters read easier and made sure we kept up to date with all the Swift betas along the way. Thanks to our colleagues at MartianCraft for always offering to help, no matter how obscure the development problem, especially Jeff LaMarche for writing us a wonderful foreword and Kyle Richter for encouraging us to write this book. Finally, we'd like to thank our wives, Clare and Sabrina, for supporting us through the not entirely easy process of creating our first book.

About the Authors

Matt Henderson has been developing for Apple's platforms since 2009 and is currently a Cocoa engineer at MartianCraft. He's given several presentations at various user groups and conferences, including 360iDev, Cocoaheads Denver, and Boulder iOS Meetup. He realized he might have a future in software when he discovered it was easier for him to program his graphing calculator to solve equations than it was to study for his math tests. He thinks that the best debugging technique is taking a walk outside in the sun or snow.

Dave Wood has been developing for iOS since 2008 and OS X since 2009. He began writing code at age 9 on a TI/99/4A and instantly fell in love. He has worked on various types of projects, including systems that interface with stock exchanges, news outlets, and banking systems, as well as newspaper websites and, of course, mobile apps ranging from games, social networks, financial apps, and productivity and developer apps. When possible, he enjoys whitewater kayaking and scuba diving. Currently he runs his own development studio, Cerebral Gardens, and freelances as a Cocoa engineer for MartianCraft.

◆ ◆ ◆ CHAPTER 1

Introducing Swift

Topics in This Chapter

- 1.1 Basic Syntax
- 1.2 Basic Data Types
- Exercises

Swift is a new programming language developed by Apple that was released to developers at WWDC 2014. Swift can be used to develop applications for Apple's iOS and OS X platforms. It is designed to work with, but eventually replace, Objective-C, the language originally used for developing applications on Apple's platforms.

Apple has a number of goals for Swift:

- Make app development easier, with modern language features.
- Produce safer code by preventing the most common programming errors.
- Create easy-to-read code with clear and expressive syntax.
- Be compatible with existing Objective-C frameworks, including the Cocoa and Cocoa Touch frameworks.

This chapter introduces the basic syntax of Swift and lays the foundation you'll need for the rest of the book.

These are the key points in this chapter:

- You use `var` to declare a variable and `let` to declare a constant.
- You execute code conditionally with `if` or `switch` constructs.
- You repeat code segments by looping with `for`, `for-in`, `while`, and `do-while` loop constructs.

- The basic data types are implemented as `struct`s, which are passed by value in code.
- Since the basic types are `struct`s, they may have additional properties or methods available.
- Arrays and dictionaries in Swift are more powerful collection types than their Objective-C counterparts.

1.1 Basic Syntax

When you learn a new language, the first complete program you're likely to see is the famous "Hello World" example. In Swift, this program consists of just one line:

```
println("Hello World")
```

The first thing you should notice here is what you don't see. The code jumps right into the guts of the program. You don't need to set anything up to get started, include or import a standard library, set up an initial `main()` function to be called by the system, or even include a semicolon at the end of each line.

NOTE

Comments in Swift are the same as in Objective-C, with one powerful addition. You can use `//` for a single-line comment, or you can use `/* */` to surround a multiline comment. Unlike C-based languages, Swift allows you to have nested comments. This is very handy when you want to comment out a whole section of code that may already have multiline comments included. Throughout this book, we use `//` comments in the examples to show results and add explanations.

1.1.1 Variables and Constants

A program that only prints a static line of text isn't very useful. For a program to be useful, it needs to be able to work with data, using variables and constants. As their names imply, variables have contents that may change throughout the execution of the code, while constants cannot be changed after they're initialized. Variables are said to be *mutable*; constants are *immutable*.

In Swift, you declare a variable by using the `var` keyword, and you declare a constant by using the `let` keyword. This applies for all data types in Swift and is different from Objective-C, where the type itself indicates whether it is mutable or not, such as `NSArray` versus `NSMutableArray`. With Swift, the mutable version of an object is the same type as the immutable version—not a subclass.

For the rest of this chapter, what we say about variables applies equally to constants (provided that we're not talking about mutating data).

NOTE

Before we look at specific types, it's important to understand that all data types in Swift are implemented as one of three different kinds of data structures. Each type is either an `enum`, a `struct`, or a `class` and thus may have properties and/or methods available. We cover these in much greater detail throughout the book, but a key fundamental you need to be aware of from the start is the difference in how these data structures are passed around in your code.

Swift employs some standard rules with regard to dealing with `enum`s, `struct`s, and `class`es. Whenever an `enum`, or a `struct` is passed somewhere, it is *passed by value*; that is, a copy of the original is created, and that copy is what's assigned to the new variable. This allows the new variable to be used, modified, or deleted without affecting the original. And the reverse is also true: The original can be used, modified, or deleted without affecting the new copy.

However, when a `class` is passed somewhere, it is *passed by reference*; that is, a pointer to the original variable is assigned to the new variable. Changes made to either variable will affect the other one.

All the basic types we're about to cover are implemented under the hood as `struct`s, and so they are always copied and passed by value. Because they are `struct`s, it's also possible for them to implement additional functionality through properties and/or methods that you wouldn't see from their Objective-C counterparts.

Swift is a strongly typed language, which means that every variable is set to a specific type at compile time, and it can only contain values of that type throughout its lifetime.

Two common types are `Int` and `Float`. (We'll get into their details a little later.) If you set a variable to be of type `Int`, it can only ever store `Int` values; it can never be coerced into storing a `Float`. Types can never be implicitly converted into other types. This means, for example, that you cannot add an `Int` to a `Float`. If you need to add two numbers together, you need to make sure they're the same type or explicitly convert one to the other. This is part of what makes Swift a safe language: The compiler prevents you from mixing types and possibly producing unexpected results.

To see the dangers involved in mixing types, consider this C code:

```
int intValue = 0;

float floatValue = 2.5;

int totalValue = intValue + floatValue;
```

This code adds an `int` and a `float` together. What would `total` be equal to here? Since the total is an `int`, it is unable to store the decimal portion of the `floatValue` variable. `floatValue` must first be implicitly converted to an `int` before it can be added to `intValue` and stored in `totalValue`. In this case, is the developer expecting the compiler to round `floatValue` to `3`, or is she expecting it to just drop the decimal portion and instead add `2`? Swift avoids this type of ambiguity by producing a compile-time error here, forcing you to tell it exactly what you want to happen. This is one way Swift avoids common programming errors.

You need to give variables and constants names so that you can refer to them in code. Names in Swift can be composed of most characters, including Unicode characters. While it's possible to use emoji and similar characters as variable names, you should rarely, if ever, actually do it. Here is the minimum code for declaring a variable:

```
var itemCount: Int
```

This code declares a variable named `itemCount` of type `Int`. A variable must be set to an initial value before you can use it. You can do this when the variable is declared, like this:

```
var itemCount: Int = 0
```

or you can do it at some later point, as long as you do it before you attempt to read the value.

Swift has a feature called *type inference*. If the compiler has enough information from the initial value you set to infer the type, you can omit the type of the variable when you declare it. For example, if your variable is going to be an `Int`, you can declare it like this:

```
var itemCount = 0
```

Because `0` is an `Int`, `itemCount` is inferred to be an `Int`. This is exactly the same as the example above. The compiler generates exactly the same machine code.

If the variable's initial value is set to the return value of a function, the compiler will infer the type to be the same as the return value's type.

Given a function `numberOfItems()` that returns an `Int` and the following line:

```
var itemCount = numberOfItems()
```

the compiler will infer `itemCount` to be of type `Int`.

Since the compiler generates exactly the same code whether you explicitly set the type or use type inference to let the compiler set the type for you, there is no advantage or disadvantage to either method at run time. Of course, if you need to explicitly set the type, you have no option. But in cases where the compiler can infer the type, it's up to you whether to let the compiler do so or whether you explicitly set the type anyway. There are two things to consider when making this decision. The first is readability. If, when you use type inference, the type of the variable would still be clear to a future reader of the code, by all means save some keystrokes and use type inference. If the initial value being set is the return value of some uncommon function, it may be clearer to the future reader if you explicitly set the type. When reading the code at a later date, you don't want to have to look up what a function returns just to determine a variable's type.

The second reason you might want to explicitly set a type when it can be inferred is to add an additional safety check. This ensures that the type you're expecting the variable to be and the type being set match. If there's a mismatch, you get a compile-time error and can make the necessary corrections.

1.1.2 String Interpolation

You've already seen how to print a line of text to the console by using the `println` command. You can add variables, constants, and other expressions to the output by using string interpolation. You do so by including variables and expressions directly in the string literal, surrounded by parentheses and escaped with a backslash:

```
var fileCount = 99
println("There are \(fileCount) files in your folder")
//outputs: There are 99 files in your folder
```

This doesn't apply just to `println`. You can use it anywhere a string literal is used:

```
var firstName = "Geoff"
var lastName = "Cawthorne"
var username = "\(firstName)\(lastName)\(arc4random() % 500)"
//username: GeoffCawthorne253
```

1.1.3 Control Flow

All but the simplest of programs require some sort of logic to determine what actions should be taken. Decisions must be made based on the information the program has available. Logic such as “If this, do that” or “Do this *x* many times” determines the flow of an app and, thus, its result.

Conditionals

Swift offers both `if` and `switch` constructs for you to execute code conditionally.

Using `if` is the simpler of the two constructs and closely follows what you’re used to in Objective-C. There are a few differences you need to be aware of, however. The first difference continues Swift’s theme of reducing unnecessary syntax: Swift does not require you to surround test expressions with parentheses, though you may, if you desire. The second difference is that braces are required around the conditional code. Third, the test expression must explicitly result in a `true` or `false` answer; an `Int` variable with a value of `0` is not implicitly evaluated as `false`, nor is a value of anything else implicitly evaluated as `true`.

Here is a minimal example:

```
var daysUntilEvent: Int = calculateDaysUntilEvent()
if daysUntilEvent > 0 {
    println("There is still time to buy a gift")
}
```

You can chain together multiple `if`s with the `else` keyword:

```
var daysUntilEvent: Int = calculateDaysUntilEvent()
if daysUntilEvent > 0 {
    println("There is still time to buy a gift")
}
else if daysUntilEvent < 0 {
    println("You missed it, better pick up a belated card")
}
else {
    println("Better pick up the gift on the way")
}
```

The `switch` construct is an alternative to `if` statements. It is based on what you’ve used in Objective-C, but in Swift, it is much more powerful. There are two important differences you need to consider when using a `switch` in Swift. The first is that every possible option must be covered. A `default` case can be used to accomplish this requirement. The second difference is a major change in how cases are handled. In C-based languages, you need to include a `break` statement at the end of each case, or execution will continue with the next case. This has been the source of many errors over time. To prevent these errors in Swift, the design was changed to automatically break when the next case begins. Some algorithms may require the old behavior, so it is available to you through the use of the `fallthrough` keyword.

Here's a basic example of a `switch` in use:

```
var numberOfItemsInCart: Int = calculateNumberOfItemsInCart()
switch numberOfItemsInCart {
case 0:
    println("Cart is Empty")
case 1:
    println("1 item in cart, standard shipping applies")
default:
    println("\(numberOfItemsInCart) items, you quality for free
➥shipping")
}
```

We'll cover advanced `switch` usage in Chapter 2, "Diving Deeper into Swift's Syntax."

Loops

In Swift, `for`, `for-in`, `while`, and `do-while` are used for looping. These are similar to what you're used to in Objective-C, with only slight differences in the syntax.

Here is a basic `for` example:

```
for var i = 0; i < 10; ++i {
    println("i = \(i)")
}
```

Just as with `if` statements, you can omit the parentheses. In this example, `i` is implicitly declared as an `Int`. The loop will iterate while `i < 10`, and it's incremented by 1 at the end of each iteration.

Another form of the `for` loop is the `for-in` loop. This lets you iterate through each item in a collection, such as an array or a range.

Swift has two new range operators for creating ranges that can be used with `for-in` loops. The `..<` operator is the half-open range operator; it includes the value on the left side but not the value on the right side. Here's an example that iterates 10 times, with `i` starting as `0` and ending as `9`:

```
for i in 0 ..< 10 {
    println("i = \(i)")
}
```

The `...` operator is the inclusive range operator. It includes the values on both sides. This example iterates 10 times, with `i` starting as `1` and ending as `10`:

```
for i in 1 ... 10 {
    println("i = \(i)")
}
```

When you use a `for-in` loop to iterate through a collection such as an `Array`, it looks like this:

```
var itemIds: [Int] = generateItemIds()
for itemId in itemIds {
    println("itemId: \(itemId)")
}
```

The `while` loop iterates for as long as the test condition is `true`. If the test condition is `false` at the start, the loop doesn't iterate at all, and it is just skipped entirely. For instance, this example will never actually print `100% complete` since the test condition becomes `false` once `percentComplete == 100`:

```
var percentComplete: Float = calculatePercentComplete()
while percentComplete < 100 {
    println("\(percentComplete)% complete")
    percentComplete = calculatePercentComplete()
}
```

If you change this to a `do-while` loop, the test condition is evaluated at the end of the loop. This guarantees that the loop will iterate at least once and also means it will iterate a final time when the test condition fails (which could be the first iteration). This version updates the display one final time once the task being monitored is complete:

```
var percentComplete: Float = 0.0
do {
    percentComplete = calculatePercentComplete()
    println("\(percentComplete)% complete")
} while percentComplete < 100
```

When using a loop, there are times when you need to adjust the iterations by either quitting iteration altogether or skipping a single iteration. Just as in Objective-C, there are two keywords you can use for these purposes: `break` and `continue`. You use `break` to immediately jump out of the loop and cancel any further iterations:

```
var percentComplete: Float = 0.0
do {
    percentComplete = calculatePercentComplete()
    if taskCancelled() {
        println("cancelled")
        break
    }
    println("\(percentComplete)% complete")
} while percentComplete < 100
```

You use `continue` to end the current iteration and immediately start the next one:

```
var filesToDownload: [SomeFileClass] = filesNeeded()
for file in filesToDownload {
    if file.alreadyDownloaded {
        continue
    }

    file.download()
}
```

With nested loops, `break` and `continue` affect only the inner loop. Swift has a powerful feature that Objective-C does not have: You can add labels to your loops and then specify which loop you would like to `break` or `continue` out of. A label consists of the name followed by a colon in front of the loop keyword:

```
var folders: [SomeFolderClass] = foldersToProcess()
outer: for folder in folders {
    inner: for file in folder.files {
        if shouldCancel() {
                break outer
            }

            file.process()
        }
    }
```

1.2 Basic Data Types

Swift has a standard set of basic data types for storing numbers, strings, and Boolean values.

By convention, types in Swift are named using camel case notation. Unlike in Objective-C, there is no prefix (NS, CG, etc.) on the standard type names.

1.2.1 `Int`

For storing integer values, the basic type is `Int`. It is 32 bits deep on 32-bit devices and 64 bits deep on 64-bit devices.

You can access the minimum and maximum values the type can store by using the `min` and `max` static properties:

```
println("\(Int.min)")
//output on 32-bit device: -2147483648
//output on 64-bit device: -9223372036854775808

println("\(Int.max)")
//output on 32-bit device: 2147483647
//output on 64-bit device: 9223372036854775807
```

When you need an integer with a specific bit depth, you use `Int8`, `Int16`, `Int32`, or `Int64`.

There are also unsigned variants of the `Int` types. You can prefix the `Int` type name with a `U` to get the corresponding unsigned version: `UInt8`, `UInt16`, `UInt32`, or `UInt64`.

Because Swift is a strongly typed language, you can't mix and match these `Int` types haphazardly. You cannot even do basic math or comparisons with mixed types. In Objective-C it's common to see `NSUInteger` values assigned to or compared with an `NSInteger`, with little regard for a possible overflow. This is especially common when using the `count` property on an `NSArray` variable:

```
for (NSInteger i = 0; i < [someNSArray count]; ++i) {
    NSLog(@"%@", someNSArray[i]);
}
```

Since `NSArray`'s `count` method actually returns an `NSUInteger` value, this example compares two different types. It even passes in the wrong type to the array's subscript. This is a bug just waiting to go BOOM!—most likely after you've shipped the app, and a user has more data than you imagined or tested with, thus hitting an overflow.

This sort of bug just can't happen with Swift. The compiler won't let you mix unsigned and signed values, and it won't let you mix variables with different bit depths. Nor will it let you assign one type to another. For this reason, Apple recommends that you always use the `Int` type unless you specifically need a certain bit depth or have to use an unsigned value (perhaps for really large numbers). This helps you avoid having to convert one `Int` type to another. Apple has modified the Cocoa classes to follow this guideline. As mentioned earlier in this chapter, in Objective-C, `NSArray`'s `count` property returns an `NSUInteger` (unsigned), but in Swift it returns an `Int` (signed), even though it can never be negative.

In cases in which you need to convert from one type to another, you can do so by creating a new instance of the destination type, using the original value as its initial value:

```
var a32BitInt: Int32 = 10
var a64BitInt: Int64 = Int64(a32BitInt)
//a64BitInt: 10 (in a 64-bit variable)
```

This works by creating a new `Int64` with an initial value of `a32BitInt`.

Be careful, however, because this can create overflow situations. The compiler will catch obvious overflows for you, but it cannot catch all instances, like this:

```
var a64BitInt: Int64 = Int64.max
var a32BitInt: Int32 = Int32(a64BitInt)
//error: a32BitInt overflows
```

CAUTION

To ensure compatibility when transferring files between devices, any integer variables that you're writing to a file (or transmitting across a network) should explicitly specify the bit depth. You should force the use of either 32- or 64-bit variables to avoid possible type mismatches and/or corruption of the data when reading in the saved values. If you store an `Int` on a 32-bit device and then read it in on a 64-bit device, bad things can happen. Even if your app doesn't allow for transferring of files, a user can still restore settings originally stored on a 32-bit device to a new 64-bit device and then encounter the same problem. Use `Int32`, `UInt32`, `Int64`, or `UInt64` for values you need to save.

1.2.2 `Double` and `Float`

When you need to work with decimal numbers in Swift, you can work with `Float` and `Double`. `Float` is always a 32-bit value, while `Double` is always a 64-bit value, regardless of the device architecture.

When using decimal literal values, the compiler always infers a `Double` type and not a `Float`. Therefore, if you don't need the precision of a 64-bit value, you should explicitly declare the variable as a `Float`, like this:

```
var distance = 0.0
//distance is a Double
var seconds: Float = 0.0
//seconds is a Float
```

The following examples show some useful properties. These examples use a `Float`, but they would work just the same with a `Double`:

```
var someFloat = Float.NaN
if someFloat.isNaN {
    println("someFloat is not a number")
}

someFloat = Float.infinity
if someFloat.isInfinite {
    println("someFloat is equal to infinity")
}
```

```
someFloat = -Float.infinity
if someFloat.isInfinite {
    println("someFloat is equal to infinity,")
    println("even though it's really negative infinity")
}
if someFloat.isInfinite && someFloat.isSignMinus {
    println("someFloat is equal to negative infinity")
}

someFloat = 0/0
if someFloat.isNaN {
    println("someFloat is not a number")
    println("note, we divided by zero and did not crash!")
}
```

1.2.3 `Bool`

The `Bool` type stores Boolean values and is very similar to what you're used to in Objective-C. However, Swift uses `true` and `false` rather than Objective-C's `YES` and `NO`.

In Objective-C, pretty much anything can be converted to a Boolean. If it is something, it's treated as `YES`, and if it is nothing (e.g., `nil`), it's `NO`. Here's an example:

```
NSInteger someInteger = 0;
BOOL hasSome = someInteger;
//hasSome: NO
someInteger = 100;
hasSome = someInteger;
//hasSome: YES
NSObject* someObject = nil;
BOOL isValidObject = someObject;
//isValidObject: NO
```

This is not the case with Swift. With Swift, only expressions that explicitly return a `Bool` may be used to define a `Bool` value. You can't implicitly compare values to `0` or `nil`. Here's an example:

```
var someInteger = 0
var hasSome:Bool = (someInteger != 0)
//hasSome: false
```

```
someInteger = 100
hasSome = (someInteger != 0)
//hasSome = true
```

1.2.4 Strings

Strings in Swift are very different from strings in Objective-C. In Swift, `String` literals are simply text surrounded by double quotes:

```
var greetingString = "Hello World"
```

A `String` is implemented as a collection of `Characters`. Each `Character` represents a Unicode character, one of more than 110,000 characters and symbols from more than 100 scripts. `Characters` are implemented with one of several character encoding methods, such as UTF-8 or UTF-16. These encoding methods use a variable number of bytes in memory to store each character. Because characters vary in size, you cannot determine the length of a string by looking at its size in memory, as you can in Objective-C. Instead, you must use the `countElements()` function to determine how many characters are in a `String`. `countElements` iterates through the string and looks at each character to determine the count. While Swift's `String` is compatible with Objective-C's `NSString`, and you can use `String` wherever `NSString` is called for, the implementations are different, and thus the element count may not be the same value you would get from the `NSString length` property. This is because `length` returns the number of 16-bit code units in the UTF-16 version of the `NSString`, and some Unicode characters use more than 1. You can use the `utf16Count` property of a `String` to access the `NSStrings length` equivalent:

```
var myPuppy = "Harlow looks just like this: 🐶"
println("\(countElements(myPuppy))")
//output: 30
println("\(myPuppy.utf16Count)")
//output: 31, 🐶 uses 2 16-bit code units
```

You can concatenate `Strings` together by using the + operator:

```
var firstName = "Sabrina"
var lastName = "Wood"
var displayName = firstName + " " + lastName
//displayName: Sabrina Wood
```

You can also append one string to another by using the `+=` operator:

```
var name = "Katelyn"
name += " Millington"
//name: Katelyn Millington
```

Since a `String` is a collection of `Character`s, you can iterate through them by using a `for-in` loop:

```
var originalMessage = "Secret Message"
var unbreakableCode = ""
for character in originalMessage {
    unbreakableCode = String(character) + unbreakableCode
}
//unbreakableCode: egasseM terceS
```

Notice that you cannot concatenate a `Character` and a `String` together. You must create a new `String` that contains the character and concatenate that to the `unbreakableCode` variable.

The syntax for comparing strings is also much improved in Swift over Objective-C. For example, compare the following Objective-C code:

```
NSString* enteredPasswordHash = @"someSaltedHash";
NSString* storedPasswordHash = @"someSaltedHash";
BOOL accessGranted = [enteredPasswordHash isEqualToString:
➥storedPasswordHash];
//accessGranted: YES
```

to this Swift code:

```
var enteredPasswordHash = "someSaltedHash"
var storedPasswordHash = "someSaltedHash"
var accessGranted = (enteredPasswordHash == storedPasswordHash)
//accessGranted: true
```

1.2.5 Arrays

Arrays are one of the collection types offered in Swift. An array is an ordered list of items of the same type. In Swift, when you declare an array, you must specify what type it will contain. Once you've done that, it can contain only that type. This ensures that when you pull an item out of the array, you're guaranteed to have the type you expect.

To create an array literal, you surround a list of elements with square brackets, like this:

```
var dogs = ["Harlow", "Cliff", "Rusty", "Mia", "Bailey"]
```

Be sure that all elements are of the same type, or you will receive a compile-time error.

There are two ways to indicate the type of an array: the long form and the short form. These two ways are equivalent and can be used interchangeably:

- Long form: `Array<ValueType>`
- Short form: `[ValueType]`

The syntax to declare and initialize an array using the short form is:

```
var people: [String] = [] //explicit type
//or, alternately
var people = [String]() //implicit type
```

This example declares an array variable called `people`, which will contain `String` values, and you initialize it to an empty array.

You can use type inference to let the compiler determine the types of objects in the array, provided that you give enough information when you declare it:

```
let bosses = ["Jeff", "Kyle", "Marcus", "Rob", "Sabrina"]
```

Because you're initializing the array with strings, the compiler infers that `bosses` is an array of type `[String]`.

NOTE

When you create an immutable array in Swift, you cannot add, change, or remove any items in that array. You can, however, change properties on the elements of the array.

Given an array variable, there are several key methods you can use to access or modify the contents:

```
var primaryIds: [Int] = [1, 2, 3]
//primaryIds: [1, 2, 3]
println(primaryIds.count)
//output: 3
primaryIds.append(4)
//primaryIds: [1, 2, 3, 4]
```

```
primaryIds.insert(5, atIndex:0)
//primaryIds: [5, 1, 2, 3, 4]
primaryIds.removeAtIndex(1)
//primaryIds: [5, 2, 3, 4]
primaryIds.removeLast()
//primaryIds: [5, 2, 3]
primaryIds.removeAll()
//primaryIds: []
println(primaryIds.isEmpty)
//output: true
```

You can also use subscripting to access a specific element or range of elements:

```
var primaryIds: [Int] = [1, 2, 3]
//primaryIds: [1, 2, 3]
println(primaryIds[2])
//output: 3 (arrays are zero based)
primaryIds[2] = 12
//primaryIds: [1, 2, 12]
primaryIds[0...1] = [10]
//primaryIds: [10, 12]
//notice the [] surrounding the 10
println(primaryIds[3])
//error: 3 is beyond the bounds (0...2) of the array
```

Make sure you don't attempt to access an element that is beyond the bounds of the array, though, or you'll encounter a run-time error, and your app will crash.

There are some important differences between the Objective-C `NSArray`s that you're used to and arrays in Swift. In Objective-C, you can only store objects that are of type `NSObject` (or a subclass) in an `NSArray`. This is why classes such as `NSNumber` exist: They're object wrappers around basic types so you can use them in collections. You don't add `3` to an `NSArray`; you add an `NSNumber` with a value set to `3` to the array. In Swift, you can add `struct`s, `enum`s, or `class`es to an array, and because all the base types are implemented as `struct`s, they can all be easily added directly to an array, including literals such as `3`. What happens when they are added to the array, however, differs depending on the type that is added. Recall the rules we discussed earlier, about how Swift passes `struct`s compared to how it passes `class`es. These rules come into play when you're adding objects to an array. If you add an `enum` or a `struct` to an array, you add a copy, not a reference to the original object. If you add a `class`, however, you add a reference to the object. The same rules apply when you pass an array. The array itself is copied because it is a `struct`, and then each element is either copied or referenced, depending on whether it is an `enum`, a `struct`, or a `class`.

This means you can alter what elements are in each array independently, without affecting another array. If the elements are `enum`s or `struct`s, you can also alter them independently. If the elements are `class`es, changing one element will have an effect on the same element in the other array (as well as that object if it exists outside the array).

Here you can see these concepts in action:

```
var coordA = CGPoint(x: 1, y: 1)
var coordB = CGPoint(x: 2, y: 2)
var coords = [coordA, coordB]
//coordA/B are copied into the coords array
//coords: [{x 1 y 1}, {x 2 y 2}]
var copyOfCoords = coords
//copyOfCoords: [{x 1 y 1}, {x 2 y 2}]
coordA.x = 4
//coordA: {x 4 y 1}
//coords and copyOfCoords are unchanged
//coords: [{x 1 y 1}, {x 2 y 2}]
coords[0].x = 10
//coords: [{x 10 y 1}, {x 2 y 2}]
//copyOfCoords is unchanged, because each element is a struct
//copyOfCoords: [{x 1 y 1}, {x 2 y 2}]
```

Because arrays are collections, you can iterate over the contents by using a `for-in` loop:

```
for coord in coords {
    println("Coord(\(coord.x), \(coord.y))")
}
```

You can also use the `enumerate()` function to access an array index inside the `for-in` loop:

```
for (index, coord) in enumerate(coords) {
    println("Coord[\(index)](\(coord.x), \(coord.y))")
}
```

1.2.6 Dictionaries

A *dictionary* is an unordered collection of items of a specific type, each associated with a unique key.

As with arrays, there are two ways to indicate the type of a dictionary: the long form and the short form. These two ways are equivalent and can be used interchangeably:

- Long form: `Dictionary<KeyType, ValueType>`
- Short form: `[KeyType: ValueType]`

The syntax to declare and initialize a dictionary using the short form is:

```
var people: [String:SomePersonClass] = [:] //explicit type
//or, alternately
var people = [String:SomePersonClass]() //implicit type
```

This example declares a dictionary variable called `people`, which will contain `SomePersonClass` values associated with `String` keys, and you initialize it to an empty dictionary.

You can use type inference to let the compiler determine the types of objects in the dictionary when assigning a dictionary literal during the declaration:

```
var highScores = ["Dave":101, "Aaron":102]
```

Because you're initializing the dictionary with keys and values, the compiler infers that the `highScores` variable is a dictionary of type `[String:Int]`.

You can use any type that conforms to the `Hashable` protocol as the `KeyType` value. All of Swift's basic types are hashable by default, so any of them can be used as a key.

You can access and/or manipulate specific values in a dictionary with subscripting:

```
println(highScores["Dave"])
//output: Optional(101)
highScores["Sarah"] = 103
//added a new player
println(highScores["Sarah"])
//output: Optional(103)
//Don't worry about the Optional() portion of the output.
//We introduce that in Chapter 2, "Diving Deeper into
//Swift's Syntax."
```

Because a dictionary is a collection, you can iterate through it with a `for-in` loop:

```
for (playerName, playerScore) in highScores {
    println("\(playerName): \(playerScore)")
}
```

You can determine the number of elements in a dictionary by using the `count` property. Dictionaries also have two array properties, `keys` and `values`, that can be iterated through independently.

Exercises

1. Declare and initialize a pair of variables for each type listed in this chapter, both explicitly and using type inference. What do you need to do to get the compiler to infer the `Float` type?
2. Create a constant with an emoji character in the name. Are you able to easily use the constant? Does this help with the readability of your code?
3. How would you explicitly declare an array that stores another array of `Int`s as each element? Show how you would access elements by using subscripts.
4. Set up a dictionary that uses `Int`s as the key. How is this different from using an array? When could using a dictionary such as this be better than using an array?
5. Create a Fizz Buzz implementation. Iterate through the numbers from 1 to 100. If a number is evenly divisible by 3, print Fizz. If the number is evenly divisible by 5, print Buzz. If the number is evenly divisible by both 3 and 5, print Fizz Buzz. For all other numbers, just print the number.

Index