

Understanding Swift Programming

Craig A. Will

Swift 2

With
Hands-On
Online
Exercises

I don't understand data types in arrays **at all**. Supposedly, all the values in an array must be the same type. I'm putting instances of different classes into the same array. It seems to work, but I don't know why.

```
class Fruit { }  
class Apple:Fruit { }  
aFruit = Fruit()  
anApple = Apple()  
var list = [aFruit, anApple]
```



Tenaya Creek Press

Understanding Swift Programming

Swift 2

Craig A. Will

Tenaya Creek Press

Northern California

TABLE OF CONTENTS

Tap the chapter title to go to that chapter.
Once there, tap *that* chapter title to come back here.

[The Hands-on Online Exercises](#)

[Preface](#)

PART 1: FUNDAMENTALS

[1 Data Types and Type Safety](#)

[2 Some Fundamentals](#)

[3 The Flow of Control](#)

[4 Operators](#)

[5 Arrays](#)

[6 Sets](#)

[7 Strings and Characters](#)

[8 Tuples](#)

[9 Optional Values](#)

[10 Dictionaries](#)

PART 2: OBJECT-ORIENTED PROGRAMMING

[11 Object-Oriented Programming](#)

[12 Classes, Objects, and Inheritance](#)

[13 Functions and Methods](#)

[14 Structures](#)

[15 Enumerations](#)

[16 Properties and Class or Type Methods](#)

PART 2: ADDITIONAL TOPICS

[17 Functions Revisited: First Class Citizens](#)

[18 Closures and Closure Expressions](#)

- [19 Memory Management](#)
- [20 The Flow of Control Revisited: Matching with Patterns](#)
- [21 Error Handling](#)
- [22 Functions Revisited Again: Input Parameters](#)
- [23 Variations in Closure Expression Syntax](#)
- [24 Custom Subscripts](#)
- [25 Operators Revisited: Overloading and Custom Operators](#)
- [26 Optional Values Revisited](#)
- [27 Access Control](#)
- [28 Generic Programming](#)

PART 3: OOP REVISITED

- [29 Classes Revisited Again: Initializers](#)
- [30 Type Checking and Type Casting](#)
- [31 Enumerations Revisited](#)
- [32 Protocols](#)
- [33 Extensions](#)
- [34 Protocol Oriented Programming](#)
- [35 Building Mixed Swift and Objective-C Apps](#)

APPENDICES

- [A What's Different About Swift?](#)
- [B Playgrounds](#)
- [C The Interactive REPL Loop](#)
- [D Additional Resources](#)

Copyright Details

First Edition September, 2015

The Hands-on Online Exercises

An important part of this book is not actually in the book itself, but available online. These are the online exercises for each chapter. They are available by going to the understandingswiftprogramming.com web site with a Macintosh or Windows PC. Type a slash after the URL and then the chapter number. Thus, the address for the exercises for chapter 5 is understandingswiftprogramming.com/5.

After reading each chapter, I suggest going to the online exercises and doing them. They will help you retain the information you learn, allow you to learn things you missed reading the chapter, and generally test your mastery of the material.

The exercises will help you understand the material and also will help you memorize the language. Although you can always look up the less-commonly-used things in the language, you want to be able to immediately write code for the common cases. Retaining the concepts better will also help your learning of additional concepts.

The exercises only work with a Macintosh or Windows PC. A personal computer (Mac) is what you will ultimately be using to write code. The exercises will not work with a mobile device, to avoid wasting your time with what you would ultimately discover to be a horrible user experience, because of the difficulty of entering on a mobile device the specific characters needed for writing code.

Humans generally aren't good at passively absorbing information for very long. I've tried to keep the chapters short, but for some of the longer ones, if you find your attention flagging, you might want to read it in two sittings. After the second sitting, skim the part you read at the first sitting. Then do the online exercises.

How to Contact Us

Any errors that I am aware of are listed on the web site for the book,
understandingswiftprogramming.com. Please report errors or address technical issues to
understandingswiftprogramming@gmail.com.

Trademarks

The book makes reference to a number of trademarks or registered trademarks owned by Apple Inc., which include Apple, Cocoa, Cocoa Touch, iPhone, iPad, iPod, Mac, Xcode, OS X, and Apple Watch. iOS is a trademark or registered trademark of Cisco Systems, Inc.

Preface

Apple's announcement in June, 2014 that they had developed Swift, a completely new language for iOS and Mac app development, shocked the programming world. It also made Objective-C a dying language, although one dying very slowly.

There was a clear need for a new language. Objective-C was designed in the early 1980s, and while it has been updated (e.g., Objective-C 2.0), those updates have been limited. Building a new object-oriented language by adding object-oriented capabilities based on Smalltalk to the existing C language made sense at the time, but the language has become increasingly anachronistic as newer languages emerged that have exciting capabilities and that do not have the safety issues and general clunkiness that Objective-C has.

For more about Objective-C and how Swift is different from it, read the paper in Appendix A, "What's Different About Swift?" This paper describes what the limitations of Objective-C have been, the philosophy of the new language, and the four goals the designers of Swift have set for the language: Safety, clarity, modernity, and performance.

As I discuss in that paper, Apple has very significantly achieved the first three of these goals, and is very likely to achieve the fourth goal of performance soon. Building a new programming language, including the compiler and other tools, and making the application programming interfaces (Cocoa Touch, etc.) work for it is a huge effort, and Apple is still struggling to complete these tasks. However, it appears that with version 2 of Swift, the language and its tools are very close to what is needed for production use, even for large projects, and when it is launched publicly in the fall I expect that adoption of the language will begin to accelerate. (Swift, as of August, 2015, is #17 on the Tiobe programming popularity index and rising rapidly, while Objective-C is #6, down from its earlier presumed peak of #3, and falling rapidly.)

Audience for the Book

This book is intended for programmers who want to learn how to use Swift to build iOS or Mac apps. Readers should have some prior programming experience with a language like C or one of the scripting languages. Experience with object-oriented programming is obviously helpful, but not required.

The main audience for the book is intended to be those new to Swift, but intermediate and advanced programmers will also benefit from some of the later chapters in the book.

Organization of the Book

This book is intended as both an introduction to the language and a description of some of its more complicated aspects. In all cases I have tried to explain things in simple terms.

Most books on Swift, when they deal with a topic, tell you all about that topic. I don't do that. Instead, the book is layered. You'll often get a chapter introducing a topic. Later in the book, you'll see another chapter, "Topic X Revisited" with less-used, or more detailed, or more complicated aspects of that topic. The purpose is to limit how much you have to absorb at first on a given topic, and also to allow you to more quickly understand how the pieces fit together.

The book includes cartoons, which are intended to lighten the mood. Each cartoon highlights some interesting aspect of Swift in a fun way and helps you visualize it.

The book is divided into four parts, plus appendices.

Part 1 describes the fundamental aspects of the language. Most of this is quite straightforward and involves just minor changes to syntax that exists in most languages. Some exceptions to this are Swift's strong emphasis on type safety, and optional values, both very important to understand.

Part 2 describes most of the object-oriented aspects of the language. There are a few confusing things, such as Apple's creating structures and enumerations that often work very much like classes. The many different ways of calling functions and methods can seem rather convoluted, and there is seemingly endless flexibility in the syntax for closure expressions that seems confusing at first but ultimately makes it very easy to write compact closure expressions. Understanding the relationships between closures, functions, methods, and closure expressions is important.

Part 3 describes a number of additional topics, ranging from functions as first class citizens to closures to memory management and error handling. It's at this point that you might want to just start writing apps, maybe skimming the rest of the book until you really need it. You need a good conceptual understanding of the Swift language as a whole, but you don't need to know all the details at first—you can look it up when you need it.

Part 4 describes additional object-oriented aspects of the language, including generic programming, initializers, type casting, protocols, extensions, protocol oriented programming, and programming with a mixture of Swift and Objective-C.

Appendix A, as mentioned earlier, analyzes what is different about Swift, particularly with respect to Objective-C.

Appendix B describes Playgrounds, which allows you to easily test out code.

Appendix C describes the interactive REPL system, a command line alternative to Playgrounds.

Appendix D describes how to find additional information about Swift and, particularly, iOS and the Xcode tools, so that you can begin writing apps.

Errors in the Book

It can be enormously frustrating to be learning a new programming language and to discover that some of the examples in the book you are using don't work. Unfortunately, it's hard to keep errors out of books, particularly when the language and tools are changing and when it is difficult to paste code into Xcode and Playground to test it. (If you try this, watch out!)

If you find example code that does not work, first check that any imports that are necessary (Foundation, UIKit) have been made. If so, check the book's web site, understandingswiftprogramming.com.

Please email me with any errors in the book, code or otherwise:
understandingswiftprogramming@gmail.com.

How to Develop Your Knowledge and Skill in Swift

Programming is a complex skill. And Swift, despite its aspects that often make it simple and clear, is ultimately a rather complex language. To be proficient, you need to develop several different kinds of knowledge and skill. You need to develop an understanding of the language—what it does and how the different pieces fit together. You also need to develop the skill of quickly understanding the Swift code you read, even thinking in that code. And finally, you need to get the Swift syntax “in your fingers”—so that when you need to write code it just comes out without much in the way of conscious thought.

Reading a book is a very efficient way to acquire information. It is often far more efficient than pounding away at code that you can’t get to work because you are missing some tiny but ultimately critical thing in your understanding. Unfortunately, people tend to have limits on how easily they can absorb information from a book, particularly in one sitting. That’s where the online exercises can help.

People learn differently, but I think it can be a mistake to get too quickly into writing code when you need to be absorbing new concepts. This is particularly the case for concepts that are new to you. You may want to first read a chapter, *away* from your computer, in a quiet area, so you are doing only one thing—absorbing the concepts. You then might want to do the online exercises for that chapter.

I suggest then using either the Playground or the REPL interactive tool to try out the code that you see discussed in the chapter. You should experiment around. Change the values from one type to another. Think of short snippets of code that you can write. Get some experience with direct feedback from the compiler.

The order in which you want to do this may depend on your own preferences as well as your background and how new the information is to you.

What You Need to Know to Develop Apps

This book will provide you with the vast majority of what you need to know about Swift. (Swift is still changing, though, and Appendix D will tell you how to tap into additional information about Swift and how it is changing.)

Knowledge and skill in Swift, however, is a relatively small part of what you will need to learn to build apps. You will in particular need to learn the various iOS (or Mac) APIs. You will also need to learn how to use the Xcode interactive development environment and its interactive tool for creating user interfaces, Interface Builder. Of course, how much of the APIs you will need to learn depends upon how sophisticated your apps are and what APIs they will need to use.

What's New in Swift 2?

The following is new in Swift 2:

Error handling. A try-catch mechanism for error handling has been added. Error types are defined with an enumeration. A function is defined that uses the `throws` keyword, indicating that statements in it can throw errors. These statements typically throw an error when appropriate using the `throw` keyword and a member value of the enumeration that defines errors. The function is called within a do-try-catch sequence, with the `try` statement indicating where the potential error is, and `catch` statements catching the errors. See Chapter 21 on “Error Handling”.

Check API availability. The compiler will produce a warning if a developer tries to use an API that does not exist in the target platform. A `statement` allows runtime checking of whether a particular operating system version is available before the code involving it is run, with the option of using different code if that version of the API is not available.

Syntax for print statements. The `println` statement has been replaced by the `print(_:)` statement, which does the same thing. A variation, `print(_:appendNewline:)`, has a Boolean in the second parameter that indicates whether a newline character should be appended after the string that is printed.

Guard statement. The guard, or guard-else statement, is an alternative to the if-else statement that has the same goal but has a syntax that is often less clumsy and more readable.

Repeat-while statement. This is a replacement for the do-while statement that works the same way but avoids confusion with the `do` keyword used in error handling.

Protocol extensions and protocol oriented programming. Protocol extensions allow the definition of method implementations (not just specifying methods) that can be obtained by structures, classes, and enumeration in much the same way that methods are inherited from superclasses. This leads to protocol oriented programming, a new approach that is claimed to be often better than using classes. See Chapter 34 on “Protocol Oriented Programming.”

Function pointers. In Swift 2 it is possible to use function pointers in C, which allows for callbacks in some APIs.

Statement labels. A limited form of a “goto” statement is implemented to allow break and continue statements to work better in nested situations.

If-case statement. The matching capabilities previously available only in switch statements are now available with a new if-case statement.

For-in filtering. When using a for-in statement to iterate through a collection such as an array, you can now use a where clause to select items.

What's New in Swift 1.2?

Swift 1.2 is considered by many developers to be a major advance over previous versions.

Much of Swift 1.2 over the previous version 1.1 involves optimizing the compiler to make it run faster, and fixing various bugs in Swift and the code that bridges it to the Cocoa Touch and Foundation APIs.

A few changes to the language itself have been made, as follows:

Sets. Sets, a minor but missing part of the three collections types of arrays, dictionaries and sets, has been added. A set is an unordered group of values that only has one copy of each value—that is, no duplicates. (See Chapter 6, “Sets”.)

Optional chaining. Changes have been made to the if-let and if-var syntax for optional bindings that make it easier to handle multiple nested if statements for optional chaining. (See Chapter 26, “Optional Values Revisited”.)

Changes to constants. Changes to constants during initialization are no longer allowed.

Zip. A new function, zip, has been added that “zips together” two separate but related arrays into a single array of tuples.

Typecasting operators. The typecasting operators are now as, as!, and as?, with the former intended to remind the programmer that downcasting can fail, and the last returning a nil if a downcast has failed.

Markdown in playground comments. A simple way of adding sophisticated displays of comments, known variously as “markup” or “markdown”, has been added. See Appendix B on the Playground.

1 Data Types and Type Safety



"Ever since North Korea started hacking into Apple's iCloud service, these nasty pointers have been turning up in our Swift code."

Everything referenced by a statement in a computer language that has a representation in memory has a *data type*. A data type, or just *type*, tells the runtime system how to interpret what would otherwise just be a sequence of unrecognizable bits.

For example, if we know that a variable has a name of `n` and a type specified as `Int`, the system can look it up and find that this is an integer that is stored in memory at a certain address and that has a certain number of bits formatted in a certain way.

One of the goals of Swift is to be safe, meaning that it helps programmers avoid errors. And a big source of errors is getting the wrong type and thus misunderstanding the sequence of bits involved. Swift avoids this by being very rigid about types, more so than Objective-C, and much more so than most scripting languages. Swift is thus said to be highly *type safe*.

Swift avoids some of the problems of languages like JavaScript by not allowing a variable's type to be changed once it has been set, and, particularly, not allowing "implicit conversion"—the automatic conversion of one type to another.

In Swift, if you have a string "5", and you want to convert it to an integer so you can do some arithmetic on it, you have to manually choose to do so. The compiler will not do it for you. This avoids ambiguities like:

```
a = "5" + 7
```

Should the variable a above be a string of "57" or an integer of 12?

In Swift, the line above results in a compiler error, and the programmer has to specifically decide what to do.



“George, what did you expect would happen when you tried to add an integer to a floating point number? This is Swift, not JavaScript!”

To some extent this rigidity of typing and other aspects of code safety in Swift reflects a certain philosophy: *Crashing is good!* It is better to catch errors than not, and it's better to catch them earlier in the development process rather than later. It is thus better to catch errors at compile time than runtime. It is also better to catch them when the system is set up to crash whenever something suspicious occurs rather than let errors to slowly cause deterioration of the data in the app so that it exhibits strange but subtle behavior that you cannot track down.

Swift is also safer in another respect in that it does not keep track of objects by defining pointers that reference a particular location in memory, as Objective-C does. Languages that use such pointers are prone to error because a small error that changes the value of a pointer can result in accidentally overwriting a substantial amount of memory, and doing so in a way that can be difficult to detect.

Variables

Fundamental to all mainstream programming languages is the idea of a *variable*. A variable associates a name with an address in the computer's memory and a sequence of bits that is stored there.

Before a variable can be used in Swift, it must be *declared*. This tells the compiler that a variable is being defined and that this is its name and data type.

The following statement declares `n` to be the name of a variable, using the keyword `var`, and assigns it the value 78:

```
var n = 78
```

This says to the compiler "take the number 78 and store it in the computer's memory in the location associated with the variable name `n`".

Variables, as suggested in the earlier discussion, have a data type.

One of the very nice things that the Swift compiler does is *type inference*. That is, when it sees a statement like the one above, setting 78 to the variable `n`, it infers that the variable `n` must be an integer and defines its type as `Int`, the Swift default integer type, and one of the fundamental types in the language.

Note that this is not like implicit conversion, in part because there is no uncertainty about it. Type inference removes most of the drudgery associated with dealing with rigid types and allows a high degree of type safety with relatively little effort.

The programmer can also, when desired, explicitly declare the type of a variable with a *type annotation*:

```
var n: Int
```

This says that the variable `n` has a type of `Int`. It consists of a colon right after the variable name (there can't be a space between the variable name and the colon) and the name of a type. A space after the colon is optional, but is the usual practice.

We can declare a variable without setting a value to it, but if we do not include a value, we have to include a type annotation with it, so the compiler knows the type. If we don't set a value, the compiler doesn't have any way to infer the type.

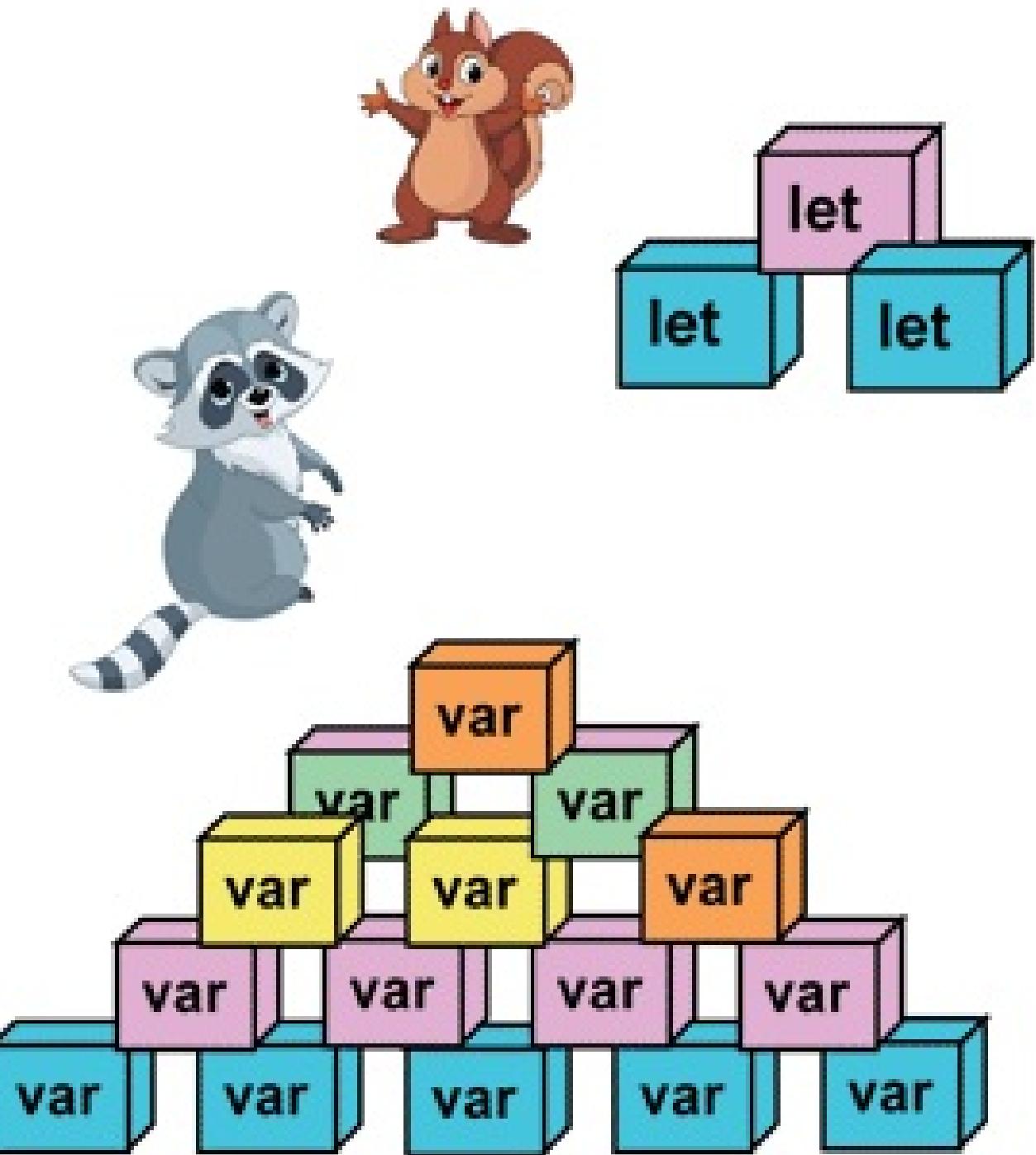
The following will result in a compiler error:

```
var n      // Compiler error
```

Multiple variables can be declared and assigned as part of the same statement (`String` and `Float` are more fundamental types):

```
var n: Int, q: String, r: Float
```

Constants



“Our programmers use very few variables these days—they use mostly constants. That new squirrel on our team ordered a huge number of var keywords, though. What are we going to do with them all?”

Constants in Swift are handled very much like variables. Apple defines a constant as a variable the value of which can be set only once.

A constant is indicated by the keyword `let`:

```
let p = 78
```

In this situation the same thing happens as if `p` were a variable. The constant `p` is declared, and its value 78 for it is stored in memory, along with the type that has been inferred. The only difference is that the value cannot be changed.

The type of a constant can also be set explicitly:

```
let p: Int = 78
```

And if a value is not provided the type must be set explicitly:

```
let p: Int // OK  
let p          // Compiler error
```

An attempt to set a value to a constant a second time will result in a compiler error:

```
let p = 78  
p = 89 // Compiler error
```

The practice recommended by Apple is to use constants, rather than variables, for any values that do not change. This practice increases safety, in that it prevents the error of changing a value that should not change. Most programmers are surprised when they start following this practice and then realize that the code they end up writing has far more constants in it than variables.

“Converting” a Type In Swift

Often you will want to do an operation on variables of different types. For example, consider the following:

```
var a = 5  
var b = 3.2  
var c = a + b // Compiler error
```

If you try to add together an integer and a floating point number that are both stored in variables, you will get a compiler error. (If you try this in Objective-C, it will work fine.)

However, just to avoid confusion here about what works and what does not, note that if the numbers are literals, the compiler will allow you to add them together:

```
var d = 5 + 3.2 // Works OK
```

This is allowed because the compiler is smart enough, when it assigns a type to the literal 5, to assign it a type of `Double`, only because it is in the context of being added to a floating point number. However, in the earlier case, the types have already been assigned.

What can we do if we have variables with mixed types that we want to add together? Once a variable is given a type (either explicitly or by type inference), that type cannot be changed. However, there are two things that can be done short of this.

First, you can create a new instance of a desired type by providing a value in another type to an initializer of the desired type.

Thus, if you have a value that is an integer but you need it as a floating point number, you can do the following:

```
var x = Double(5)
```

This declares a variable `x` of type `Double` and assigns it a value of `5.0`.

You can also do:

```
var j = 5  
var k = Double(j)
```

which does the same thing.

Returning to the example where we got the compiler error:

```
var a = 5  
var b = 3.2  
var c = Double(a) + b // Works OK
```

The `Double(a)` created a new value of type `Double` and value 5.0, and it could then be added to the `b` value of 3.2.

A second thing that can be done short of changing a type is known as *type casting*. This is normally allowed only for instances of classes that are part of an inheritance hierarchy. What is usually done is to *downcast* the definition of a type of a variable from one higher up in the inheritance hierarchy to one lower in the hierarchy. This is not actually changing the type but only allowing it to be used temporarily as the type that is lower in the hierarchy. Although the general rule is that if a variable is declared to be a particular type, you can only assign values of that type to it, the rules of polymorphism mean that you can also assign values that are instances of subclasses. This issue is covered more thoroughly in Chapter 30 on “Type Checking and Type Casting”, and is also discussed in Chapter 11 on “Object-Oriented Programming” in the section on subtyping.

Types in Swift

Aside from the desire to be rigid about types for the purpose of safety, the way that types are defined also says a lot about the architecture of the Swift language. The language consists of a relatively small number of fundamental types, plus a few types that have the capability of allowing a programmer to create a new custom type in the language. In the remainder of this chapter, I will provide a brief description of all of the types that have been defined in Swift. This will help you get a sense of the language.

Fundamental Data Types

Int

Stores integer number as 32 or 64 bits,
depending on device

**Int8, Int16,
Int32, Int64**

Signed integers of various lengths

**UInt8, UInt16,
UInt32, UInt64**

Unsigned integers of various lengths

Double

High precision floating point number
(default)

Float

Low precision floating point number

Bool

Boolean, value can be true or false

Fundamental Types in Swift

The fundamental types are as follows:

INTEGERS

The basic data type for an integer presented earlier was `Int`. The actual size of the integer as represented in memory, in terms of bits, depends upon the machine. A 32-bit machine will use 32 bits to represent an integer declared as an `Int`. A 64-bit machine will use 64 bits to represent an integer declared as an `Int`.

In addition, it is possible to declare specialized integer types that are unsigned or that have a specific size, or both.

Thus `UInt` declares a variable or constant to be unsigned, meaning that it is neither positive nor negative. (An unsigned integer can hold a larger positive value than a signed integer of the same size.) As in the case of `Int`, a 32-bit machine will represent a `UInt` with 32 bits, while a 64-bit machine will represent a `UInt` with 64 bits.

The types `Int8`, `Int16`, `Int32`, `Int64`, `UInt8`, `UInt16`, `UInt32`, and `UInt64` all represent valid integer types with different combinations of signed vs unsigned and size.

The usual practice is to just use `Int` for all integer types. The other types may be useful if it is desired to store a lot of data in integer form in a particularly efficient way.

FLOATING POINT NUMBERS

There are two types for floating point numbers, `Double` or `Float`, but `Double` is the default and, realistically, the only one you need to be concerned with unless you are dealing with some kind of legacy code or legacy APIs that need `Float` types, or need to conserve space. A standard 64-bit processor, which most devices now use, will represent a `Double` level of precision.

The statement:

```
var p = 5.7
```

results in the variable `p` having a type of `Double`. If you want the type to be `Float`, you have to declare this explicitly:

```
var p: Float = 5.7
```

BOOLEANS

In Swift, there is only one kind of Boolean type, `Bool`, and it must have one of only two values, `true` and `false`.

Collection & Related Data Types

String Ordered sequence of characters, Unicode

Character Single character.

Array Ordered sequence of values contained in variable or constant

Set Unordered group of unique values

Dictionary Unordered group of key-value pairs

Tuple Ordered sequence of values within parentheses

Other Types in Swift

In addition to the fundamental types, there are other types that are often composed of the fundamental types.

STRINGS

A string is a sequence of one or more characters, which may be Unicode characters that represent single letters, logographic characters, or emoji:

```
var a = "5"  
var b = "Good afternoon"  
var c = "的" // Chinese  
var d = "🍔" // Emoji
```

CHARACTERS

In Swift, a literal single character is represented by being enclosed in double quotes and, in addition, being explicitly defined as having a type of Character:

```
var ch: Character = "a"
```

ARRAYS

An array is an ordered sequence of values. The values may be of any type but all of the values in an array must be of the same type. The first line of the example shown below is an array containing floating point numbers that have a type of Double. The second line shows the same thing but with the type explicitly defined.

```
var temperature = [67.8, 34.2, 76.3]  
var temperature: [Double] = [67.8, 34.2, 76.3]
```

SETS

A set is an unordered group of values for which any value is only represented once. The values may be of any type but all of the values in a set must be of the same type. The first line of the example shown below is a set containing strings that have a type of `String`. The second line shows the same thing but with the type explicitly defined. (Sets are new in Swift 1.2.)

```
var airportCodes = Set(["SFO", "OAK", "LAX"])
var airportCodes: Set<String> = Set(["SFO", "OAK", "LAX"])
```

DICTIONARIES

A dictionary is an unordered group of key-value pairs. The keys and values may be of any type but all of the keys must be of the same type and all of the values must be of the same type. The first line of the example shown below is a dictionary containing a key-value pair that has that have a type of `String` for the key and `Int` for the value. The second line shows the same thing but with the type explicitly defined.

```
var personsAndTheirAge = ["Tom": 12, "Peter": 22, "Beth": 74]
var personsAndTheirAge: [String: Int] = ["Tom": 12, "Peter": 22, "Beth": 74]
```

TUPLES

A tuple is a simple way to pass or store a group of related values. These typically have different types and can't be put in an array because the values in an array must all be of the same type. The first line of the example shown below is a tuple that has an `Int` for the first position in the tuple and a `String` for the second position. The second line shows the same thing but with the type explicitly defined.

```
var errorMessageTuple = (432, "Error: Something Bad Happened")
var errorMessageTuple: (Int, String) = (432, "Error: Something Bad Happened")
```

Type Declarations for Functions, Methods, and Closures

Functions, methods, and closures might normally not be seen as things that have types, but they actually do. Their types, known as *compound types* because they consist of multiple fundamental types, consist of the types of their input parameters and return parameter.

FUNCTIONS AND METHODS

Functions and methods have types that are defined by the types of the parameters passed to them and their return value. Thus:

```
func addTwoNumbers (a: Int, b: Int) -> Int {  
    var c = a + b  
    return c  
}
```

This function has a type of `(Int, Int) -> Int`. This might be expressed in English as “A function with two input parameters, each with an integer type, and a return type of integer”. (A method in Swift is just a function that is associated with a class or similar type.)

CLOSURE EXPRESSIONS

A closure expression, like a function, has a type that is defined by the types of the parameters passed to them and their return value. Closures have a very flexible syntax and may or may not have input parameters and/or return values.

A simple form of a closure is shown in the first line of the example, while a complex closure is shown beginning in the second line. The comment next to each indicates the type of the closure.

```
{ print("hey") } // Simple syntax  
// Type is ()-> Null  
  
{ (a: Int, b: Int) -> Int in  
    var c = a + b
```

```
return c
}
// Full syntax
// Type is (Int, Int) -> Int
```

Optional Types

Everything in Swift that has a type and can contain a value can alternatively be defined as having an *optional* version of that type. Thus, the following code declares `n` to be a variable with a type that might be described in English as an “optional integer”:

```
var n: Int? = 5
```

An optional version of a type is represented by a type name followed by a question mark. Ordinary, non optional types require that a value be stored in them; that is, they cannot be `nil`. An attempt to store a `nil` in them will result in a runtime error. A variable that is declared as an optional, however, is allowed to be `nil`.

The causing of a runtime error when a `nil` is stored in a non optional variable is a first line of defense in ensuring that variables have values. There is also a second line of defense. A variable declared as an optional that contains a value must be *unwrapped* to get to that value. An attempt to use a value from a variable declared as an optional that has not been unwrapped will normally cause a runtime error. An attempt to unwrap a value from a variable declared as an optional when that value is `nil` will also normally cause a runtime error. Swift has a number of capabilities in the language for dealing with optionals depending upon the particular situation. See Chapter 9 on “Optional Values”.

Creating Custom Types

The following types, including classes, structures, and enumerations, are used for creating new, custom types. The data types discussed above that were referred to as “fundamental” types are not actually fundamental types hard coded in the language. They were created as custom types of structures.

CLASSES

A new custom type named Banana can be defined with the following code:

```
class Banana {  
}
```

STRUCTURES

A new custom type named Kumquat can be defined with the following code:

```
struct Kumquat {  
}
```

ENUMERATIONS

A new custom type named DayOfTheWeek can be defined with the following code. This defines the only allowed values for the new type. The type DayOfTheWeek has seven allowed values.

```
enum DayOfTheWeek {  
    case Monday  
    case Tuesday  
    case Wednesday  
    case Thursday  
    case Friday  
    case Saturday
```

```
case Sunday
```

```
}
```

Aliases for Types

A type can be referred to by another name if that name is defined with the `typealias` keyword. For example, the statement:

```
typealias Byte = UInt8
```

will allow you to refer to an eight bit unsigned integer by using the alias `Byte`.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 1 exercises, go to

understandingswiftprogramming.com/1

Do NOT go to the site with a tablet or phone—typing in code with a mobile keyboard is a horrible experience, and the web site will not work with mobile devices, in order to not waste your time.

2 Some Fundamentals



**“What are we going to do with all
these semicolons?”**

This chapter describes some rather miscellaneous but important aspects of Swift that you should know early on. This includes what names are allowed for variables and constants, Swift not usually needing semicolons, comment syntax, printing, string interpolation, assertions and errors, the scope of variables, range operators, underscore (`_`) characters, and checking the availability of specific iOS (or Mac) APIs. These are aspects of Swift that are often different for Swift than for other programming languages.

Variable and Constant Names

Names for variables and constants can include letters of the alphabet and numbers, although they cannot begin with a number. They can include any public Unicode character. Logographic characters such as those in Chinese and Japanese are allowed, as are emoji characters. (See Chapter 7 for a discussion of Unicode.) The following are all valid names:

```
var a = 5  
var b378 = 378  
var 🍔 = "Hamburger"  
var 的 = "Chinese"  
var 絵 = "Japanese"
```

Names cannot contain whitespace characters and cannot contain symbols unless those symbols represent valid Unicode characters. The following are not valid names

```
var 378a = 378 // Not Valid, begins with a number  
var error Code = 345 // Not Valid, contains whitespace
```

Once you've declared a constant or variable of a certain type, you can't redeclare it again with the same name, or change it to store values of a different type.

Semicolons and Multiple Statements on a Line

Swift does not require a semicolon at the end of a statement, although you can use one if you like:

```
var m = 78 // Usual practice  
var m = 78; // Allowed
```

Multiple statements can be placed on a single line, but semicolons are then required to separate the statements:

```
var m = 78; var p = 8.9; var r = "hey"
```

Note that there is no semicolon at the end of the line. Semicolons are statement separators in Swift, and are not required at the end of the line although, again, they are allowed:

```
var m = 78; var p = 8.9; var r = "hey";
```

Comments

As in most C-like languages, the compiler will ignore text to the right of a “//” sequence on a line:

```
// This is a comment, please ignore it Mr. Compiler
```

Similarly, the compiler will ignore text between /* and */ sequences, either on one line or across multiple lines.

```
/* This text will be ignored */
```

```
/* And also this text and the text on the line below it,  
including this text */
```

An unusual aspect of Swift is that it allows nested /* */ comments:

```
/*  
x = 37;  
/* this is an important parameter and if it is wrong it  
might cause a crash */  
*/
```

The idea here is that you sometimes want to use a /* */ sequence to comment out a block of code to see if, say, the crash you are getting goes away. But in most languages, if you have already embedded comments in the block using /* */ you can't do this. In Swift you can.

Printing

You can “print” a string or a variable with `print`. There are two versions of it. The most commonly used version has a single parameter consisting of a string to be printed. This prints the string and follows it with a newline character. (In other words, the statement prints the string and moves the cursor to the beginning of the next line.)

The code is:

```
print ("You got here in your program")
// Prints: You got here in your program
```

The alternative version has two parameters. The first parameter contains the string to be printed, while the second parameter is a Boolean that indicates whether the string should be followed by a newline (`true`) or not (`false`).

```
var m = 78
print (m, appendNewline: true)
// Prints: 78\n (78 followed by newline character)

print(m, appendNewline: false)
print(m, appendNewline: false)
// Prints: 7878
```

Note that “print” here usually means “display on the console log”.

(This is a change from Swift 1. In Swift 1, there were two print statements: `print`, and `println`. The `print` statement printed a value with no newline character appended. The `println` statement printed a value with a newline character appended.)

String Interpolation

You can insert values from constants and variables into strings with *string interpolation*. This is particularly useful with the `print` statement discussed above.

In string interpolation, the name of a variable or constant is placed within a string and surrounded by parentheses, with a backslash just before the left parenthesis.

```
var m = 78
var s = "This is an ordinary string with an embedded value
of \(m)"
print(s)
// Prints: This is an ordinary string with an embedded value of 78
```

More commonly, the printing would be done directly:

```
print("This is an ordinary string \$(m)")
```

Assertions

An assertion is a runtime check to determine whether a particular condition is true. If the condition is NOT true, the execution of the program halts with a runtime error and a message is printed (displayed) on the console log.

The keyword `assert` is used, along with two parameters: a conditional expression and a string to be displayed as a message should the condition be false. Thus:

```
x = 5
assert (x > 6, "x must be greater than 6")
// Will halt with runtime error
```

In this example the variable `x` is set to 5, but the assertion conditional expression is `x > 6`, meaning that the normal expectation is that `x` will be greater than 6. Since the program fails this test, the program will halt and the message “assertion failed: `x` must be greater than 6” will be printed out on the console log.

Error Handling—New in Swift 2

Swift has (new in Swift 2) a sophisticated error handling system that is similar to, but in some ways different from, the try-catch approach used in many other languages.

See Chapter 21, “Error Handling”, for details.

Scoping of Variables

Whenever a variable is declared it is done so in a particular scope, meaning a location where that variable is known and can be used.

Consider the variables in the following code:

```
var n = 5
if x == 0 {
var p = 7
}
```

And let's assume that this code is in the global space in a Swift file—it is not in a class, function, or the like.

The variable `n` is defined at global scope. If we added a couple of `print` lines to the code above, as follows, what would happen?

```
var n = 5
if x == 0 {
var p = 7
    print(n) // Prints: 5
}
print(n) // Prints: 5
```

Both would work. The variable `n` works both in the global scope it was defined in and in the narrower scope of the `if` clause.

However, suppose we attempt to print the value of `p` in the same way:

```
var n = 5
if x == 0 {
```

```
var p = 7
print(p) // Prints: 7
}
print(p) // Error
```

The first `print` line would work, but the second will blow up. Why? It's because the variable `p` was created in the scope of the `if` clause. It exists only in that scope, and an attempt to access it outside of that scope will fail.

Range Operators

Swift has what are known as *range operators*, that allow referring to a range of items rather than a specific item.

A *closed range operator* has a syntax like `1...7`. With a closed range operator, we can define a range that runs, for example, from 1 to 7 as follows:

```
for k in 1...7 {  
    print("k is \(k)")  
}
```

A *half-open range operator* works similarly, but does not include the ending value. The following example iterates 6 times, rather than the 7 in the example above:

```
for k in 1..<7 {  
    print("k is \(k)")  
}
```

1.

WTF is this?

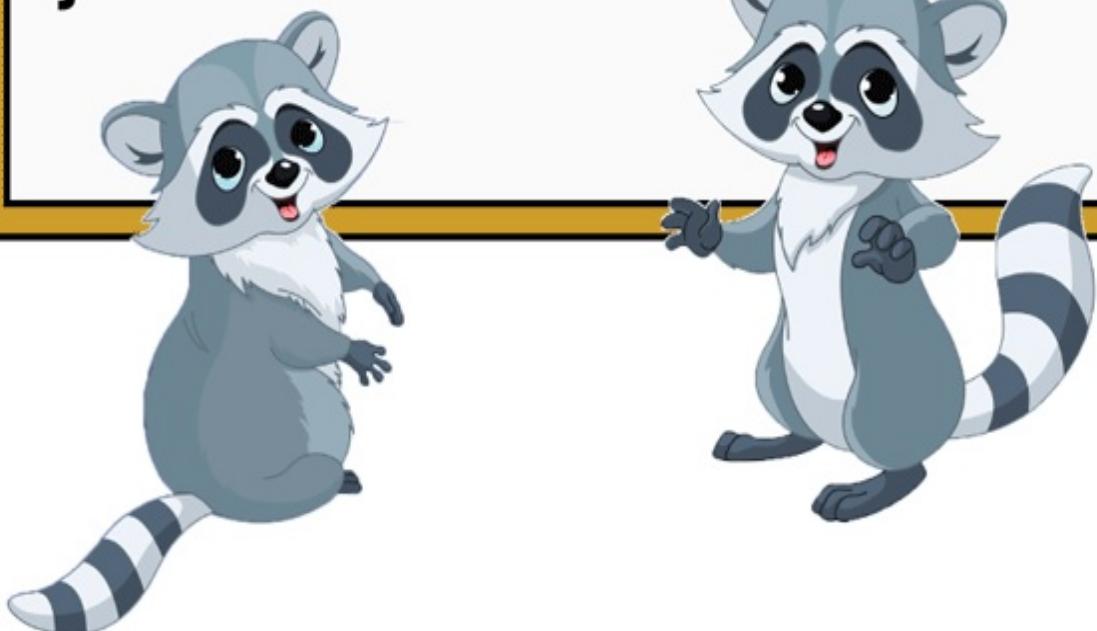
```
switch (name) {  
    case ( _, "Peter") println ("Pe
```



2.

It's an *underscore* character. It is a sort of wildcard. Values do not have to match, or even exist. It's used in input parameters, switch matching, and iterations.

```
callThis ( _, second: Int, third: Int) {  
    for _ in cities {  
        println("One iteration")  
    }  
}
```



Underscore Character

An underscore (`_`), also known as a wildcard, is used in Swift in a number of different situations. The commonality across these situations is that a variable or constant that is normally used in that situation isn't actually needed, and the underscore is used instead.

For example, a certain kind of for-loop can iterate through an array using code like this:

```
for k in cities {  
    print("One iteration")  
}
```

In this particular situation the variable `k` isn't actually needed for anything beyond being a sort of placeholder in the for loop, and an underscore character alone can work just as well:

```
for _ in cities {  
    print("One iteration")  
}
```

Underscores are also used to indicate that an expected input parameter in a method call is not necessary, and in switch statement matching of tuples to indicate that any value will match for that particular position. Details of how this is done are provided in Chapter 20, “The Flow of Control Revisited”.

Aliases for Types

An alias for a type can be defined using the `typealias` keyword, allowing the type to be referred to in a more easily understood way. For example, the following code creates a type alias for three different levels of speech quality, depending upon how many bits are used to represent it:

```
typealias excellentQualitySpeech = UInt32
typealias goodQualitySpeech = UInt16
typealias marginalQualitySpeech = UInt8
```

A programmer can then just refer to one of the quality levels:

```
var speechSample: goodQualitySpeech
```

This has a couple of advantages. First, it's easier to remember, to document, and to communicate with others that one chose the "good" quality speech option. And second, it makes it easier to change. You may end up with a program with a lot of references to "goodQualitySpeech", but decide that that speech isn't good enough quality, or is taking too much memory. You can then change the `typealias` to a different integer type, either to experiment, or permanently. You can do this once rather than having to change multiple references.

```
typealias Byte = UInt8
```

will allow the type `UInt8` to be referred to by the alias `Byte`, as in the following:

```
var byteCoding: Byte = 354
```

This explicitly sets the type of the variable `byteCoding` to `UInt8`, which may be a more obvious way to indicate why you are choosing that integer length.

Checking API Availability

New in Swift 2 is a capability for checking the availability of an API (e.g., in Cocoa Touch) before executing code making use of it.

There are two parts to this capability. The first part is in Xcode. Xcode will give you a warning if you attempt to use an API that is only available in an iOS version that is newer than your defined target platform. (When you create an app project in Xcode you must define a “Deployment Target”, which is the version of iOS that is the earliest (“oldest”, “minimum”) that you will allow your app to run on. This might be, for example, 8.1 or 7.1.)

Suppose that you are building an app and want to make use of an API that is only available in iOS 9. However, you set your deployment target to iOS 8 so that devices that have not yet installed iOS 9 can use your app. If you have code that uses an API that is only contained in iOS 9, however, Xcode will warn you that you need to do something about the problem.

The second part of the availability capability is seen when the app runs. You can use an if-else statement to check to see if a particular API is available on the platform the app is running on. If it is available, your code for that API runs. If it is not available, code contained in the else clause runs.

For example, you may want to use a new API in iOS 9. But if it is not available on a particular device, you can use a less-desirable, but still workable API for the version of iOS that the device is running. The code is:

```
if #available(iOS 9, *) {  
    // code to use the iOS 9 API goes here  
}  
  
else {  
    // code to use a previous iOS version API goes here  
}
```

The #available condition takes as parameters the names of operating systems and versions. If more than one is specified, they are separated by commas. (A Mac OS might be specified by OS X 10.10). Both general version numbers (e.g., iOS 9) and specific versions (e.g., iOS 9.01) can be specified.

The asterisk (“*”) used as the last argument is required. This is looking ahead for when the code might be run on a different operating system, such as the Apple Watch operating system. What the asterisk says is that on that different operating system, the code in the main block (if statement) will be run if the version of that operating system is at or after the earliest (that is, “minimum”) deployment target set for that operating system.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 2 exercises, go to

understandingswiftprogramming.com/2

3 The Flow of Control



**"I'm taking these surplus parentheses
to the GooglePlex."**

Swift has the basic C-like capabilities for handling the flow of control in a program, including if, if/else, and if/else-if/else statements.

Swift also has the familiar for and while statements that allow iteration by means of an integer counter or while a certain condition is still true. In addition, Swift has a repeat-while statement, similar to the do-while statement in many languages.

In addition, Swift has a for-in statement that allows iterating through arrays, dictionaries, strings, and ranges.

Swift also has the conventional switch statement, but has changed its logic slightly and made its matching capabilities more flexible.

A number of minor changes in the syntax of these statements have been made to Swift that are intended to increase safety and clarity.

Changes to Improve Safety and Clarity of Conditional Expressions

The following changes have been made to improve safety and clarity, and apply to all conditional expressions:

First, parentheses are not required around conditional expressions.

Thus, the following expressions are legitimate in Swift:

```
if p == 5 {  
    print("p must be 5")  
}  
  
for var i = 0; i < 5; i++ {  
    print("iteration with i=\\"(i)")  
}
```

This is arguably cleaner and easier to read, and if it is easier to read it should be easier to detect errors and is thus safer. (Not everyone agrees that this makes the code easier to read.)

Parentheses are allowed if desired:

```
if (p == 0) {  
    print("p is equal to 0")  
}
```

A second change is that braces are required around statements that are part of conditional clauses, even in the case of a single statement. The above expressions are correct. The one below is not:

```
if p == 0 print("p is equal to 0") // Compiler error
```

Putting braces around single statements that are part of conditional clauses has been a

common practice among many programmers, intended to promote consistency and good habits, although it has not been required by most languages.

ALL HANDS MEETING TODAY

1. The California Drought—Our Plan
2. Shortage of Curly Braces



“I think I’ve solved No. 2, but it’ll cost us. I’m going to trade a truckload of parentheses to the Android team for half a truckload of curly braces.”

The third change is that assignment statements are not allowed within conditional expressions. So you are not allowed to do the following:

```
var m = -2
if var p = m + 2 { // Compiler Error
print("p is equal to 0")
}
```

This technique is often used in C to make the code more compact but is an invitation to error when a programmer leaves off a “=” when intending to use a “==”. If such assignments are simply not allowed, the compiler will catch that error.

The fourth change is that conditional expressions must use actual Boolean types, not just integers with values of 0, 1, or some other nonzero value that are posing as Booleans:

```
var m = 1
if m { print("m is true") } // Compiler Error

var m = true
if m { print("m is true") } // OK
```

If, If/else, and If/else-If/else Statements

Swift has the conventional if, if/else, and if/else-if/else statements. We have already seen examples of the if statement.

Swift also allows the usual if/else statement:

```
if m < 5 {  
    print("m is less than five")  
}  
  
else {  
    print("m is greater than or equal to five");  
}
```

And it allows if/else-if/else statements:

```
if m < 5 {  
    print("m is less than five")  
}  
  
else { if p == 7 {  
    print("m is >= 5 and p is equal to 7");  
}  
  
else {  
    print("m is >= 5 and p is NOT equal to 7");  
}  
}
```

Logic Operators and Expressions in Conditional Tests

The following conditional operators are used, as is common:

&& Logical AND

|| Logical OR

< Less than

<= Less than or Equal

> Greater than

>= Greater than or equal

! Logical NOT

Complex expressions are allowed:

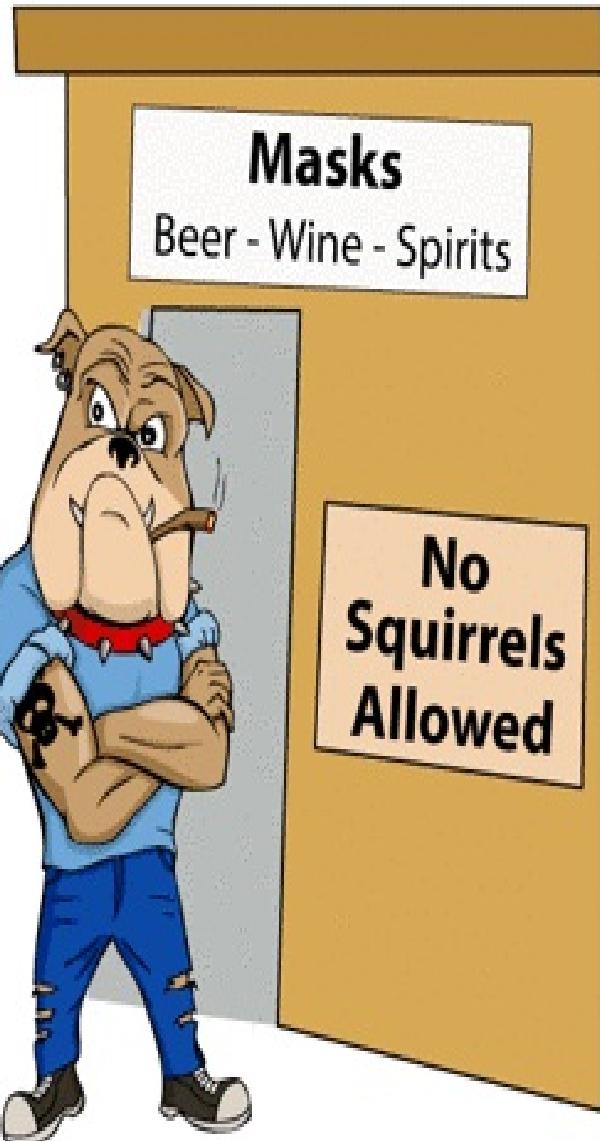
```
if m < 3 && n < 8 || w > 2 {  
    print("The condition is met")  
}
```

In many such cases parentheses are desirable because they make the expression more readable:

```
if (m < 3 && n < 8) || (w > 2) {  
    print("The condition is met")  
}
```

The Guard Statement

Not making an exception is an error.



"Hey, you can't come in here. We have rules. And Chris, the owner, doesn't believe in exceptions."

The guard statement is an alternative to the if-else statement that often has a cleaner, more readable syntax. The guard statement works like an if-else statement, but has a different syntax and flow of control. It is basically an if-else statement with the logic inverted. If the “guard” (if) condition is true, it does not execute an associated statement that is contained within curly braces like an if statement would. Instead, control passes to the statement just after the end of the “else” part of the guard statement.

An if statement might look like this:

```
if a == 5 { print("a is 5") }
else { print("a is NOT 5") }
```

A guard statement, in contrast, looks like this:

```
guard a == 5
else { return }
print("a is 5")
```

Note that there is no pair of curly braces analogous to those for the “if” statement. This makes sense, given where control is passed. It also means that any variables or constants created after the `guard` keyword will remain in scope after the end of the `else` part of the guard statement.

The guard statement does have a requirement that an if-else statement does not: The `else` clause must transfer control out of the current flow. The guard statement is thus most commonly used in situations where that is likely, either by returning from a function/method or by throwing an error. The `else` part of the statement is obviously required—there would be no point to the statement without it.

Eric Cerney, in a discussion (ericcerney.com/swift-guard-statement) about why he likes Swift guard statements, calls the pattern that guard statements make easy to create the “Bouncer Pattern”. (See the cartoon above.) The idea is to quickly get rid of the bad cases (like a bouncer for a bar does), then you can focus on what needs to be done for the good cases.

A guard statement can use variable binding, and the variables or constants that are created

will be available for code after the end of the guard's else clause.

Guard statements are often used in error handling, and they can help avoid the convoluted multiple nested if-else statements that can result. For an example of this, see Chapter 21 on “Error Handling”.

For Loops

The usual C-like for loop is used in Swift, but does not require (but allows) the parentheses:

```
for var m = 0; m < 5; m++ {  
    print("hey")  
}
```

Semicolons, however, are still required in for loops to separate the parts of the statement that set the initial value, test during each iteration, and perform the iteration.

The same rules about parentheses being optional and braces required around even single statements apply to for loops, while loops, do-while loops, and for-in loops. Assignments are similarly not allowed within the iteration or conditional testing expressions.

The variable in a for loop being iterated must be declared with a `var` keyword (it obviously cannot be a constant). The logic of the loop is conventional: the variable `m` is (usually but not necessarily) initially set to 0, and then before each iteration is made a test is made. If the result of that test is true, the statement or statements between the braces are executed. Afterwards, an iterating statement is executed, in this case iterating the variable `m`. A test is then again made to see if another iteration should be performed. When the test is false, the iteration stops, and control passes to the statement after the last statement in the for loop.

While Loops

While loops test a condition and execute a statement or statements if the condition is true, and keep executing repeatedly as long as the statement is still true.

```
m = 0
while m < 5 {
print("m is \\"m\\")
m++;
}
```

Do-While Loops [Deprecated in Swift 2]

Do-while loops, which exist only in Swift 1, are similar to while loops, but the test is after the first iteration of the statement(s), ensuring that the statement(s) will be executed at least once:

```
var m = 0
do {
    print("m is \(m)")
    m++
}
while m < 5
```

Repeat-While Loops

Repeat-while loops are similar to while loops, but the test is after the first iteration of the statement(s), ensuring that the statement(s) will be executed at least once:

```
var m = 0
repeat {
    print("m is \(m)")
    m++
}
while m < 5

// Prints:
m is 0
m is 1
m is 2
m is 3
m is 4
```

This is new in Swift 2 and replaces the previous do-while loop defined in Swift 1. It works exactly the same as the do-while loop. The reason for the change is to avoid confusion with new do-try-catch syntax for error handling.

Iterating Through Arrays, Dictionaries, and Strings

Swift has a for-in statement that allows iterating through an array, dictionary, or string. Thus, if an array is:

```
let cities = ["San Francisco", "Oakland", "San Jose"]
```

It can be iterated through as follows:

```
for city in cities {  
    print("The city is \(city)")  
}
```

The variable `city` is set to the value of the element in the array and can be accessed within the statements for each particular iteration. Note that the variable does not have to be declared with a `var` keyword (and if you include it you will get a compiler error); the declaration has already been made as part of the for-in syntax.

The for-in statement can also iterate through a *range* in Swift.

A range indicates a range of values. Swift expresses this in a shorthand syntax, with two forms:

The range `0...5` (using 3 period characters) indicates the values 1 through 5. This is called a *closed range*.

The range `0..<5` (using 2 period characters) indicates the values 1 through 4. This is called a *half-closed range*. This latter expression is often used for iteration.

For-in Filtering

For-in filtering allows a way to select items when iterating through a collection using for-in. This avoids the messiness of another nesting of a control statement, such as an if statement. Instead, a where clause is used.

For example, if you have a collection of numbers and you want to print out only the even ones, you can do:

```
let numbers = [1,2,3,4,5,6,7,8,9,10]
for number in numbers where number % 2 == 0 {
    print(number)
}
```

(Example from Natasha the Robot)

Break and Continue Statements

The statements `break` and `continue` can be used in any of the statements above that do iteration. The `break` statement causes the iteration to stop immediately, with control passing to the statement just after the iterative loop. The `continue` statement causes the iteration to stop immediately and for control to be passed to the beginning of the next iteration.

Statement Labels

A *statement label* is a string that is followed by a colon that appears in front of certain statements in Swift. A statement label is accompanied by a reference to it in a break or continue statement that is used on one of the various loop statements (for, while, repeat) or in a switch statement.

This is a very limited form of the controversial “go to” statement, which is not available in Swift. A go to statement transfers control to anywhere in a program. (See the well-known “Go to statement considered harmful” letter to the editor of the *Communications of the ACM*, published by E. W. Dijkstra in 1968.)

The following code shows the problem and how a labeled statement solves the problem:

```
Loop1: for j in 0...10 {  
    Loop2:  
        for i in 0...10 {  
            print("hey")  
            var result = getIt(j,b: i)  
            if (result == true) {  
                break Loop1  
            }  
        }  
    }  
}
```

The code has two for loops, one nested within the other. The idea here is that the `getIt()` function does some calculation and may or may not get a result. If it does not, we want to continue in the loop. If it does, we want to get out of the loop.

If there was no break statement, the `print ("hey")` statement would be executed 100 times.

A break statement will cause the immediate passing of control to the first statement outside of the loop. The remaining code for the rest of that cycle, and for any additional cycles of the loop, is not executed.

The problem here is that an ordinary break statement will get us out of the second loop, the one with the `i` being incremented. But it will not get us out of the first loop. We want to get out of both loops.

The labeled statement allows this. When the break statement references a label, it says that it wants to end execution of the statement that is named by that label.

Switch Statements

A switch statement is an alternative to an if/else-if/else statement, useful when things get complex. A variable or constant that contains a value is matched against values that are contained in multiple case clauses. When a match is found, the statement(s) contained in the case that matched is executed. When that code is done executing, control passes to the first statement after the full switch statement. A match is always found because switch statements in Swift are required to be exhaustive. If the existing cases are not exhaustive, a default case is added, as shown below, that is executed when the other cases do not match. The default case must come at the end, after the other cases.

The way to think of a switch statement is that it looks at a value and then “switches” execution to a particular piece of code depending upon that value.

Switch statements in Swift have a number of variations from those in C that are intended to increase safety and clarity and also make the statement considerably more powerful.

Switch statements in C are actually quite primitive, with the value being matched against either an integer or something that can be converted into an integer value, such as a character code or enumeration expressed as an integer.

In Swift, the matching is very flexible. The following example shows a simple matching against values that are strings.

```
var name = "George"

switch name {
    case "Peter":
        print ("The name is Peter")

    case "George":
        print ("The name is George")

    case "Jennie":
        print ("The name is Jennie")
```

```
default:  
    print ("No match found")  
}
```

The variable or constant being matched against (In this case, name) does not have to be enclosed in parentheses, although it may be.

When a match is found, the code in the case statement that matches is executed. Each case must contain at least one executable statement.

The behavior for the flow of control through the `case` statements is different from C. In C, if the test for the first `case` statement is `true`, the statement or statements below the `case` statement are executed. The flow of control then passes to the next `case` statement. In C, the programmer will typically put a `break` statement in as the last statement for that case, causing the program to exit the switch statement. However, programmers often forget to put in the `break` statements, which can cause problems if there is also a match with the other cases.

Swift will exit the switch statement after the execution of the statements associated with a case, if there is a match for that case. The effect of this is as if there was a `break` statement at the end of every group of statements after a case; thus `break` statements are no longer required for this purpose. `Break` statements are still allowed to be used and might be used in a more complex set of statements within a case to break out of that particular case. A `break` statement can also be used as the only statement executed in a case. This is sometimes useful when it is desired to do nothing if a particular case is matched.

If the statement `fallthrough` is used, it will cause the code in the next case to be executed (in much the same way that switch statements work in C.)

A `case` statement can contain more than one match, with the different matches separated by commas. They can be placed on more than one line if desired:

```
switch name {  
    case "Peter", "George":  
        print ("The name is either Peter or George")  
  
    case "Jennie",
```

```
"Susan":  
    print ("The name is Jennie or Susan")  
  
default:  
    print ("No match found")  
}
```

Multiple values can be listed for each case, with the values specified after the case keyword but before the colon, and separated by commas.

In this chapter I have described the basic syntax of switch statements and the differences between C and Swift, particularly the difference in behavior relating to `break` statements.

Switch statements also have a lot of flexibility in terms of matching against different types, using wildcards in matching, matching against ranges, and matching against tuples

A later chapter (Chapter 20 on “The Flow of Control Revisited: Matching with Patterns”) will discuss the more sophisticated matching capabilities that are in the switch statement that use patterns, as well as the (new in Swift 2) if-case and for-case statements that also use patterns for matching.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 3 exercises, go to
[understanding swift programming.com/3](http://understandingswiftprogramming.com/3)

4 Operators



“Is it because programming was invented by humans that there is so much puffery? The **nil coalescing operator?** Big deal. It tests for NIL and assigns a variable with one value or another depending upon the result.”

The most basic part of most programming languages is the statement, with its variables, constants, literal values and, above all, the operators that serve as the glue that holds the statement together:

```
var c = a + 37
```

Here we have the *addition operator* (“+”) that we all learned about in elementary school, and also the *assignment operator* (“=”), the latter used in programming but not in mathematics.

What is an operator? An operator is a symbol representing an operation that produces a result from one or more *operands*. An operand is a value that may be in the form of a variable, constant, expression, or literal value. The symbol for an operator is usually one or two special characters, but can also be a word. For example, Swift includes the operator `is`, used in type checking, or determining the type of a variable.

Another way to look at an operator is as a function. Like a function, it takes input values and produces a return value.

Swift includes the typical operators that most C-like languages have, plus a few new ones, including the range operators ... and ..< and the type checking and type casting operators `is`, `as?`, and `as!` Most operators work like those in other languages, but a few do not. The remainder (“%”) operator in Swift can produce floating point values, and the arithmetic operators do not allow underflow or overflow (that is, producing a result that is larger or smaller than the type can represent.)

Arithmetic Operators

Operators are typically divided into groups, according to the number of operands each operator uses. The defined Swift arithmetic operators are as follows.

Operators with a single operand (unary operators).

- Minus
- ++ Increment
- Decrement
- ! Negation
- + Plus

Unary operators must be placed immediately adjacent to their operand, with no intervening whitespace. Thus:

```
var a = -5
```

is correct, while

```
var a = - 8 // Compiler error
```

is not correct.

Operators with two operands (binary operators).

- + Add
- Subtract
- / Divide
- * Multiply
- % Remainder

The remainder operator works “sort of” like division. Imagine `a` being divided by `b`. The highest integer that can be multiplied by `b` and still be less than `a` is found. This is then subtracted from `a` to produce the remainder. This operator in Swift will, unlike other languages, produce a remainder that is a floating point number, if the operands are floating point types. Thus, for integer types:

```
var a = 7 // Compiler infers a to be an Int  
var b = 2 // Highest number is 3 to be multiplied by 2  
var c = a % b // Result is 1
```

For floating point types:

```
var a = 8.0 // Compiler infers a to be a Double  
var b = 1.5 // Highest number is 5 to be multiplied by 1.5  
var c = a % b // Result is 0.5
```

Operators can also be placed in different positions relative to operands. In the case of operators with only a single operand, there are two possibilities: prefix and postfix:

```
++a // Prefix  
a++ // Postfix
```

Whether an operator is in the prefix or postfix position often does make a difference. In the case of the increment operator, we can see this with the code below:

```
var m = 5  
print(++m) // Prints: 6  
  
var n = 5  
print(n++) // Prints: 5
```

In the first example of printing `++m`, the increment is done and then the expression evaluated before printing, with the expression value 6. In the second example of printing `n++`, the expression is evaluated first, yielding a value of 5, and the increment is done

afterwards.

In the case of operators with two operands, there is only one possibility: infix:

a + b // Infix

Comparison Operators

Swift has six comparison operators that each compare two values. These are used in C and many similar languages:

- `==` Equal
- `!=` Not equal
- `>` Greater than
- `>=` Greater than or equal to
- `<` Less than
- `<=` Less than or equal to

Each operator is used in an expression with two operands and tests whether the comparison is true or false, yielding a Boolean `true` or `false` value.

Thus, the expression

```
4 == 5
```

is false, while

```
"George" == "George"
```

is true.

The result of such an expression can be assigned to a variable:

```
var result: Bool = 3 == 5
```

Although perfectly legitimate, this looks a bit odd and it is common to put parentheses around such expressions to make them easier to read:

```
var result: Bool = (3 == 5)
```

Comparison operators are commonly used directly in conditional expressions:

```
if x >= 5 { print ("x is greater than or equal to 5") }
```

Logical Operators

`&&` // AND

`||` // OR

`!` // NOT

Other Operators

Operators with two operators (binary operators)

. . < Half-open Range

... Closed Range

?? Nil Coalescing Operator

Operators with three operands (ternary operators).

? : Ternary

The range operators were discussed in Chapter 2 on “Fundamentals”, while the Nil Coalescing Operator will be discussed in Chapter 9 on Optional Values.

The “ternary condition operator” uses two symbols. The expression to the left of the ? symbol is evaluated. If it is true, the statement just to the right of the ? symbol is executed; otherwise the statement just to the right of the : symbol is executed.

Operator Precedence

When more than one operator is in a statement, the issue comes up of what order the operators are executed in. Swift, like other C-like languages, evaluates expressions from left to right.

Consider the following statement:

```
a = 2 + 3 * 6
```

If we follow a strict left-to-right evaluation, `a` will be assigned a value of 30.

However, Swift, like most C-like languages, gives some operators precedence over others, in particular, multiplication over addition. So in the statement above, `a` will be set to 20, not 30.

Swift's compiler has a number that indicates the precedence levels, as follows:

Multiply and Divide 150

Addition and Subtraction 140

Range 135

Type check and Type cast 132

Compound Assignment Operators

A *compound assignment operator* combines assignment with another operation. For example, to concatenate two strings:

```
nameOfMammalianSouthAmericanSpecies = "South American  
Jaguar"
```

```
nameOfMammalianSouthAmericanSpecies += ".txt"
```

Without the compound operator, you would have to use:

```
nameOfMammalianSouthAmericanSpecies = "South American  
Jaguar"
```

```
nameOfMammalianSouthAmericanSpecies =  
nameOfMammalianSouthAmericanSpecies + ".txt"
```

If you have a lot of long variable names in your code, this kind of operator will save you a lot of typing and make your code easier to read, once you get used to it. The assignment/addition operator is by far the most heavily used, but all of the following compound assignment operators can be used. They are shown along with their longer alternative.

a+=b a = b + a

a-=b a = b - a

a*=b a = b * a

a/=b a = b / a

Advanced Operators

In addition to the basic operators described above, Swift also has what Apple calls *advanced operators*. These include *bitwise operators*, *bit shifting operators*, and *overflow operators*. I'll describe these only briefly here, since they aren't that commonly used in apps. For details see the Apple documentation.

Bitwise Operators. The bitwise NOT operator (“`~`”) inverts all of the bits in a binary number. (That is, changes the 1s to 0s and the 0s to 1s.) The bitwise AND operator (“`&`”), given two binary numbers, produces a third number which is a logical AND of the bits in the given numbers. The bitwise OR operator (“`|`”), given two binary numbers, produces a third number which is a logical OR of the bits in the given numbers. The bitwise XOR operator (“`^`”), given two binary numbers, produces a third number which is a logical XOR, that is, an exclusive OR, of the bits in the given numbers.

Bit Shifting Operators. The bitwise left shift operator (“`<<`”) moves all of the bits in a binary number a specified number of places to the left. The bitwise right shift operator (“`>>`”) moves all of the bits in a binary number a specified number of places to the right.

Overflow Operators. The overflow operators “`&+`”, “`&-`”, “`&*`”, “`&/`”, and “`&%`” prevent triggering a runtime exception that would otherwise occur, should the result of a computation be a value too large or too small to be stored in a particular integer type. The extraneous bits are truncated. `&/` also allows division by zero without an exception, with the result set to 0.

Controversy Over Operators

Operators seem mostly straightforward, but there is still a little controversy over them. Matt Thompson (NSHipster.com) questions using the “+” operator for concatenation as well as addition:

Why should adding two strings together concatenate them? $1 + 2$ isn't 12 (except in JavaScript). Is this really intuitive, or is it just familiar?

The ternary condition operator, which doesn't even have a nice name, is hated by many programmers.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 4 exercises, go to
understandingswiftprogramming.com/4

5 Arrays

1.

I don't understand data types in arrays *at all*. Supposedly, all the values in an array must be the same type. I'm putting instances of different classes into the same array. It seems to work, but I don't know why.

2.

Your Apple class is a subclass of your Fruit class. The compiler infers a type of Fruit for the array. The instance of Apple is allowed in an array of its supertypes. It's called subtyping.

```
class Fruit {}  
class Apple:Fruit {}  
aFruit = Fruit()  
anApple = Apple()  
var list = [aFruit, anApple]
```

```
class Fruit {}  
class Apple:Fruit {}  
aFruit = Fruit()  
anApple = Apple()  
var list = [aFruit,
```



An *array* in Swift is an ordered list of values. Arrays in Swift are declared to have values only of a specific type, and all values must be of that type or you will get a compiler error. This is part of the emphasis on type safety in Swift.

However, note that “the same type” has a liberal interpretation, given *subtyping*. See the discussion on subtyping in the section on Polymorphism in Chapter 11 on “Object-Oriented Programming” and the discussion in Chapter 30 on “Type Checking and Type Casting”.

Examples of arrays are:

```
var birds = ["Cardinal", "Steller's Jay", "Finch"]  
var temperatures = [100.3, 88.3, 299.2]
```

In the case of the `birds` array, the values are inferred to be of type `String`, while in the case of the `temperatures` array, the values are inferred to be of type `Double`.

Arrays can be variables, like those above, or constants:

```
let marineMammals = ["Sea Lions", "Sea Otters", "Humpback  
Whales", "Spotted Dolphins"]
```

If an array is specified as a constant, it cannot be modified. Elements cannot be added to it, replaced, or deleted. An array specified as a variable can have elements added to the end of it, inserted in the array at any point, replaced, or deleted from the array.

Constant and variable arrays in Swift are basically replacements for the Objective-C classes `NSArray` and `NSMutableArray`. The values of `NSArray` cannot be changed once set (that is, they are immutable), while the values of `NSMutableArray` can be changed.

Because of the requirement that in Swift arrays must be all of the same type, arrays in Swift are easier to use than their Objective-C counterparts but are less flexible. Objective-C arrays can accept any object, but it must be a true object, not a fundamental type like an integer or floating point number. Fundamental types must be converted to an object like `NSNumber` before being placed in an array, and converted back when retrieved.

The type of an array consists of the type of the values in it surrounded by brackets. The following arrays have their type explicitly declared:

```
var birds: [String] = ["Cardinal", "Steller's Jay",
"Finch"]

var temperatures: [Float] = [100.3, 88.3, 299.2]
```

The type is defined as `[String]` or `[Float]`. This means an array consisting of values with the type `String` or `Float`.

The declaration and the assignment can be separate statements:

```
var cities: [String]

cities = ["San Jose", "Ventura", "San Luis Obispo"]
```

Note, however, that to declare a variable or constant without initializing it, you must do it explicitly with a type annotation.

An empty array can be created as follows:

```
var cities = [String]()

var cities: [String] = []

var numbers = [Int]()

var numbers: [Int] = []
```

There are two ways of accessing arrays: using subscripts, or using the methods associated with arrays. We will demonstrate access with subscripts first, and then with the methods associated with arrays.

Accessing Arrays Using Subscripts

Subscripts are commonly used as an easy way to access values in an array.

We can use a subscript to retrieve a value:

```
var cities = ["San Jose", "Ventura", "San Luis Obispo"]
var nameOfCity = cities[1]
print (nameOfCity) // Prints: Ventura
```

(The first item in a Swift array has an index of 0, as in most languages.)

Subscripts can be used to change an array in any way allowed, including appending a value to the end of the array (if indirectly), replacing one or more values of an array, inserting one or more values into an array at a given location, deleting a value from an array, and even deleting all of the values of an array.

Thus, the following will change the value of the second item in the array from “Ventura” to “San Diego”:

```
cities = ["San Jose", "Ventura", "San Luis Obispo"]
cities[1] = "San Diego"
```

The result is:

```
["San Jose", "San Diego", "San Luis Obispo"]
```

Note, however, that we cannot use a subscript to refer to an item in an array that does not exist, or we will get a compiler or runtime error:

```
cities = ["San Jose", "Ventura", "San Luis Obispo"]
cities[3] = "San Diego" // Compiler Error
```

We can also replace multiple values at the same time. This makes use of the *range* syntax

discussed in an earlier chapter. Thus, the following will replace both “San Diego” and “San Luis Obispo”:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
cities[1...2] = ["Mendocino", "Humboldt"]
```

The numbers in the range refer to the subscript indices in the existing array. This says “replace the values in locations 1 to 2 with the following elements”.

The result is:

```
["San Jose", "Mendocino", "Humboldt"]
```

We can do this even if the number of values we are adding is different from the number of values that we are specifying in the subscripts, allowing us to effectively insert additional values to or delete values from the array:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
cities[2...2] = ["Mendocino", "Humboldt", "Redding"]
```

This yields the following:

```
["San Jose", "San Diego", "Mendocino", "Humboldt",
"Redding"]
```

Note that this replaces the last element in the array. We cannot use subscripts directly to append an item to the end of the array because we cannot use a subscript that points to beyond the boundaries of the existing array.

We can add a new value to the end of the array, however, with the `append` method:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
cities.append ("Sierra Madre")
```

The array will now be:

```
[“San Jose”, “San Diego”, “San Luis Obispo”, “Sierra  
Madre”]
```

And note that arrays can hold multiple copies of the same value. If we do the same append operation again, we will get two “Sierra Madre” values:

```
cities.append(“Sierra Madre”)
```

The array will now be:

```
[“San Jose”, “San Diego”, “San Luis Obispo”, “Sierra  
Madre”, “Sierra Madre”]
```

Accessing Arrays Using Methods

We have already seen that we can add elements to the end of an array with the `append` method:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
cities.append ("Sierra Madre")
```

The `removeAtIndex` method will remove a particular element:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
cities.removeAtIndex(2)
```

will result in the array:

```
["San Jose", "San Diego"]
```

The method `insertAtIndex`:

```
cities = ["San Jose", "San Diego"]
cities.insert("Monterey", atIndex:1)
```

will result in the array:

```
["San Jose", "Monterey", "San Diego"]
```

The method `removeAll`:

```
cities = ["San Jose", "San Diego", "Monterey"]
cities.removeAll()
```

will result in the empty array:

```
[ ]
```

The method `removeLast`:

```
cities = ["San Jose", "San Diego", "Monterey"]  
cities.removeLast()
```

will result in the array:

```
["San Jose", "San Diego"]
```

The Reference Manuals for the classes `NSArray` and `NSMutableArray` contain potentially useful methods that can be used to operate on Swift arrays. The `NSArray` class contains methods for operations that are read-only, while the `NSMutableArray` class contains methods for operations that change the values in the array.

Note that if you want to use APIs like `NSArray` and `NSMutableArray` in Playground, you have to import either the Foundation framework or the UIKit framework (which includes Foundation).

Additional Array Capabilities

We can use the “+” or addition operator to add two arrays together:

```
let cities = ["San Jose", "San Diego"]
let cities2 = ["San Luis Obispo", "San Francisco"]
let cities3 = cities + cities2
print(cities3)
// Prints: ["San Jose", "San Diego", "San Luis Obispo", "San Francisco"]
```

Two useful properties are associated with arrays. The `count` property will tell you the size of the array:

```
print("The cities array has \$(cities.count) cities in it")
```

The `isEmpty` property (a Boolean) will tell you whether the array has no elements (has a count of 0):

```
if(cities.isEmpty) {
    print("The cities array has no values in it")
}
else {
    print("The cities array has at least one value")
}
```

You can also iterate over an array:

```
let cities = ["San Jose", "San Diego", "San Luis Obispo",
"San Francisco"]
for city in cities {
    print(city)
}
```

The details of the for-in syntax has been explained in the chapter on the flow of control. This is just a for loop that goes through every item in the array, from beginning to end, setting the variable `city` each time to the value in the array.

Arrays are value types. (They are implemented as structures). That means that if you make a copy of an array, and change an element in it, it will not change the original. Thus:

```
cities = ["San Jose", "San Diego", "San Luis Obispo"]
var cities2 = cities
cities2[0] = "Monterey"
print (cities) // Prints: ["San Jose", "San Diego", "San Luis Obispo"]
print (cities2) // Prints: ["Monterey", "San Diego", "San Luis Obispo"]
```

Zipping Together Two Arrays

A capability that was added in Swift 1.2 for arrays is the `zip` method. This allows you, given two arrays, to “zip them together” to create a single array of tuples. Each tuple has an element from the first array in the first position, and an element from the second array in the second position.

For example:

```
let array1 = [1, 2, 3]
let array2 = ["one", "two", "three"]
let zippedArray = zip(array1, array2)
```

The above code will produce the following array:

```
[ (1, "one"), (2, "two"), (3, "three") ]
```

This can be seen most easily with:

```
print (Array (zippedArray))
```

(At present zipped arrays do not seem to be accessible via subscripts.)

Note that there is a potential problem when the two arrays are not exactly the same length. If this happens, elements at the end of the longer array will be discarded as necessary.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 5 exercises, go to
[understanding swift programming.com/5](http://understandingswiftprogramming.com/5)

6 Sets



“That nutty squirrel is telling everyone that Set is like an array, except that arrays are actually useful. That’s nonsense. With a Set you can check to see if something is in a list, and it is very fast. And you can use a for-in loop to get every item in the list, and they are unique values.”

A *set* is an unordered list of elements that contains only one copy of each particular value. A set in Swift has the name Set.

Sets are new in Swift 1.2 and are one of three forms of what are known as *collections*, or types that contain multiple elements. The others are arrays and dictionaries.

By *element* I mean an object or some lesser data type such as an integer, string, character, Boolean, or an instance of a custom enumeration or structure.

A set is similar to an array, but is different in two ways according to the definition above. First, it is unordered. You can access a set using an index, but there is no guarantee about what element you might get. Second, a given value can be stored in a set only once. With an array, you can set every element of the array to the same value if you like.

Compared to an array, a set may seem to have a rather limited value. It certainly has a limited range of uses, but is very valuable for those situations in which it is used.

It is sometimes useful to know whether a value has already been dealt with, and thus is can be helpful to keep a list of values, using a data type that does not allow duplicate values.

A set is optimized for what it is good at (such as determining whether a particular values is contained in the set) and is very fast.

Creating a Set

To create a new set, you initialize it using the following syntax:

```
var fruits = Set<String>()
```

This creates an empty set and assigns it to the variable `fruits`, which will be of the type `Set<String>`, meaning a Set containing elements that are each of the type `String`.

A set can also be created by initializing it with the same sequence of literal values that is used to initialize an array:

```
let fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
```

You can declare the type explicitly as follows:

```
let fruits: Set<String> = Set(["Apple", "Blueberry",  
"Peach", "Mango"])
```

Determine Whether a Value is Contained in a Set

Once a set has been created, it can be checked to see if a given value is contained in it:

```
let blueBerryContained = fruits.contains("Blueberry")
```

The above statement will set the constant `blueBerryContained` to `true` if the literal string value “Blueberry” is contained in the set `fruits`. If that value is not contained in `fruits`, the constant will be set to `false`.

Inserting and Removing Values

Once a set has been created, a new value can be inserted into it as follows:

```
var fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
fruits.insert("Apricot")
```

Once a set has been created and contains at least one value, that value can be removed as follows:

```
var fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
fruits.remove("Blueberry")
```

Once a set has been created and it contains values, all of the values can be removed as follows:

```
var fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
fruits.removeAll()
```

Determining the Number of Elements in a Set

Once a set has been created, the number of values contained in it can be determined as follows:

```
let fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
let numberOfElements = fruits.count
```

And once a set has been created, a check can be made of whether it is empty:

```
let fruits = Set(["Apple", "Blueberry", "Peach", "Mango"])
let isEmpty = fruits.isEmpty
```

The constant `isEmpty` will be set to `true` if `fruits` is empty; `false` if it is not empty.

Other Functions

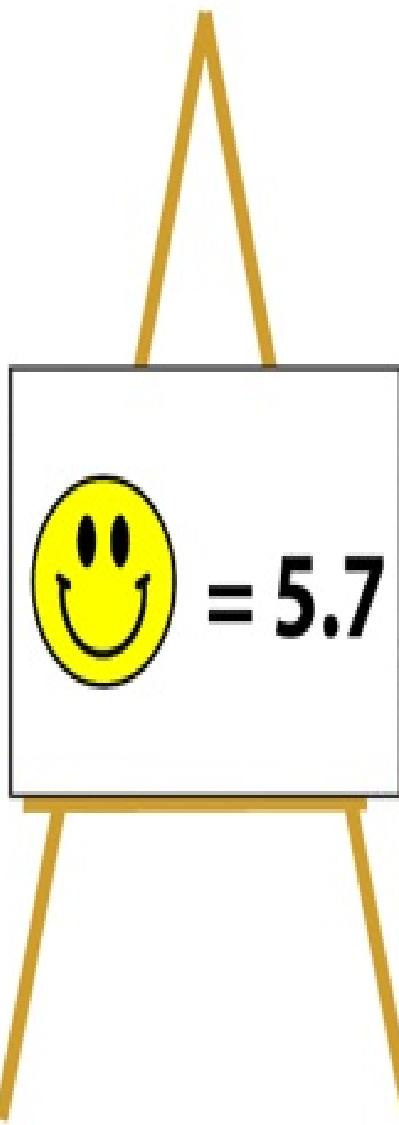
The `Set` class can also do a number of other functions, including determining if a set is a subset, strict subset, superset, or strict superset of another set, determine whether two sets have duplicate values, combine sets, subtract one set from another, and create a set of common or uncommon members.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 6 exercises, go to
understandingswiftprogramming.com/6

7 Strings and Characters



“Don’t you think that there are a few too many things in Swift that you’ll never use, like using a smiley face as a variable name?”

Swift handles text by the use of *strings* and *characters*. A string in Swift is an ordered sequence of characters, and has the type `String`.

A variable or constant that contains a single character can have the type `Character`, but can also have the type `String`.

Thus,

```
var s = "Hello"
```

assigns the variable `s` to the string “Hello”, and, through type inference, sets the type of `s` to `String`.

Alternatively, the type can be set explicitly:

```
var s: String = "Hello"
```

Note that double quotes must be used around literal character sequences. Unlike in Objective-C, single quotes will not work and will result in a compiler error:

```
var s = 'Hello'    // Compiler Error  
var ch = 'H'       // Compiler Error
```

An assignment to a variable of a single letter will result in a type being inferred of `String`, rather than `Character`:

```
var s = "H"    // Type inferred is String
```

To assign a single letter to a variable and have it be a type of `Character` requires that the type be set explicitly:

```
var ch: Character = "H"    // Type set explicitly to Character
```

If you have a character stored in a variable with a type of `Character`, and you need it to be a string, you can't just change the type of the variable, or assign it to a variable that you explicitly declare to be of type `String`. You need to create a string from the character with the `String` function:

```
var ch: Character = "H"    // Type set explicitly to Character  
var s = String(ch) // Will be inferred to be String
```

Similarly, if you have a variable of type `String` that contains a single character, and you need it to be of type `Character`, you must create it with the `Character` function:

```
var s: String = "H" // Type is String  
var ch = Character(s) // Type inferred to be Character
```

Strings can be added together, or *concatenated*, with the “`+`” operator:

```
let a = "The sun is "  
let b = "rising."  
var c = a + b  
print(c) // Prints: The sun is rising.
```

Strings in Swift can represent a large number of different characters, including those from logographic writing systems like Chinese:

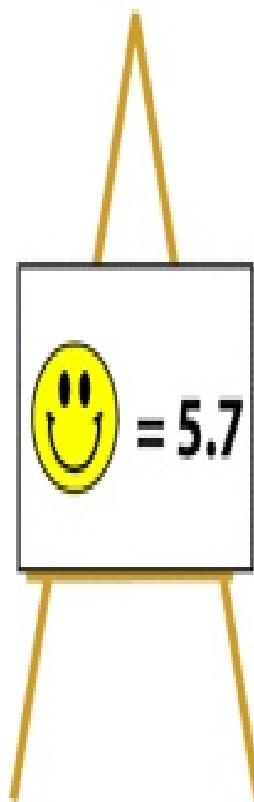
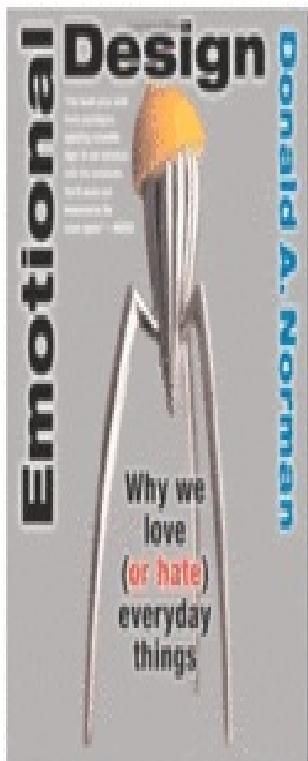
```
var s = "的" // Chinese  
var s = "繪" // Japanese
```

Swift can also represent the specialized characters known as “emoji”:

```
var hamburger = "🍔"
```

To allow these and the more conventional characters to be represented, Swift makes use of the Unicode standard.

(To enter emoji characters inTextEdit on a Macintosh, go to Edit > Emoji & Symbols (or, on older Macs, Special Characters) and you will get a popup menu that you can navigate through with a large number of emoji characters.TextEdit works with Unicode; Microsoft Word 2011, the latest production version of Word available for the Macintosh as of mid 2015, does not. However, the 2016 Preview edition of Microsoft Word, available as of mid 2015, *does* support Unicode.)



“For our new emotion-based user interface, we need a variable that represents happiness.”

The Unicode Standard

Unicode is a standard for representing characters that goes way beyond the traditional ASCII and allows the definition of some 110,000 characters. This includes characters for a large number of written languages, including the word-based (logographic) characters used in Chinese and Japanese, and the syllable-based characters used in Korean.

This makes things a little more complicated because each character does not take a fixed number of bits, like it does in Objective-C. Instead, a character can take from 1 to 4 bytes. A character like “A” takes one byte; a complicated emoji like 🍔 might take 4 bytes.

An emoji like 🍔 is simply a pictogram that is a single character. The idea of an emoji—and the term—comes from Japan, where it literally means “picture character”.

An emoji is not the same as an “emoticon”, such as the smiley face :–), which just uses existing character shapes to crudely draw something resembling a picture. The terms have no relationship with each other: *emoji* is a word borrowed from the Japanese language, while *emoticon* is an English word derived from the words “emotion” and “icon”.

Swift allows any Unicode character to be used in a string. Unicode characters are represented by a unique number, known as a *code point*, that is intended to represent a particular abstract character, known as a *grapheme*. A grapheme in a writing system like English that is based (more or less) on phonemes is an orthographic character, like “Z”. A grapheme in a writing system like Chinese uses a logographic character, like “的”. Each has its own Unicode code point.

Unicode is concerned with the representation of abstract characters (graphemes). It does not involve itself with the specific shape that represents a grapheme, known as a *glyph*. That is left to the particular system software implementation. Thus Unicode has a single grapheme that represents a “hamburger”. The iOS operating system provides a glyph that can be displayed as the visual form of the “hamburger” grapheme that looks like this: 🍔

The picture of the hamburger will look slightly different, however, on an Android system, for example, as that system will provide its own glyph for “hamburger”, such as the following: 🚫

Unicode characters can also be defined with a numeric code that represents the code point. Thus:

```
var a = "\u{1F354}"
```

```
print(a) // Prints: 🍔
```

will display the emoji character representing a hamburger.

It is common to refer to Unicode characters in terms of their code points using “U+” followed by the hexadecimal number for the code point. Thus, the code point for the hamburger character would be referred to as U+1F354.

The characters that you will be concerned with are known as *Unicode scalars*, the most commonly used Unicode characters. Unicode scalars have code points in the range U+0000 to U+D7FF or U+E000 to U+10FFFF. They do not have code points in the range U+D800 to U+DFFF, which are reserved for what are known as *surrogate pair code points*. Surrogate pair code points are used to represent characters that cannot be represented by a single code point (because there are not enough code points). They are instead represented by a pair of code points.

The Number of Characters in a Swift String

Characters in Unicode can be expressed in either the UTF-8 or UTF-16 encodings. In UTF-8, some characters, such as simple ASCII characters, require only one byte (8 bits) to represent. Others, such as the emoji characters, can require up to four bytes. UTF-16 usually represents characters with 16 bits, or two bytes, but in some cases can require up to four bytes.

Unlike with Objective-C, which uses UTF-16 for strings using the `NSString` class, there is no simple `length` property that indicates the length of a string. Instead, Swift considers the length of a string to be the number of characters contained in it. And each character is defined as a perceptually different character, according to the Unicode approach.

Thus, the English orthographic character “Z”, the Chinese logograph 的, and the pictogram for a hamburger 🍔 are all considered to be a single character, even though each requires a different number of bytes to represent it.

The idea that a character is defined as what is perceived is taken quite seriously, as can be seen in examples in which a character can be displayed either by a single character or with a combination of characters. For example, the character é (e with an acute accent mark) can be displayed either by its own Unicode character:

```
var ch: Character = "\u{E9}" // Displays the character é
```

or by displaying two Unicode characters, an ordinary “e” followed by an acute accent character:

```
var ch: Character = "\u{65}\u{301}" // Displays ordinary e, then adds acute accent mark.
```

In both cases, Swift considers this to be a single character. A sequence of Unicode characters that displays a single perceptible character is known as an *extended grapheme cluster*.

From the Apple documentation:

Two String values (or two Character values) are considered equal if their extended grapheme clusters are canonically equivalent. Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they are composed from different Unicode scalars behind the scenes.

Another character that is produced with two successive Unicode characters is the U.S flag, the so-called “regional indicator for US”, which looks like this: 

Its Unicode representation is:

```
var usFlag: Character = "\u{1F1FA}\u{1F1F8}"
```

To count the number of characters in a string, you must use the `characters.count` property for the string variable. (This is Swift 2 only.) This only works with strings, not characters. Below, we first convert the `Character` to a string, then access `characters.count`. We can do this first with the sequence of “characters” consisting of “e” followed by the acute accent character:

```
var ch: Character = "\u{65}\u{301}" // e followed by acute accent
var s: String = String(ch)
print(s.characters.count) // Prints: 1
```

We will see the same thing for the U.S. flag:

```
var ch: Character = "\u{1F1FA}\u{1F1F8}" // US Flag ("Regional
indicator")
var s: String = String(ch)
print(s.characters.count) // Prints: 1
```

In both cases `characters.count` returned a count of 1, because there is only one perceived character, even though it is actually represented by two underlying characters.

And:

```
var s = "Hello Mr. 🍔"
```

```
var numberOfCharacters = s.characters.count  
print(numberOfCharacters) // Prints: 10
```

This will count 10 as the number of perceptibly different characters, reflecting the nine alphabetic characters (including one space) and single emoji character. (In the example shown, there should be no space between “Mr.” and the adjacent emoji character; my technique for representing emoji characters here, and making sure they display on all ebook devices, is a bit crude.)

We can determine the actual number of bytes used to represent this string by looking at the 8-bit codes one by one. The property `utf8` contains a representation of the string in UTF-8 format, that is, byte by byte.

We can use a for-in loop to look at each byte and print out its code in decimal:

```
var s = "Hello Mr. 🍔"  
for code in s.utf8 {  
    print("\u202a(code)", appendNewline: false)  
}
```

This prints: 72 101 108 108 111 32 77 114 46 240 159 141 148

There are 13 numbers, suggesting that there are 9 bytes for each of the alphabetic characters, and 4 bytes for the hamburger glyph.

If we look at the numbers, we see 72, which is standard ASCII for “H”, and, 46, the fifth from the end, which is standard ASCII for the “period” character. Thus it makes sense that the final 4 bytes in the sequence represent the emoji.

Note that there is no length property for a string in Swift, as there is in `NSString`. (`NSString` is the standard Cocoa library class for strings that is used by Objective-C.) This is presumably deliberate, to avoid misleading programmers who should be using `characters.count`. There are properties like `s.utf8.count` and `s.utf16.count` (which may not agree) and which provide counts with different rules. (Swift 2.)

Testing for an Empty String and Comparing Strings

TESTING FOR AN EMPTY STRING

The following will test to see if a string is empty:

```
let s = "" // Creates an empty string
if s.isEmpty { print("The string s is empty") }

let s = "This string has content"
if s.isEmpty {
    print("s is empty")
} else {
    print("s has content")
}
```

COMPARING TWO STRINGS

In Swift, a test for equality of two strings uses the “`==`” operator, and it compares the content of the strings, not just their pointers as in Objective-C. When characters are compared, they are compared based on how they are perceived by the user, not the actual data representing them.

```
var a: Character = "\u{E9}" // The character é
var b: Character = "\u{65}\u{301}" // e, then an acute accent character
if a == b {
    print("a and b are the same")
}
else {
    print("a and b are different")
}

// Prints: a and b are the same
```

In the example shown, the strings in `a` and `b` are considered the same because they will be perceived the same, even though the underlying data is different.

Methods for Manipulating Strings

A variety of methods are available for working with strings. Some of them work by executing the method directly on a Swift string of type `String`. Some of them will not work this way, but will work if you first use type casting to (temporarily) create a string with a type of `NSString`. These show just some of the methods available. Apparently all of the methods described in the `NSString` documentation can be called with the type casting approach.

For any of these methods to work, either `UIKit` or `Foundation` must be imported. This is especially important to remember when trying them out in Playground or the interactive REPL. Some string functions, particularly casting a Swift `String` type to an `NSString` type, do not appear to work in the REPL in at least some versions of Xcode.

Only a few string manipulation functions have been implemented in the Swift language itself as of Swift 1.2. However, bridging between Swift and the `NSString` class is relatively seamless, and it is thus relatively easy to use the `NSString` functions in ordinary Swift code. The strings are represented in `NSString` as 16-bit codes, and will not work properly if any of the characters are larger than this, such as the 4-byte emoji characters.

CREATING A STRING

The following creates an empty string:

```
var s = String()
```

Alternatively, the following will also create an empty string:

```
var s = ""
```

MANIPULATING THE CASE OF LETTERS

The methods `lowercaseString` and `uppercaseString` will take a string and make all of the characters lowercase or all uppercase, as follows:

```
let s = "Mixed Case"  
var sUpperCase = s.uppercaseString  
print(sUpperCase) // Prints: MIXED CASE
```

```
let s = "Mixed Case"  
var sLowerCase = s.lowercaseString  
print(sLowerCase) // Prints: mixed case
```

ACCESSING CHARACTERS IN A STRING

To access a single character at a particular index in a string:

```
let s = "A string with characters"  
let index = advance(s.startIndex, 4)  
var a = s[index] // Gets a character at index  
print(a) // Prints: r
```

Because characters that take as many as four bytes are allowed, it is not possible (in unextended Swift) to use a subscript with an integer index to access a character in a String. You have to use the rather unwieldy `advance` function. There is a trick can be used to do this, however, as an alternative. You can define an extension that allows a String to use a subscript that takes a type of `Int`. This will work, however, only if you avoid characters like emoji and Chinese characters and the like.

ACCESSING SUBSTRINGS OF A STRING

Swift:

```
let str = "Hello cruel world"  
let index = advance(str.startIndex, 6)  
let endIndex = advance(str.endIndex, -6)  
let sub = str[Range(start: index, end: endIndex)]  
print(sub) // Prints string "cruel"
```

Using NSString:

You can also get a substring by using the type casting operator `as!` to have Swift treat the string as an `NSString` and execute the `NSString.substringFromIndex` method:

```
var s = "HeyCharlie"  
var sub = (s as! NSString).substringFromIndex(2)  
print (sub) // Prints: yCharlie
```

And you can get the substring from the beginning of the string to a character pointed to by an integer index:

```
var s = "Hey Charlie"  
var sub = (s as! NSString).substringToIndex(4)  
print (sub) // Prints: HeyC
```

Warning: The numeric index refers to the sequence of 16-bit `NSString` words in the string, not Unicode's idea of a perceivable character. If characters that take more than 16 bits are involved, such as the hamburger emoji, and the index points to something other than the beginning of that character, you will get gibberish.

Or in Swift:

```
let str = "Hello, world!"  
let index = advance(str.startIndex, 4)  
str.substringFromIndex(index) // Returns String "o, world!"  
str.substringToIndex(index) // Returns String "Hell"
```

REPLACING SEQUENCES OF CHARACTERS IN A STRING WITH OTHER SEQUENCES

You can replace a sequence of characters (a substring) in a string with another sequence of characters:

```
var s = "Hey Charlie"  
var s2 = s.stringByReplacingOccurrencesOfString("Charlie",  
withString: "George")  
print(s2) // Prints: Hey George
```

Note that this will replace ALL occurrences of “Charlie” in the string with “George”:

```
var s = "Hey Charlie, meet the other Charlie"  
var s2 = s.stringByReplacingOccurrencesOfString("Charlie",  
withString: "George")  
print(s2) // Prints: Hey George, meet the other George
```

GETTING NUMERIC AND BOOLEAN VALUES FROM STRING

Using Swift Directly

If you have a string consisting of the digits representing an integer, you can get the integer value of it with:

```
var s = "12345"  
var number = Int(s)  
print(number) // Prints: Optional(12345)
```

The `Int` initializer returns an optional integer, rather than a pure integer. We haven’t discussed optional values yet much (see Chapter 9), but an optional integer is a type that can contain either an actual integer or no value, otherwise known as a `nil`. To actually use an optional, you have to “unwrap” it:

```
var s = "12345"  
var numberOptional = Int(s)  
print(numberOptional) // Prints: Optional(12345)  
if(let n = numberOptional) {  
    print("Value is \(n)") // Prints: Value is 12345
```

```
}

else {
print("No value, this is a nil")
}
```

This form of unwrapping is known as “optional binding” and unwraps it only if it has a value.

Using NSString

It’s also possible to use the `NSString` property `intValue`. This is simpler than the direct Swift approach, but it is not as safe:

```
var s = "12345"
var m = (s as! NSString).intValue
print(m) // Prints: 12345
```

If you have a string consisting of a floating point value, you can get it as a `Float` value:

```
var s = "3.8877"
var n = (s as! NSString).floatValue
print(n) // Prints: 3.8877
// n is inferred to be Float
```

And as a `Double` value:

```
s = "3.8877138786"
var n = (s as! NSString).doubleValue
print(n) // Prints: 3.8877138786
// n is inferred to be Double
```

CONVERTING A STRING THAT HAS A BOOLEAN VALUE

If you have a string consisting of a Boolean name, you can convert it to a Boolean type, which is quite a different thing, although it doesn't look any different when you print it out:

```
var s = "true"  
var p = (s as NSString).boolValue  
print(p) // Prints: true
```

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 7 exercises, go to
understandingswiftprogramming.com/7

8 Tuples

Why is it that you squirrels
pronounce the word *tuple*
“too-puhl”?

I dunno. Why do you
raccoons pronounce it
“tuhl-puhl”?

like “quadruple”

like “quintuple”



A *tuple* is a simple way to pass, or store, multiple pieces of data, particularly those of different types. The initial motivation for tuples was probably the need to pass more than one return value from a function, especially when the values are of different types. An array cannot be used in this situation because all of the values in a Swift array must be of the same type. But this is easy to do with a tuple.

Tuples are generally used to do simple things.

A tuple normally consists of at least two values—variables, constants, or other types—contained within parentheses. Thus:

```
var returnThis = (283, "Error: Bad syntax")
```

A tuple can contain any combination of types, and any number of types.

The code above defines a tuple that has two components, an integer and a string. Its type is `(Int, String)`, as we can see if we explicitly declare the variable:

```
var returnThis: (Int, String) = (283, "Error: Bad syntax")
```

A tuple can be accessed by position using dot syntax:

```
var errorCode = returnThis.0  
var errorText = returnThis.1
```

A tuple can also define names for its components:

```
var returnThis = (code: 283, text: "Error: Bad syntax")
```

The values can then be retrieved using these names with dot syntax:

```
var errorCode = returnThis.code  
var errorText = returnThis.text
```

Returning a Tuple from a Function

```
func getPlayerNameAndNumberOfHomeRunsThisYear() -> (name:  
String, runs: Int) {  
    var PlayerName = "Babe Ruth"  
    var PlayerRuns = 55  
    return (PlayerName, PlayerRuns)  
}
```

This shows both the use of a tuple to return mixed values from a function, as well as setting that tuple from a variable.

The function has no input values and always returns the same thing. The result is a tuple with a value of ("Babe Ruth", 55).

```
let playerNameAndRuns =  
getPlayerNameAndNumberOfHomeRunsThisYear()  
  
print(playerNameAndRuns.0)      // Prints Babe Ruth  
print(playerNameAndRuns.1)      // Prints 55
```

Empty Tuple

It's possible to have an empty tuple, defined like this:

```
var empty = ()
```

Single Tuple—NOT!

A tuple containing a single value is theoretically not defined and makes no sense, as the compiler just ignores (or presumably should ignore) the parentheses:

```
var singleTuple = ("Babe Ruth")
```

Switch Statements

Tuples can be used as literals to match cases in switch statements.

```
var tup = (45, "Don Drysdale")
switch(tup) {
    case (35, "Fernando Valenzuela"):
        print("Valenzuela with 35 home runs")
    case (44, "Babe Ruth"):
        print("Ruth with 44 home runs")
    case (45, "Don Drysdale"):
        print("Drysdale with 45 home runs")
    default:
        print("Can't find anything that matches")
}
```

Tuples can be used quite flexibly in switch statements. See Chapter 20, “The Flow of Control Revisited” for details.

Unusual Tricks with Tuples: Decomposition

The following will assign the values of 1, 2, and 3, respectively, to the variables a, b, and c:

```
var (a,b,c) = (1,2,3)
```

In other words, the above is equivalent to:

```
var a = 1  
var b = 2  
var c = 3
```

This is known as *decomposition*.

How to Pronounce the Word “Tuple”

How should the word “tuple” be pronounced? It can be correctly pronounced in two ways, as suggested by the last part of the words “quadruple” and “quintuple”. In other words, “too-puhl” or “tuh-puhl”. This is in the United States. British English speakers may pronounce it slightly differently.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 8 exercises, go to

understandingswiftprogramming.com/8

9 Optional Values



“That raccoon thinks he is a squirrel! I’ve never seen a raccoon climb that high up in a tree. Still, he’s not likely to fall. What’s really risky is all the forced unwrappings he’s got in his Swift code.”

A major source of errors in programming results from the situation in which a variable is expected to have a value but has in fact no value. The problem is not that this is particularly difficult to deal with. The problem is that it is often simply *not* dealt with. The code might be prototype code that suddenly is deemed to be production code. Or it is code where the programmer defers dealing with the problem and then forgets, or disappears to work on some other project. The consequences vary from some function simply not working in some circumstances to an app crashing. In some cases the problem only manifests itself at a later time and is difficult to debug.

It is common for iOS and Mac APIs to return a `nil` (that is, the absence of a value) when a value, such as the contents of a file from a remote server or on the device's own file system, is not available. Some functions will return `nil` rather than a value: For example, the function `toInt()` normally takes a number coded as a string (e.g., "1234") and converts it to an integer. However, if the function is given a string like "abcd", it will return a `nil`. Similarly, an attempt to access a dictionary with a key that does not exist will return a `nil`.

If *type safety* refers to the goal of designing a language that tries to help programmers avoid making errors about the correct data type, then the goal of designing a language that tries to help programmers avoid making errors when variables lack values might be called *value safety*.

Swift has a capability for quickly catching errors resulting from the absence of values. This capability is implemented in the form of what are known as *optional values*, or *optionals*.

Ordinary variables, known as *non optionals* in this scheme, are *required* to have values. They are also required to have an initial value. If an attempt is later made to set them to `nil`, a runtime error will result.

In order for a variable to be allowed to not contain a value, that is, be set to `nil`, that variable must have been declared as some form of "optional" data type. (Constants aren't involved in this discussion, because they can't change value and thus can't suddenly become `nil` unexpectedly.)

The fact that a runtime error will result from setting a non optional variable to `nil` and that `nil` can only be set to a variable declared with an optional data type might be considered a first line of defense (of two) in avoiding this kind of error.

The First Line of Defense—Only Optional Variables Can Not Have a Value

Every data type that can contain a value has an optional version of it that allows a variable of that type to not have a value.

Thus, an integer data type has an optional version formally defined as:

`Optional<Int>`

and known as an “optional integer”.

You won’t actually see a notation like `Optional<Int>` much. What you will commonly see is an alias (meaning the same thing) that consists of the type name followed by a question mark—in this case `Int?`.

Thus, the following code will declare a variable as an optional integer:

```
var n: Int? = 5
var n: Int? = nil
```

In the first line `n` must be explicitly declared as an optional with the `: Int?`. If `n` is simply declared with no type annotation and then set to 5, the type inferred will just be `Int`, a non optional.

In the case of the second line, an explicit declaration is also required, because although setting it to `nil` makes it clear that it is an optional, the compiler would otherwise not know that the type desired was an integer.

When a variable with an optional type is set to a value that value is *wrapped*. This means that it cannot be directly accessed in the normal way. To access it, it must be *unwrapped*.

The Second Line of Defense—Values in Optionals Must Be Unwrapped



**"Good thing we scanned that optional.
Another few minutes and that
squirrel on our team would have
unwrapped it and crashed the app!"**

The second line of defense in using optionals is the necessity to unwrap them before use. If `n` is an optional and there is an attempt to execute the following code, the program will have a runtime error.

```
var n: Int? = 5
var p = n // Program will crash
```

The rule is that a reference to an optional data type that is made without unwrapping it will cause a runtime error as a way of reminding the programmer that some effort must be put into ensuring that the value is actually there.

The simplest (but a dangerous) way to do this is by *forced unwrapping*. This is done by appending an exclamation point to the variable being unwrapped:

```
var n: Int? = 5
var p = n! // Works but dangerous
```

This works but is not particularly recommended and should only be used if you are *absolutely sure* that the value is not `nil`. If the value is `nil`, a runtime error will result.

The best practice here is to use what is known as *optional binding*:

```
var n: Int? = 5
if let p = n {
    print("The unwrapped value is \(p)")
}
else {
    print("There is no value")
}
// Prints: The unwrapped value is 5
```

What this does is to set `p` to the value contained in the wrapped value `n`. The value of `p` is then tested in the if expression: if the value exists then the test succeeds and the first part

of the if statement is executed; if there is no value the test fails and the second part of the if statement is executed.

Once the value of `p` has been set the value is available in an unwrapped form by referencing `p`. Thus, when it is referenced the value can be assigned or printed without any use of an exclamation point to unwrap it: it is already unwrapped.

This particular behavior is an idiom that only sets a value if the second variable is an optional type.

Note that this if statement and the ability to assign a value to a variable in it is against the general rule in Swift that assignment statements cannot be made in if clauses. This is allowed because the constant is declared within the if statement and thus its scope is local to it. It is thus relatively safe. The usual practice is to use a constant. It is also possible to use a variable, which can be useful in some circumstances. This behavior also works with while statements as well as if statements.

In optional binding it is common to use the same variable name in the binding clause as was used for the optional value.

```
var n: Int? = 5
if let n = n {
    print("The unwrapped value is \(n)")
}
else {
    print("There is no value")
}
```

Optional binding is a better practice than testing for `nil`, which I will describe just below, because it is a little safer. I'm just describing this alternative here for completeness so that you understand it. But you should use optional binding in preference to testing for `nil`.

This alternative simply tests whether the variable has a value or not:

```
var m: Int? = 5
if m != nil {
```

```
print("The value of m is \(m!)")
}
else {
print("m has no value")
}
// Prints: The value of m is 5
```

This is just a simple test of whether the variable has a value or not. If there is a value, it is necessary to do forced unwrapping (with an !) on the variable to extract the value. The reason that this is considered less safe than optional binding is that the programmer may accidentally omit the exclamation point and thus cause a runtime error.

In the present chapter I've covered the basics of optional values—why they make code safer, albeit at some cost of some annoyance to the programmer. I've described the reality that non optionals that get set to `nil` will crash, and that values stored in optional variables are wrapped and must be unwrapped before use. I've then described three ways of unwrapping—the potentially dangerous forced unwrapping, the best practice of optional binding, and the less safe alternative of just testing for `nil` before unwrapping.

In a later chapter, Chapter 26 on “Optional Values Revisited”, I will describe some of the less known aspects of optionals. This includes marking an optional as *implicitly unwrapped*, so that you do not have to unwrap it. It also includes the use of the `nil` coalescing operator, which forces any optional variable that is `nil` to have a default value. It includes the description of how to do optional chaining, which is a simplified syntax for unwrapping when you have dot syntax that includes multiple optional variables. And, finally, it includes a description of how, in Swift 1.2, a simpler syntax allows optional binding for multiple optionals to be done more easily.

Optionals Syntax

x: Int? Declaring a variable as an optional

x! Forced unwrapping an optional (used on a variable that has a value)

Testing an optional for nil

```
if x!= nil {  
    print("The value of x is \(x!)")  
} else {  
    print("x has no value") }
```

Optional binding

```
if let x = x {  
    print("The value is \(x)")  
} else {  
    print("There is no value") }
```

x: Int! Implicitly Unwrapped Optional

?? Nil Coalescing Operator

x? Optional Chaining Operator

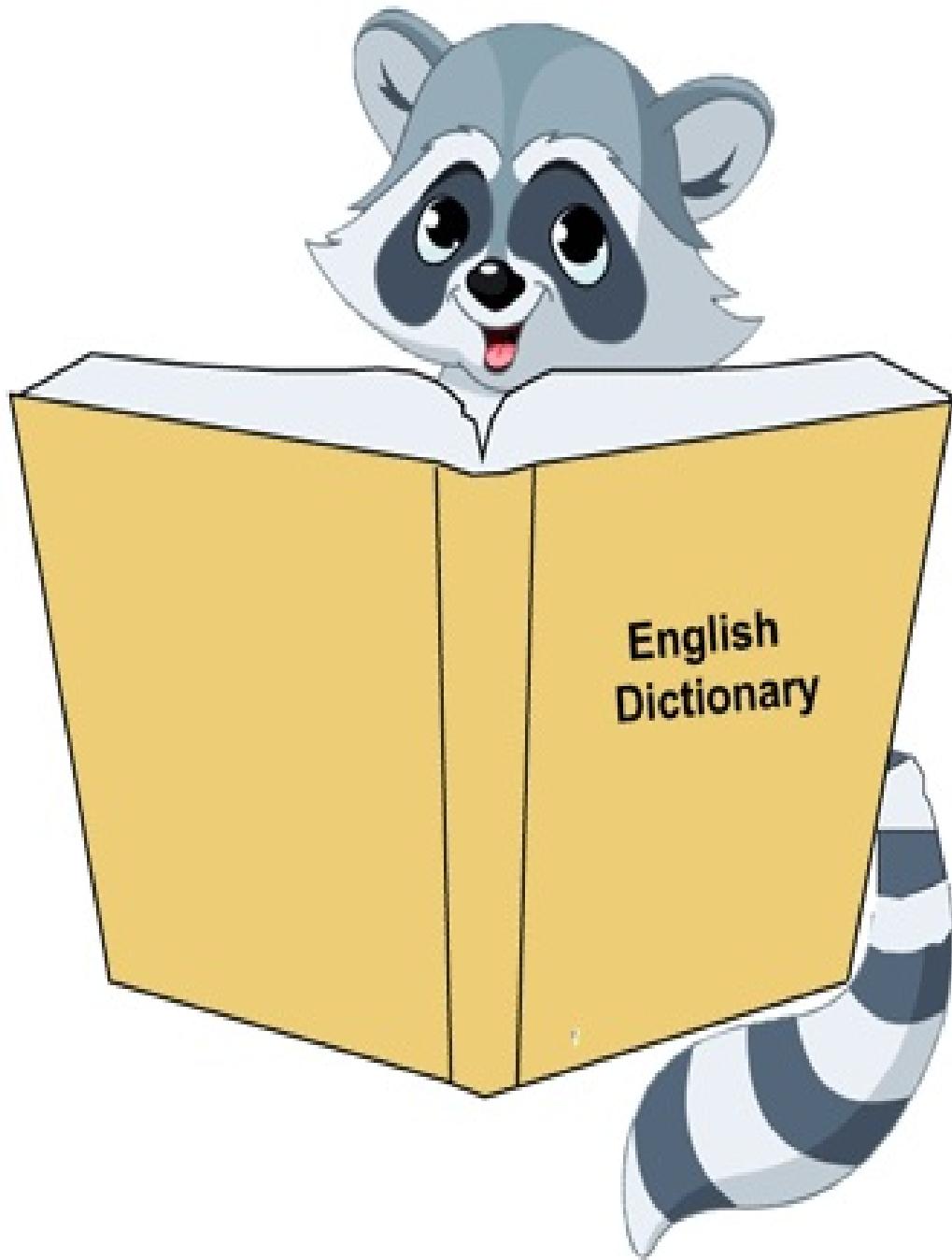
Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 9 exercises, go to

understandingswiftprogramming.com/9

10 Dictionaries



"This dictionary is so lame. It doesn't even have the word 'selfie'. We'll have to return a NIL. I hope the programmer didn't use forced unwrapping."

A dictionary in Swift stores pairs of keys and values in an unordered fashion. For example, a dictionary might store names of airports (“San Francisco”), each stored and retrieved by a key serving as an identifier that consists of the standard airport code (“SFO”). Every key in a dictionary must be unique.

All values must be of the same type, and all keys must be of the same type. The type for the values does not have to be the same as that for the keys.

A Swift dictionary that is a variable can be modified; if it is a constant it cannot be modified. This is analogous to the iOS API dictionaries used with Objective-C, where `NSMutableDictionary` can be modified and `NSDictionary` cannot be modified.

A dictionary may be initialized with a *dictionary literal*, as follows:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
```

The dictionary literal consists of one or more key-value pairs, with each key and value separated by a colon. The key-value pairs themselves are separated by a comma, and the entire set of key-value pairs is contained within square brackets.

The example above relies on inferring the type for the key and the value for the dictionary from the initializing information. The following has the same effect, but makes explicit the types:

```
var airports: [String: String] = ["SFO": "San Francisco",  
"OAK": "Oakland"]
```

A Swift dictionary that is a constant is initialized in just the same way as above, except that `let` is used instead of `var` to declare that it is a constant:

```
let airportsConstant = ["SFO": "San Francisco", "OAK":  
"Oakland"]
```

or:

```
let airportsConstant: [String: String] = ["SFO": "San
```

Francisco", "OAK": "Oakland"]

Retrieving a Value from a Dictionary: The Optionals Problem

Once data has been entered in a dictionary, it can be retrieved by providing a key for the desired value. If we, for example, want to retrieve the name of a city given the airport code, we can do so by:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
var nameOfCity = airports["OAK"]
print (nameOfCity)
// Prints: Oakland
```

There is one problem, however. It's possible, when retrieving a value from a dictionary, to get the value `nil` instead. This occurs when an attempt to retrieve a key-value pair is made but the key that is provided does not match a key in the dictionary.

Because of this possibility, the compiler will infer `nameOfCity` to be an optional type, (`String?`). Before that value can be used, it must be unwrapped.

The recommended way to unwrap the optional is to use optional binding, as follows:

```
if let nameOfCity = nameOfCity {
    print("The unwrapped value is \(nameOfCity)")
} else {
    print("There is no value")
}
// Prints: The unwrapped value is Oakland
```

For more details see the two chapters on Optionals, Chapter 9 and Chapter 26.

Operations on Dictionaries

To add a new key-value pair to the dictionary, we can do the following:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
airports["SJO"] = "San Jose"
```

This would result in the dictionary containing the following key-value pairs:

```
"SFO": "San Francisco"
"OAK": "Oakland"
"SJO": "San Jose"
```

Note that these pairs are in no particular order.

If we decide that we want to change the value in a key-value pair, we can do that with the same syntax:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
airports["SFO"] = "San Francisco International"
```

This would result in the following key-value pairs:

```
"SFO": "San Francisco International"
"OAK": "Oakland",
```

Again, these pairs are in no particular order.

If we decide that we no longer want the key value pair SJO: San Jose to be in the dictionary, we can remove it as follows:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland",
"SJO": "San Jose"]
```

```
airports["SJO"] = nil
```

This would result in the following:

```
"SFO": "San Francisco"  
"OAK": "Oakland"]
```

And again, these pairs are in no particular order.

This removes the entire key-value pair from the dictionary; it does not set the value associated with “SJO” to `nil`. (If you now try to access the value using the “SJO” key, you will get `nil` back, but not because “SJO” has a value stored of `nil`, but because “SJO” is unknown as a key.)

The number of key-value pairs in a dictionary can be determined by accessing the `count` property of the dictionary:

```
var numberOfItems = airports.count
```

The `isEmpty` property (a `Bool`) will tell you whether the dictionary has no elements (has a count of 0):

```
if(airports.isEmpty) {  
    print("The airports dictionary has no values in it")  
}  
  
else {  
    print("The airports dictionary has at least one value")  
}
```

You can create an empty dictionary in the following ways:

```
var airports: [String: String] = Dictionary()
```

```
var airports: [String: String] = [:]
```

You can replace a value in a dictionary with the following and have the old value returned so that it can be assigned to a variable:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
var oldValue = airports.updateValue("San Francisco International", forKey: "SFO")
print(oldValue) // Prints: Optional("San Francisco")
```

This not only replaces the value “San Francisco” with “San Francisco International”, but assigns the old value to the variable `oldValue`.

The method `RemoveValueForKey()` works similarly, in that as part of deleting a key and value from the dictionary it will return the old value.

The method `removeAll()` will remove all of the key-value pairs in a dictionary.

Dictionaries also have a `keys` property and a `values` property, which contain a collection of all of the keys or all of the values in the dictionary, respectively.

These have a type of `Swift.LazyBidirectionalCollection`, and to look at them you can use a for-in loop to iterate through them. To look at the values:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
for value in airports.values {
    print(value)
}
```

This will print:

San Francisco

Oakland

(Not necessarily in this order)

To look at the keys:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]
```

```
for key in airports.keys {  
    print(key)  
}
```

This will print:

SFO

OAK

(Again, not necessarily in this order)

Dictionaries, like arrays, are value types. (They are implemented as structures). That this means is that if you make a copy of a dictionary, and change an element, it will not change the original. Thus:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland"]  
var airports2 = airports  
airports2["SFO"] = nil  
print(airports) // Prints: [SFO: San Francisco, OAK: Oakland]  
print(airports2) // Prints: [OAK: Oakland]
```

In the Objective-C version of dictionaries, doing the same thing would delete SFO from both airports and airports2, since dictionaries in Objective-C (`NSMutableDictionary`) are reference types, and copying a dictionary is only copying the reference to it, not the actual dictionary.

We have already seen how to look at all of the keys, or all of the values, of a dictionary. We can also use a for-in loop to look at both the key and the value:

```
var airports = ["SFO": "San Francisco", "OAK": "Oakland",  
               "LAX": "Los Angeles"]  
  
for (name, airportCode) in airports {  
    print(name)  
    print(airportCode)  
}
```

This will print:

San Francisco

SFO

Oakland

OAK

Los Angeles

LAX

(And again, not necessarily in this order)

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 10 exercises, go to

understandingswiftprogramming.com/10

11 Object-Oriented Programming



“No, it's not true that object-oriented programming is mostly a lot of hot air. It's just that those silly humans make it sound that way. If it were up to us raccoons, we'd ban the word *polymorphism*.”

Although many of the readers of this book will be Objective-C programmers looking only to understand Swift, some will have programming experience but know little or nothing about object-oriented programming. This chapter is for those readers. If you are familiar with object-oriented programming, you may want to skip or just skim most of this chapter. However, everyone should probably read the section on polymorphism, which describes the different ways Swift uses this characteristic of object-oriented programming.

Computing began with the “stored program computer” in which instructions for how to manipulate bits were stored in memory. The first programming was in machine language, meaning that the programmer entered the ones and zeroes that defined the instructions directly into the computer by hand. Programs were sequences of steps that performed calculations and stored the result in memory. Assembly languages were then developed that automated the tedium of hand coding. Later, high level languages were developed that used named variables, and allowed statements like this:

```
var m = 5;
```

The same basic model was still used, though—some calculation followed by storing the result in memory, now often known as “changing state”, or changing the state of the computer.

This led to *procedural programming*, organizing code in subroutines, functions, or methods that allowed the same code to be executed multiple times in different contexts and thus reused. Efforts were made to organize these procedures as structured modules so as to avoid the dreaded “spaghetti code” that so often resulted. The procedures operated on data structures that were typically large and that often involved the entire program or large parts of it rather than specific modules.

Object-oriented programming is a quite different style of programming. The idea with object-oriented programming is to create *objects* that combine both data structures and procedures. Those objects are often intended to represent real-life entities, including both their data (in the form of what are known as *instance variables* or *properties*) and their behavior (in the form of *methods*.)

Objects (combining both data structures and code) served as the basic unit, or module, of computing, and considerable effort is made to isolate each object from other objects.

Twenty-five years ago object-oriented programming was only one contender among many styles of programming. But it was adopted seemingly overwhelmingly by industry—to the surprise of many academics. However, despite its seeming dominance, studies comparing

object-oriented approaches with more traditional procedural and modular methods haven't demonstrated significant gains. (It is harder and harder to do these kinds of comparison studies because the object oriented approach is now so dominant.)

According to the TIOBE Programming Community Index (tiobe.com) for August, 2015, the C language, which is not object-oriented, is the second most popular programming language. But nearly all of the top ten, namely Java, C, C++, C#, Python, Objective-C, PHP, Visual Basic .NET, JavaScript, and Perl are all at least in theory object-oriented. Many of these were not originally object-oriented languages, but object-oriented capabilities were added because of the popularity of the approach. (Objective-C ranked #6, down from its peak of #3, while Swift was at #17.)

As winners of battles tend to write their history, you will read a lot of nonsense written by proponents of object-oriented programming. You might get the impression that it is impossible to build a large scale software system without the use of object-oriented programming (many successful large scale systems have been built with other kinds of languages.) You might get the impression that before object-oriented programming, nobody ever broke programs down into smaller modules—it was just one big pile of spaghetti. You might get the impression that object-oriented programming is an advantageous approach for *every* programming problem.

For building apps for devices like the iPhone, iPad, iPod Touch, Apple Watch, and the Macintosh, object-oriented programming is actually a pretty good choice. This is not because it is necessarily a good way to design the business logic in apps (it often helps, and is unlikely to hinder). But more important, it is because iOS (and the Mac) have very large libraries—Foundation and Cocoa Touch in the case of iOS—that are organized as classes. And as the developers of Smalltalk found—when the first GUI (Graphical User Interface) oriented user interfaces were built in the 1970s at Xerox Palo Alto Research Center—object-oriented programming is a very good way to build these kinds of user interfaces.

Despite the value of the existing class-based libraries, there is a growing recognition that classes are often not a good way to solve certain programming problems, and Swift has itself been designed to make substantial use of alternative approaches, including using structures and protocols rather than classes. See Chapter 34 on “Protocol Oriented Programming”.

I went into some detail earlier about the strong emphasis that programming over the decades has placed on storing results to memory, that is, “changing state”. This may be so seemingly necessary to today’s programmers as water might be to a fish—so obvious that programmers, or fish, don’t even think about it. But there is a very different style of programming now becoming more popular that deliberately does *not* change the state, or

memory, of the computer. There are some significant advantages to this style, known as *functional programming*. Swift has been influenced to an extent by functional programming, and you can actually do some functional programming in it. The principal pure functional language, Haskell, is now at #41 in the TIOBE list. Swift may be a good chance for functional programming to gain at least some traction.

However, Swift is primarily an object-oriented (and, arguably, protocol-oriented) language. There are three principles of object-oriented programming that have been considered fundamental to object oriented languages. These are *encapsulation*, *inheritance*, and, arguably, *polymorphism*.

Encapsulation



**“Pete, maybe you are taking
this encapsulation thing a little
too far.”**

Encapsulation bundles together information and behavior in one component, or object, and object-oriented programming languages provide mechanisms for this bundling and for isolating the object from the rest of the world.

Typically, methods within an object can operate on the data in the same object. However, there are limits on what another object, outside of the object in question, can access. In general, direct access to data in an object is not allowed, with outside objects required to execute so-called getter and setter methods of the bundle to read or write information, even when this indirect access is allowed.

Often, the ability of the rest of the program to access properties or methods within an object is carefully controlled with the intent of minimizing that access to what is absolutely required. This is typically done as a general practice to prevent errant code from causing damage.

It is also a common practice to provide other objects with a specific interface that does not change, while hiding the details of the implementation within an object from outside objects. This allows code within objects to be rewritten to fix bugs, make the code run faster, or be more general without “breaking” outside code that depends upon it.

Swift includes mechanisms to enforce encapsulation, including the requirement of accessing properties of objects only with dot syntax (which requires getters and setter methods) and having access control keywords (e.g., `private`, `public`) that, along with other mechanisms, control access to data and methods for particular objects.

Inheritance

Swift, Objective-C, and most other object-oriented languages have *inheritance*. As part of the basic dividing up of the world into objects, entities called *classes* are defined. Objects are created based on these classes, which serve essentially as blueprints that define what data structures the objects contain, and what methods the objects are allowed to use. Each object has its own copy of the data structure so created. An object that has such a copy is known as an *instance* of the class. Thus, if the class is `Apple` and it has a property defined named `color`, multiple objects can be created using the `Apple` class blueprint. Some might have their `color` property stored as “red”, while others may have their `color` property stored as “green”.

When a new class is created, it can define itself as being a *subclass* of another existing class. If it does this, it will inherit all of the data structures (properties) of the class it is a subclass of. (The original existing class is known as the subclass’s *superclass*.) It will also inherit all of the methods of its superclass. Often, a superclass will have inherited data structures and methods from its own superclass, and that superclass’s superclass, and these will also be inherited by a new subclass. This is part of what makes the iOS and Mac APIs so powerful. They are organized into classes that form a hierarchy. A new subclass can access large amounts of functionality because of inheriting, directly and indirectly, data structures and methods.

This makes it easy to reuse substantial amounts of code. And it also makes maintenance easier and more productive. If a bug is fixed in a method at a level high in the hierarchy, it will be fixed for all of the methods below it in the hierarchy. (It is also true that if a bug is *introduced* into a method high in the hierarchy, it will appear in all of the methods below it in the hierarchy.)

There are some downsides to inheritance. Although much of the effort in object-oriented programming goes into making objects largely independent from each other and only “loosely coupled”, it has been increasingly recognized that inheritance often couples objects that have a subclass-superclass relationship awfully tightly. Even twenty years ago this was recognized and advice given to “favor composition over inheritance” in the influential book *Design Patterns*. This meant that composing classes out of other classes without inheritance was often desirable. And other criticisms have been offered, leading up to Swift being designed with as much of an eye to noninheritance approaches as to inheritance. See Chapter 34 on “Protocol Oriented Programming.”

Swift has *single inheritance*, meaning that it can only direct inherit from one superclass. Only one popular programming language, C++, allows *multiple inheritance*, or the ability to directly inherit from more than one superclass. The problem is that there are conceptual

problems with multiple inheritance. For example, suppose classes B and C both inherit from class A, but B overrides one of the inherited methods with an implementation of its own. And suppose further that class D inherits from both class B and class C. Which version of the method that was overridden should class D inherit? The one that was overridden from B or the one that was not overridden from C?

Some object-oriented languages do not have inheritance, but are based on *prototypes*. A new class can be defined as a *clone* of a given object, and is given copies of its properties and methods. However, in prototype-based languages like JavaScript it is possible to organize things so as to create what are effectively classes and inheritance.

Some apps may have substantial business logic that might be usefully organized by the use of classes that inherit from each other. Other kinds of business logic may not fit easily into this structure.

Polymorphism

Gray fur



Brown fur



“Would you believe that me and my brother here are identical twins? If that isn't polymorphism, I don't know what is.”

It is commonly said that to be object-oriented, a programming language must have encapsulation, inheritance, and polymorphism. Requiring encapsulation is reasonable: it is fundamental to the approach. Requiring inheritance is reasonable, especially if languages like JavaScript that are based on prototypes rather than classes are included, given that this is a similar approach.

Polymorphism is more controversial. It's a very fancy word that seems to cause as much confusion as clarity and feeds the suspicion of critics of object-oriented programming that the approach is as much fluff as substance.

Matt Gallagher—hardly a critic of object-oriented programming—on his blog *Cocoa With Love*, says that “the term is so abstract as to be worthless” and “only serves to obfuscate and confuse”. Gallagher doesn’t have an issue with the concept itself, but with the term, saying that other words that better describe specific situations should be used instead.

The term *polymorphism* was apparently first used in biology, where it refers to two different forms of the same organism occurring simultaneously. Thus, most jaguars in South America are tan/orange in color, while about 6% are dark, nearly black, though they have the same genetics. The term is from the Greek language and basically means “many forms or shapes.”

If you look up the topic in Wikipedia, you will read:

If a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations, it is called ad hoc polymorphism. [From wikipedia.org, Polymorphism (computer science). Retrieved 2015.]

This is more commonly known as *function overloading*.

The same source discusses *parametric polymorphism*, which turns out to mean *generic programming*.

So I agree with Gallagher that the word isn't very helpful, and I will try to avoid using the term *polymorphism* and its variations in the rest of this book, relying on the more specific and concrete terms instead. Apple's documentation on Swift in fact doesn't use the word at all; nor do most books on Swift.

Swift does have many aspects that make use of polymorphism, many more than Objective-C, which actually makes rather minimal use of it. I've identified six aspects of Swift that seem to qualify, and I have described them below:

METHOD OVERRIDING

Often, a class will deal with a general concept, and subclasses of it with more specific versions. Thus, a class may exist named `Shape`, with subclasses `Rectangle` and `Circle`. All may include a method named `draw` that can draw the shape. The `draw` method in `Rectangle` is created by inheriting, and then overriding (changing the code) in the `draw` method inherited from `Shape` to what is necessary to draw a rectangle.

The system must then choose which version of `draw` will be executed based on the type of the particular instance.

SUBTYPING

In Swift an array can contain only instances of the same type. In fact, the compiler and runtime will allow either instances of a given type or an instance of a subclass of that type. If you try to initialize an array by providing a list of values that are not instances of subclasses or superclasses of each other but have a common ancestor, the compiler will infer the type of the array to be the type of that common ancestor, and then allow all of the values to be placed into the array because they are instances of subclasses of the common ancestor.

Allowing an object of a given type to be contained in a variable typed as its superclass is known as *subtyping*, or *subtype polymorphism*. (The reverse is not allowed. An object of a given type cannot be stored in a variable that has the type of the object's subclass.)

OPERATOR OVERLOADING

The same operator can be used in different contexts, with each context having a particular set of types of operands. For example, the “+” operator is used for addition with integer types, but, when used with strings, causes different code to be executed that results in concatenation of the strings. For details, see Chapter 25, “Operators Revisited: Custom Operators and Operator Overloading.”

FUNCTION OVERLOADING

Functions can be defined that have the same name but with different combinations of types for input parameters and return values. The compiler is smart enough to call the appropriate code in a given situation, based on the types that are used in the call to the function. Overloading is also often used with initializers in Swift (which are similar to but not quite the same as functions or methods).

SUBSCRIPT OVERLOADING

The use of a subscript in a particular context is defined much like a function, with input parameters and a return value that have specific types. If multiple subscript definitions are included in a particular class, structure, or enumeration, the compiler will select the proper code based on the type.

GENERIC PROGRAMMING

In generic programming (see Chapter 28 on “Generic Programming” for details) code is written that uses an abstract symbol (such as T) wherever a particular type would otherwise be used. This allows a programmer to write a single piece of code rather than multiple pieces of code that each handle a different type or pattern of types. For example, a programmer might have to write one function to do a calculation with an integer type, and a second function to do the same calculation with a floating point type. By using generic programming, a single piece of code is written with abstract symbols replacing the names of particular types. The compiler creates the object code for different functions based on the types needed and sets up the function call so that it calls the appropriate version of the function for the particular type involved when it is run.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 11 exercises, go to
understandingswiftprogramming.com/11

12 Classes, Objects, and Inheritance

1.

Apple did a good thing when they made these new class-like things—structures and enumerations—but it made everything very confusing—we don't know what to call them.

2.

I agree. They use the word **type** when they want to say “class, structure, or enumeration”—but **type** means something else. They need a new word. Here's some possibilities.

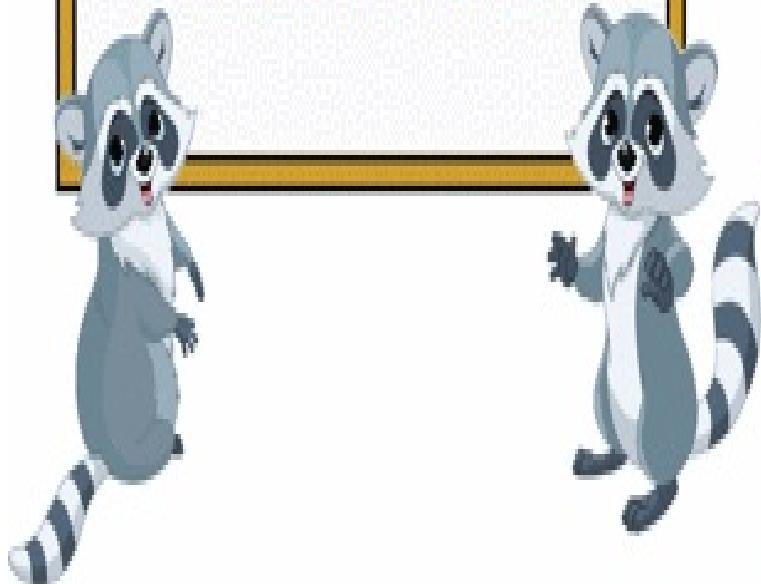
Class use keyword **class**

Structure use keyword **static**

Enumeration use keyword **static**

metatype **csetype**

instance creating type



Send ideas for new words to understandingswiftprogramming@gmail.com

Swift is a conventional object-oriented language that encapsulates data (in the form of *properties*) and behavior (in the form of *methods*) together in individual *objects*.

Swift is based on *classes*. A class is a sort of blueprint for creating objects, or *instances* of a class.

The definition of a class includes the definition of properties, which are variables and constants associated with the class. A property uses the same syntax as a variable or constant and is associated with a class when the property is declared within that class's definition. In Swift a property, which can be accessed from outside the class with dot syntax, is not any different from what in other object oriented languages is known as an *instance variable* and can only be accessed from within the class.

A definition of a class also typically defines methods, which are functions associated with a class.

Objects in Swift have *inheritance*, in which a class, and its objects, inherit the properties and methods of another class. Swift has *single inheritance*, meaning that an object can inherit from only one other class, known as its *superclass*. (A class also inherits all of the properties and methods that *its* superclass has inherited.) A class that is derived from a superclass is known as its *subclass*.

Swift classes and their objects have many capabilities. In addition to the usual *instance properties*, which are associated with each instance of the class, Swift classes can also have *class or type properties*, which are associated with the class itself rather than any instance. (Note: Class/type properties don't yet work for classes as of Swift 1.2.) They can have *computed properties*, which calculate values rather than retrieve them from memory. Swift classes can have *lazy properties*, for which values are not created until they are needed. And they can have *property observers*—code that is executed when a value is stored to a property). These additional properties and related capabilities are described in a later chapter, Chapter 16 on “Classes Revisited: Properties”.

When a new instance of a class is created, it must be *initialized*. This chapter will describe how this is done in the simplest cases, but initializing classes can get quite complicated, particularly when classes inherit from other classes. Chapter 29 on “Classes Revisited Again: Initializers” describes these in detail.

Classes can also define custom subscripts, allowing information to be read or written to

like this: `johnsTenFavoriteMovies[3]`. See Chapter 24 on “Custom Subscripts” for a description of how this works.

Objects that are created based on a class definition, that is, instances, are stored (along with their properties) in the heap part of random-access-memory as a reference type. That means that when they are accessed—assigned to a variable or passed as a parameter in a function or a return value of a function—only a reference is created to the stored information. There is only one copy kept, and if one entity that references it makes a change to the information, other entities that reference it will see that change. The alternative is a value type, which is used by structures. For more details about how this works see Chapter 14 on Structures.

Defining A Class

An example of the syntax needed for defining a new custom class that includes a few properties and a method is as follows:

```
class Fruit {  
    var skinColor:String = "red"  
    var fleshColor:String = "white"  
    var countryGrownIn = "United States"  
    var hasSeeds = true  
  
    func sayWhereItWasGrown () {  
        print("This fruit was grown in: \(countryGrownIn)")  
    }  
}
```

The name of the new class is `Fruit`. This class does not inherit from any other class, and as such is known as a *a base class*.

By convention (though not required), the first letter of a class name is in upper case and the remainder of the name is in camel case. (If the name was `FruitCake` the camel case would be obvious.) Also by convention, the first letter of a property or method name is in lower case and the remainder of the name is in camel case.

Properties can be either variables or constants, and are defined using the keywords `var` and `let` just like variables and constants are if they are not defined in a class. Types can be explicitly defined with type annotation, or they can be inferred.

A class definition often contains an initializer, code that sets the initial values of properties when a new instance is created. In this case we don't need an initializer because we have set the initial values of properties where they are declared. We'll discuss initializers in a later chapter: Chapter 29 on Classes Revisited Again: Initializers.

This class holds information about different kinds of fruit that you might have in your kitchen, including their skin color, flesh color, whether they have seeds, and the country they were grown in.

We can create an instance of the class representing a particular piece of fruit by specifying the class name followed by a pair of empty parentheses:

```
let thisPieceOfFruit = Fruit()
```

We first declare a constant `thisPieceOfFruit`. We then call `Fruit()` to instantiate, that is, create, a new instance of, the class `Fruit`, based on the class definition (blueprint for an object) and assign a reference to that instance to `thisPieceOfFruit`. This new instance is an object.

Accessing the Properties of a Class

We can then access the properties associated with this instance (object) by using this constant and dot syntax.

Thus,

```
var colorOfFruit = thisPieceOfFruit.skinColor  
print("The outside color of this piece of fruit is \  
(colorOfFruit)")  
  
// Prints: The outside color of this piece of fruit is red
```

If we discover that this particular piece of fruit is green on the outside, we can change its skin color with dot syntax:

```
thisPieceOfFruit.skinColor = "green"
```

And if we run these same lines of code again we'll get a different result:

```
colorOfFruit = thisPieceOfFruit.skinColor  
print("The outside color of this piece of fruit is \  
(colorOfFruit)")  
  
// Prints: The outside color of this piece of fruit is green
```

We can also create two instances of the class `Fruit`, and set each to a different color:

```
let pieceOfFruit1 = Fruit()  
let pieceOfFruit2 = Fruit()  
pieceOfFruit2.skinColor = "yellow"  
print("Colors are \ (pieceOfFruit1.skinColor) & \  
(pieceOfFruit2.skinColor)")
```

```
// Prints: Colors are red & yellow
```

(The first instance was set to red as part of initializing the instance, while the second was specifically set to yellow.)

Each instance has its own copy of the instance property `skinColor`, and can store a different value in that property.

Using the Methods of a Class

We can execute the method associated with this class by specifying it with dot syntax:

```
let thisPieceOfFruit = Fruit()  
thisPieceOfFruit.sayWhereItWasGrown()  
  
// Prints: This piece of fruit was grown in: United States.
```

A method of a class (that is, an instance method) is executed by specifying the name of a variable or constant that contains a reference to the instance, a dot, the name of the method, and a pair of parentheses.

The method accesses the copy of the properties associated with the particular instance that called the method.

A New Class is a New Data Type

The new class we have defined is a new data type named `Fruit`, and we can use it like any other data type in Swift.

We earlier created an instance of the class as follows:

```
let thisPieceOfFruit = Fruit()
```

We didn't discuss the data type of the constant that was created, but in fact the constant `thisPieceOfFruit` was inferred to be of type `Fruit`.

If we like we can explicitly set the type:

```
let thisPieceOfFruit: Fruit = Fruit()
```

We can also create two instances, that is, two objects of the class `Fruit`, and put both into an array:

```
let pieceOfFruit1 = Fruit()
let pieceOfFruit2 = Fruit()
pieceOfFruit2.skinColor = "green"
let arrayWithFruit: [Fruit] =
[pieceOfFruit1, pieceOfFruit2]
```

Here we have explicitly declared the constant `arrayWithFruit` to be an array that consists of objects of the class `Fruit`, and then initialized the array with different objects of that class. (I changed the `skinColor` property of the second instance to "green" so that the array would have objects that were truly different.)

The point here is that objects that you create with a new class that you have defined have their own data type as defined by that class, and this type is used when you manipulate these objects, such as by putting them into an array.

Inheritance

Now let's say we are interested in the banana we have in our kitchen. The class above is for fruits in general, and assumes that every fruit is red in color and was grown in the United States. (In a later chapter we will discuss better ways of initializing instances.)

We could just create a new instance of the class `Fruit` and then set the `skinColor` property to "yellow" and the `countryGrownIn` property to where it was grown.

But suppose we have lots of bananas, some purple. We'd like to have a class where the default color is yellow and the default country is somewhere that actually grows bananas. We'd also like to set the default value of the `hasSeeds` property for this new class to `false`, if most of the bananas we will be dealing with are the consumer variety, which do not have seeds.

We can create a class named `Banana` that is a subclass of the existing class, `Fruit`.

The following is a simplified version of the `Fruit` class:

```
class Fruit {  
    var countryGrownIn = "United States"  
    func sayWhereItWasGrown () {  
        print("This fruit was grown in: \(countryGrownIn)")  
    }  
}
```

And now its subclass, `Banana`:

```
class Banana: Fruit {  
}
```

We can see in this simple case how the inheritance of both properties and methods work. We can create an instance of the class `Banana`:

```
var aBanana = Banana()
```

This should have not only created an instance of the class Banana, but this instance should have inherited the property `countryGrownIn`. We can see if this happened:

```
print("Value of aBanana.countryGrownIn is \  
(aBanana.countryGrownIn)")  
// Prints: Value of aBanana.countryGrownIn is United States
```

Not only did the new instance of Banana inherit the property `countryGrownIn` from its superclass `Fruit`, but it inherited a value. What happened here is that as part of the default initialization of `aBanana`, a call was made to initialize the properties of the superclass `Fruit`.

This tells us that instances of subclasses like `Banana` can in some circumstances inherit not only the properties of their superclasses, but also their values.

We can create an instance of the class `Fruit`:

```
var aPieceOfFruit = Fruit()
```

And we can change the value of the property `CountryGrownIn` for both the fruit and the banana:

```
aPieceOfFruit.countryGrownIn = "Canada"  
aBanana.countryGrownIn = "Panama"
```

And see what happens:

```
print("Value of aPieceOfFruit.countryGrownIn is \  
(aPieceOfFruit.countryGrownIn)")  
// Prints: Value of aPieceOfFruit.countryGrownIn is Canada  
  
print("Value of aBanana.countryGrownIn is \  
(aBanana.countryGrownIn)")  
// Prints: Value of aBanana.countryGrownIn is Panama
```

```
(aBanana.countryGrownIn)" )  
// Prints: Value of aBanana.countryGrownIn is Panama
```

This tells us that when a subclass creates a new instance, that instance inherits a property from its superclass and its initial value. But thereafter the values of these properties are independent.

In addition to inheriting the property `CountryGrownIn`, the subclass `Banana` also inherits the method `sayWhereItWasGrown`. We can verify that this works as follows:

```
aBanana.sayWhereItWasGrown()  
// Prints: This fruit was grown in Panama
```

Overriding a Method of a Superclass

In the example above, we have a method `sayWhereItWasGrown()` in the superclass `Fruit`. If we create an instance of `Fruit`, we can execute that method, which prints the value of the property `countryGrownIn`:

```
let aPieceOfFruit = Fruit()  
aPieceOfFruit.sayWhereItWasGrown()  
// Prints: This fruit was grown in United States
```

If we create an instance of `Banana`, we get the same thing, because the method `sayWhereItWasGrown` is inherited:

```
let aBanana = Banana()  
aBanana.sayWhereItWasGrown()  
// Prints: This fruit was grown in United States
```

We can if we wish *override* an inherited method. So we can, for example, provide the following definitions of `Fruit` and `Banana`:

```
class Fruit {  
    var color: String = "red"  
        var countryGrownIn = "United States"  
    var hasSeeds = true  
    func sayWhereItWasGrown () {  
        print("This fruit was grown in: \(countryGrownIn)")  
    }  
}  
  
class Banana: Fruit {  
    override func sayWhereItWasGrown () {  
        print("This BANANA was grown in: \(countryGrownIn)")  
    }  
}
```

```
}
```

The definition of the class `Banana` only does two things: (1) Defines a new class `Banana`; and (2) Provides a method that overrides, that is, replaces, the method that it inherited from its superclass `Fruit`.

If we create an instance of the class `Fruit`, we can see that the original method is still there:

```
aPieceOfFruit = Fruit()  
aPieceOfFruit.sayWhereItWasGrown()  
// Prints: This fruit was grown in: United States
```

If we create an instance of the class `Banana`, we can execute the overridden method:

```
aBanana = Banana()  
aBanana.sayWhereItWasGrown()  
// Prints: This BANANA was grown in: United States
```

Note that the keyword `override` is used before the `func` keyword in the definition of the subclass's version of the function. This is required for redefinitions of functions that are inherited. If it is not included you will get a compiler error. Furthermore, if it *is* included and the function was not inherited (because it was not defined in a superclass), you will also get a compiler error. These are both safety checks that are, for example, not in Objective-C.

This chapter has described the basic aspects of how classes work, including how to define classes, create instances, how to use (instance) properties and (instance) methods, and how inheritance works.

These are the most-used aspects of classes. There are also some other kinds of properties that are used a little less, as well as an alternative type of method, the class method. These are described in a later chapter, Chapter 16, on “Classes Revisited: Properties and Class or Type Methods.”

In addition, initialization, particularly when inheritance is involved, can get quite complex.

In the example we have seen, we just initialized properties directly in their declaration, and depended on default behavior to initialize the superclass when necessary. But it often gets more complicated. This is described in detail in Chapter 29, on “Classes Revisited Again: Initializers.”

	Classes	Structures	Enumerations
Memory Type	Reference	Value	Value
Has instances	✓	✓	✓
Inheritance	✓		
Type casting	✓		
Conform to protocol	✓	✓	✓
Can be extended	✓	✓	✓
Deinitializer	✓		
Instance methods	✓	✓	✓
Class (type) methods	✓	✓	✓
Stored instance properties	✓	✓	
Stored class (type) properties		✓	✓
Computed properties	✓	✓	✓
Lazy properties	✓	✓	✓
Property observers	✓	✓	✓
Respond to notifications	✓		
Custom subscripts	✓	✓	✓
Memberwise initializer		✓	

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 12 exercises, go to
understandingswiftprogramming.com/12

13 Functions and Methods

Functions, according to Apple, are “self-contained chunks of code that perform a specific task.”

This definition is very much the same as what Apple uses for another entity in Swift, that of a *closure*. Functions and their siblings, *closure expressions*, are actually both specific cases of closures. The main difference is that functions have names while closure expressions do not, and each has its own syntax.

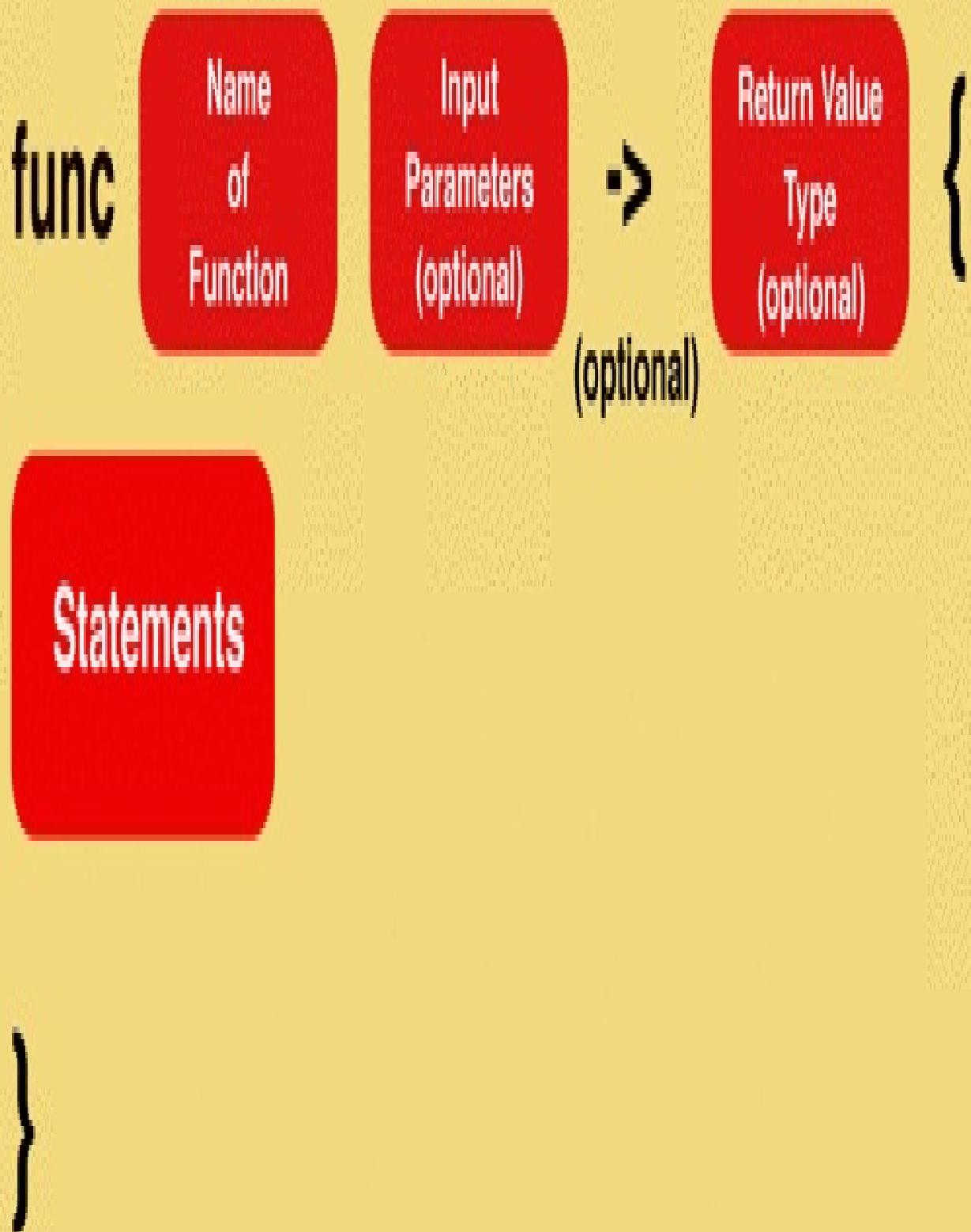
This chapter describes the fundamentals of functions and of methods. The latter are functions associated with, and defined within, a class (or structure or enumeration).

A later chapter, Chapter 17, “Functions Revisited: First Class Citizens” will describe how functions in Swift can be assigned to variables, passed as input arguments to other functions, and returned from functions. The chapter also describes how functions can be nested within other functions and how nested inner functions can capture variables and the like and be passed to a global environment and then executed.

Another chapter, Chapter 18, “Closures and Closure Expressions”, will describe closures and closure expressions, and how functions are related to these.

Still another chapter, Chapter 22, “Functions Revisited Again: Input Parameters”, will describe the wide variety of input parameter syntax that is allowed for functions.

The chunk of code that is in the body of a function is given a name, which is used to call the function when the task is to be performed, and the function has a syntax that allows for passing the values of parameters to the code and for the code to return a value. The general syntax of a function is shown in the figure below:



New in Swift 2, there is no difference between the syntax of a method and that of a function. The preferred form is to name it so that the name of the function or method suggests what the first parameter is. Starting with the second parameter, all parameters must have external names.

A typical example of a Swift function is:

```
func addTwoNumbersWithFirst (first: Int, second: Int) ->
Int {
    var c = first + second
    return c
}
```

This function, which adds together two integers, is called as follows:

```
var d = addTwoNumbersWithFirst(2, second: 5)
print(d) // Prints: 7
```

The syntax of a function includes the keyword `func` followed by the name of the function. Input parameters, if used, are contained within parentheses. The parentheses are required even if there are no input parameters.

This approach in Swift of calling functions and methods using external parameter names is pretty clearly motivated by the way that methods are named and parameters passed in Objective-C. Here's an example of the `addTwoNumbers` method as it might be called in Objective-C:

```
int d = [math addTwoNumbersWithFirst: 2 second: 5] //  
Objective-C syntax, not Swift
```

The important part of this Objective-C method call is the `addTwoNumbersWithFirst: 2 second: 5` part of it.

Methods in Objective-C are typically named so that the method name makes sense as a name not only for the method but for the first parameter. (This sometimes results in some fairly strange method names.) Any additional parameters have names just for those

parameters. Thus we have `addTwoNumbersWithFirst`, a single name that both calls the method and provides a name for the first parameter, which comes just after a colon following the method name. A space follows, then a second parameter name, a colon, and a second parameter.

The idea is that this makes the method call readable as something like English: “add two numbers, with the first being 2 and the second being 5”.

If you aren’t familiar with calling methods with external parameter names, this may seem pretty strange. But they make code very readable, and, in particular, they prevent errors resulting from mixing the order of the argument values. (This isn’t so readily seen in the example, because obviously it makes no difference when adding numbers which one comes first, but there are many calculations where it is important which value goes first.) Note that when you use parameters with external names, you still have to provide the arguments in the correct order—the names are just to make it easy for you to do this.

This is arguably a somewhat clumsy aspect of the Swift language definition. The Objective-C version always has argument values coming right after a method or parameter name, while the first argument value in the Swift version comes after a left parentheses. This is something programmers have to get used to. But the named parameters are a nice aspect of Objective-C, and it is useful for Swift to use some form of it.

(In Swift 1, external names are not required, and cannot be used, in a function unless specifically defined in the function.)

There are many variations for the syntax for input parameters. In the variation shown here, each input parameter has a name for the parameter, used to refer to it within the function, and the type of the parameter. This “local” name is also used in the function call, but only for the 2nd and subsequent parameters (and only in Swift 2).

The most unusual aspect of the syntax comes right after the input parameters and before the left brace indicating the beginning of the body of the function. We have a “->” character sequence (known as a *return arrow*), followed by a type. The latter indicates the data type of the return value of the function (if there is one).

This is quite different from the syntax of a function in C. In C there is no `func` keyword, there is no `->`, and there are no external parameter names.

I've shown this same function below, together with what it would look like if coded in C, to easily compare the two:

```
func addTwoNumbersWithFirst (first: Int, second: Int) ->
Int {      // Swift version
    var c = first + second
    return c
}

int addTwoNumbers (int a, int b) {
// C version
    var c = a + b;
    return c;
}
```

Both are called in a similar way, although (in Swift 2) names are used for the second and subsequent parameters.

No Input Parameters or No Return Value

It is often the case that you have no input parameters—you just want the function to *do something*. In this case there are no specified parameters but a () is included in both the function definition and the call:

```
func takeAction () -> Bool {  
    // Do something and get a response  
  
    let response = true // Worked ok  
  
    return response  
}  
  
let m = takeAction()
```

Sometimes you want to do something and not even provide a response. There is no return value. In this case both the return arrow (->) and the type after it are omitted:

```
takeAction2 () {  
    // Do something and do not provide a response  
}  
  
takeAction2()
```

Functions That Return Multiple Values

We can use a tuple to return multiple values from a function. We saw this briefly in the chapter on tuples; you can now see it again with more knowledge about what functions are supposed to look like:

```
func getErrorCodeAndText (a: Int) -> (Int, String) {  
    var errorCodeAndText: (Int, String) = (0, "") // Init  
    if (a == 0) {  
        errorCodeAndText.0 = 111  
        errorCodeAndText.1 = "Error: Disk out of space."  
    }  
    else {  
        errorCodeAndText.0 = 333  
        errorCodeAndText.1 = "Error: Disk not in drive."  
    }  
    return errorCodeAndText  
}  
  
var d = getErrorCodeAndText(0)  
print("Error code is \u202a(d.0) and Message is \u202a(d.1)")
```

The type for the return value is `(Int, String)`. That is, it is a compound type that is a tuple that includes values of `Int` and `String`.

Methods

A method, as discussed earlier, is simply a function that is associated with, and defined within, a class, structure, or enumeration. In Swift 2, the syntax for a method is exactly the same as it is for a function. There are two kinds of methods: *instance methods*, and *class or type methods*.

INSTANCE METHODS

An instance method is associated with a particular instance of a class, and is called from that instance. It can access the values of the stored properties of classes and structures that are associated with that instance.

Suppose we have a class called `Dog` that has three instance properties:

`numberOfFirstPrizeRibbons`, `numberOfSecondPrizeRibbons`, and `totalNumberOfPrizeRibbons`.

We might create an instance of `Dog` that represents a particular dog:

```
var particularDog = Dog()  
particularDog.numberOfFirstPrizeRibbons = 3  
particularDog.numberOfSecondPrizeRibbons = 2
```

And, assuming that we have defined the method

`addTwoNumbersWithFirst:second:` within the class definition of `Dog`, we can use it to calculate the value of the property `totalNumberOfPrizeRibbons`:

```
particularDog.totalNumberOfPrizeRibbons =  
particularDog.addTwoNumbersWithFirst(particularDog.numberOfFirstPrizeRibbons,  
second: particularDog.numberOfSecondPrizeRibbons)
```

Note how we call a method: we use the variable or constant that has a reference to the instance, a period, the name of the method, and then the input parameters enclosed in parentheses.

CLASS OR TYPE METHODS

Class (type) methods are defined with the same syntax as instance methods. However, they cannot use instance properties, but only global variables and constants and class (type) properties.

They are called from the name of the class (or structure or enumeration) rather than from an instance:

```
var a = 2  
var b = 5  
var c = Dog.addTwoNumbersWithFirst(a, second: b)
```

(This assumes that we included the method `addTwoNumbersWithFirst: Second:` in the definition of the class `Dog`, and marked it with the `class` keyword.)

Avoiding the Use of Named Parameters

You can change the default behavior that requires named parameters with an underscore (“_”) character. For example, if you place an underscore just before the name of the second parameter, you don’t need to specify the parameter name in the call, and, if you do, you will get a compiler error:

```
func addTwoNumbersWithFirst (first: Int, _ second: Int) -> Int {  
    var c = first + second  
    return c  
}  
  
var d = addTwoNumbersWithFirst(2, 5) // Works  
var d = addTwoNumbersWithFirst(2, second: 5) // Compiler error
```

Scoping of Variables and Constants in Functions and Methods

Swift follows conventional C rules for the scoping of variables (and constants). Variables that are declared within a function are local to that function and cannot be accessed outside of that function.

Variables (and constants) that are declared at a global scope that is at the same scope of a function, or higher, can be accessed from within that function.

Variables (and constants) that are declared as properties in a class or structure can be accessed from any methods (functions) that are defined within that class or structure. (Class or type methods can only access class or type properties, not instance properties.)

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 13 exercises, go to:

understandingswiftprogramming.com/13

14 Structures

1.

That new gray squirrel programmer is cute but she seems a little nuts. She's been telling everyone that the things that structures and enumerations create are objects!



2.

There's a couple of Swift books out now that make that same claim. Does it take inheritance to have an object? What about an object created by a class that doesn't happen to inherit anything?



3.

That's like saying that a Maserati isn't a Maserati if you don't drive it over 50 miles per hour! Not to mention, what kind of object is stored in stack memory rather than the heap?

4.

It's going to drive everyone nuts if there's no agreement on what the words mean. I think Apple says that instances of structures are just instances, not objects.



A *structure* in Swift combines related data and methods together. As this definition suggests, structures are very similar to classes. Both serve as blueprints for specific instances that each have their own copy of data that is stored in the form of properties. This data can then be manipulated with methods associated with the structure.

When would you use a structure rather than a class? I'll discuss this in more detail later, but the short answer is that structures are safer, and that you would normally use a structure when you don't need inheritance (which structures do not have) and don't have any other reason to use a class. Apple recommends using a structure whenever you don't need a class.

Many of the types that you might imagine would be implemented as primitive data types in Swift, such as integers, floating point numbers, Booleans, strings, dictionaries, arrays, and the like, are actually implemented as structures.

Structures have received more attention with Swift 2 because of the protocol extension capability, which allows structures to obtain methods by conforming to protocols, much like classes obtain methods from superclasses.

Creating a Structure

A structure, like a class, allows you to create a custom data type. For example, a custom data type that defines a `Point`, a pair of `x`, `y` coordinates representing a point on a screen, might be defined as follows:

```
struct Point {  
    var xCoordinate: Double  
    var yCoordinate: Double  
}
```

So far the syntax is identical to that used for a class except that the keyword `struct` is used rather than the keyword `class`.

A specific instance of this structure can be created as follows:

```
var particularPoint = Point(xCoordinate: 2.3,  
yCoordinate: 3.8)
```

This creates an instance of the structure `Point` with an `x` coordinate of `2.3` and a `y` coordinate of `3.8`.

The data type of the variable `particularPoint` will be inferred by the compiler to be of type `Point`. If we wish, we can explicitly declare the type as follows:

```
var particularPoint: Point = Point(xCoordinate: 2.3,  
yCoordinate: 3.8)
```

Representing a point is a pretty typical use for a structure. Having a single value that is a point on a screen is a little less clumsy than worrying about two different `x` and `y` coordinates. It uses the structure type quite efficiently. It is lightweight and the code will run nearly as fast as code defining two separate `x` and `y` coordinates.

Note that the names of the properties must be used when a new instance (`point`) is created. The following will not work:

```
var particularPoint = Point(2.3, 3.8) // Compiler error
```

As with classes, when a new instance of a structure is created, the properties defined for the structure must be set to initial values, or initialized. The syntax shown above (the line above the one with the compiler error), in which the names of the properties appear as names for the input parameters in the call to `Point` that creates the new instance, makes use of what is known as a default *memberwise initializer* that is not found in classes.

With a memberwise initializer, as long as the property names are used along with a value in the statement creating the instance, it is not necessary to provide an initializer, the method-like code within the structure that sometimes performs initialization.

If a variable is used to refer to the instance, it is possible to modify the properties with dot syntax:

```
var particularPoint = Point(xCoordinate: 2.3,  
yCoordinate: 3.8)  
  
particularPoint.xCoordinate = 1.2  
  
print(particularPoint.xCoordinate) // Will print 1.2
```

However, if the instance is defined with a constant, the instance is read-only. (This is true even if the properties are defined with the `var` keyword):

```
let particularPoint = Point(xCoordinate: 2.3,  
yCoordinate: 3.8)  
  
particularPoint.xCoordinate = 1.2 // Compiler error
```

Other Ways to Initialize a Structure

A second way of initializing a structure is by providing default values for each of the properties when declaring those properties:

```
struct Point {  
    var xCoordinate: Double = 0.0  
    var yCoordinate: Double = 0.0  
}
```

The explicit declaration of type is actually not required; with a value of 0.0, the compiler would infer it to be `Double` anyway.

If this approach is taken an instance would normally be created with the properties set to initial values of 0.0 and 0.0. These values could then be set again to their proper values with dot syntax:

```
var particularPoint = Point()  
particularPoint.xCoordinate = 2.3  
particularPoint.yCoordinate = 3.8
```

A third way to initialize an instance of a structure is to provide an initializer. Like an initializer for a class, it works like a method, but does not have a `func` keyword and is called automatically when an instance is created:

```
struct Point {  
    var xCoordinate: Double;  
    var yCoordinate: Double;  
    init(x: Double, y: Double) {  
        self.xCoordinate = x  
        self.yCoordinate = y  
    }  
}
```

The use of `self.xCoordinate` and `self.yCoordinate` is actually optional; referring to the properties just with `xCoordinate` and `yCoordinate` would work. However, it is a good coding practice to use `self` to remove any ambiguity.

This works mostly the same as using a memberwise initializer, except that the parameter names are now different from the names of the properties. An instance would be created with:

```
var particularPoint = Point(x: 2.3, y: 3.8)
```

This approach might be especially useful if only some of the properties needed to be initialized by the statement creating the instance. Others would be initialized with default values set when the property is declared.

The approach also allows initialization without the use of parameter names. An underscore character is used in the definition of the structure:

```
struct Point {  
    var xCoordinate: Double;  
    var yCoordinate: Double;  
    init(_ x: Double, _ y: Double) {  
        self.xCoordinate = x  
        self.yCoordinate = y  
    }  
}
```

Creating a new instance is then done without the use of parameter names:

```
var particularPoint = Point(2.3, 3.8)
```

If an initializer is defined within a structure, it is no longer possible to use the default memberwise initializer.

Initializer Delegation

It is often efficient to have multiple initializers in a structure (or class). Frequently, there are many properties, but most get set to some initial value that is always the same. Only a few need to be set to an initial value that is different depending upon the situation when a new instance is being created.

In such cases, *initializer delegation* is commonly used. One initializer accepts the values for those properties that need to be changed, and then passes them to a second initializer that performs initialization for all the values. Classes often do this, and have two kinds of initializers, designated initializers and convenience initializers, that each handle a different part of the task.

Structures do things very similarly. However, there isn't any distinction between types of initializers. Instead, one initializer that has parameter names and types that match those in the call that creates the instance is executed when the instance is created. That initializer then calls a second initializer that does the actual initialization. In other words, it delegates part of the initialization task to another initializer.

As an example, we have a structure, `Fruit`, that allows creation of instances that represent particular pieces of fruit that have different values for the properties `skinColor`, `countryGrownIn`, `fleshColor`, and `hasSeeds`.

```
struct Fruit {  
    var skinColor: String  
    var countryGrownIn: String  
    var fleshColor: String  
    var hasSeeds: Bool  
  
    init(skinColor: String, countryGrownIn:String,  
fleshColor:String, hasSeeds: Bool) {  
        self.skinColor = skinColor  
        self.countryGrownIn = countryGrownIn  
        self.fleshColor = fleshColor  
        self.hasSeeds = hasSeeds  
    }  
    init(skinColor: String) {
```

```
    self.init(skinColor: skinColor, countryGrownIn: "United  
States", fleshColor: "white", hasSeeds: true)  
}
```

Let's say that for our particular application, most of the time three of the four properties defined in the class are normally the same: "United States" for countryGrownIn, "white" for fleshColor, and true for hasSeeds. The only property that changes a lot is skinColor, which might be "red", "green", or "yellow", and maybe other variations. We create one initializer that initializes all of the properties in the class. But most of the time, when we create a new instance, we won't define the values of all four properties in our statement that creates the new object. Instead, we'll create a new instance with just a single parameter, as shown below:

```
var aPieceOfFruit = Fruit(skinColor: "green")
```

This creates a new instance but does not call the initializer with all four parameters because the parameter names and types do not match. Instead, the initializer that matches the parameter names and types is called. That initializer, in turn, calls the other initializer. In doing so, it passes along the value for skinColor ("green"), and then specifies (default) values for the other three properties: countryGrownIn, fleshColor and hasSeeds.

If we actually want to create an instance and specifically set all of these parameters to particular values, we can still do it, by specifying all of the parameters in the call so that the initializer with all of these parameters in its input parameters will be called.

```
var aPieceOfFruit = Fruit(skinColor: "green",  
countryGrownIn: "United States", fleshColor: "white",  
hasSeeds: true)
```

No Deinitializer for Structures

Although structures have (default or otherwise) initializers, structures do not have deinitializers, which are method-like pieces of code (using the `deinit` keyword) that are called automatically when an instance of a class goes away. These are used to allow the programmer to delete any resources that may have been created that iOS would not automatically remove. This is assumed to be unnecessary for structures.

Value Versus Reference Types in Memory

Aside from inheritance, the biggest difference between structures and classes is the way that they are stored in memory. Classes are reference types, meaning that they store only a single copy of the data associated with each instance of a class in memory. This goes into the heap memory (See Chapter 19 on “Memory Management” for a discussion of the differences between heap and stack memory.) You can see the implications of storing a single copy of this data as follows. Suppose you have a class named `Apple` with a property `skinColor`, and you create an instance of it as follows:

```
class Apple {  
    var skinColor = "red"  
}  
  
var aFirstApple = Apple()  
print(aFirstApple.skinColor) // Prints: "red"
```

This creates an instance of the class `Apple` and sets the `skinColor` property to “red”, the default (since most apples are red).

Now suppose we create the instance and assign it to `aFirstApple`. And then we make a copy of the instance and assign it to another variable `aSecondApple`:

```
var aFirstApple = Apple()  
var aSecondApple = aFirstApple
```

We further set the value of the `skinColor` property of the `aSecondApple` reference to “green”.

```
aSecondApple.skinColor = "green"
```

We then print out the values:

```
print(aFirstApple.skinColor) // Prints: green
```

```
print(aSecondApple.skinColor) // Prints: green
```

In both cases, the value of the property `skinColor` is “green”. What is happening is that there is only one copy of the property, stored in heap memory, and shared by everything that references it. If its value gets changed, it is changed for everything that references it.

We can compare this with what happens when we implement `Apple` as a structure rather than a class:

```
struct Apple {  
    var skinColor = "red"  
}  
  
var aFirstApple = Apple()  
print(aFirstApple.skinColor) // Prints: red
```

This creates an instance of the (now structure) `Apple` and sets the `skinColor` property to “red”.

We again make a copy of the instance and assign it to another variable `aSecondApple`:

```
var aFirstApple = Apple()  
var aSecondApple = aFirstApple
```

Now we set the value of the `skinColor` property of the `aSecondApple` reference to “green”, just as before.

```
aSecondApple.skinColor = "green"
```

We then print out the values:

```
print(aFirstApple.skinColor) // Prints: red  
print(aSecondApple.skinColor) // Prints: green
```

In this case the `skinColor` property for `aFirstApple` is red, while the `skinColor` property for `aSecondApple` is green. Why? When the instance reference contained in `aFirstApple` was assigned to the variable `aSecondApple`, it did not just assign a reference. Instead, it made a copy of the entire structure and stored it in stack memory. So there were then two separate copies of the instance. The value of the property `skinColor` for one of the copies could be changed to “green” while allowing the value of the property `skinColor` for the other copy to remain the same, “red”.

Choosing Between a Structure and a Class

If you need inheritance, then the decision is simple: you must use a class.

If you don't need inheritance, then the major factor in deciding between a class and a structure is whether a reference type or a value type is better.

You will hear advice from some developers to the effect that making copies is “expensive”, and thus reference types are more efficient. This is true in some cases. It depends on what is contained in the data structure that is being copied. If it is something big, like a large image, it may well be true that copies are expensive and a class may be a better choice. If, however, the structure consists of small pieces of information, tightly coupled together, then this is a situation for a structure. If runtime performance is an issue, a structure may be preferred because accessing the stack memory is much faster than accessing the heap memory.

Unless the data in the structure is unwieldy, safety can be a major consideration. Making copies is safer than passing references that can then be used to corrupt the single existing copy of data. This can be particularly risky when you are dealing with multiple threads and you have different pieces of code making changes to that single copy simultaneously. This can make things very difficult to debug.

In cases where the data comes from, or is being passed to, a Cocoa Touch API, you may not have much of a choice. Many Cocoa Touch APIs expect that the types that it deals with will not only be a class, but a subclass of `NSObject`.

For whatever reason, structures tend to be used much less frequently than classes in apps. (Structures are obviously a very important part of the Swift language itself, since the language's fundamental types have been implemented with structures.) This may well change with Swift 2 if “protocol oriented programming” and the use of protocol extensions to define methods catches on.

Using Methods With Structures

It is possible to use instance (and also “class” or “type”) methods with structures, which are used in the same way that they are used for classes, with one exception. The exception is that methods are not allowed to modify the values of properties defined with structures unless the keyword `mutable` is used when the method is defined.

To use our example of an Apple structure, we can include the following method within the structure:

```
struct Apple {  
    var skinColor = "red"  
    func printSkinColor () {  
        print(self.skinColor)  
    }  
}
```

We can then create an instance of an apple with it, and call the method to print the color of its skin:

```
var aParticularApple = Apple()  
aParticularApple.printSkinColor() // Prints: red
```

Note that this did not change the value of the (single) property of the instance.

Suppose, however, we want to call a method that *does* change the value of that property:

```
struct Apple {  
    var skinColor = "red"  
    func setSkinColor (color: String) {  
        skinColor = color // Compiler error  
    }  
}
```

We would try to create a new instance of Apple, then call the `setSkinColor` method with a parameter that will change the value of the property `skinColor`:

```
var aParticularApple = Apple()  
aParticularApple.setSkinColor(color: "green")
```

However, this won't work. The compiler won't allow us to define a method for the structure that writes to a property, unless it has the `mutating` keyword just before the `func`:

```
struct Apple {  
    var skinColor = "red"  
    mutating func setSkinColor (color:String) {  
        skinColor = color // Will now compile and work OK  
    }  
}
```

Differences Between Classes and Structures

There are far more similarities than differences between classes and structures. The following chart describes the similarities and differences:

	Classes	Structures	Enumerations
Memory Type	Reference	Value	Value
Has instances	✓	✓	✓
Inheritance	✓		
Type casting	✓		
Conform to protocol	✓	✓	✓
Can be extended	✓	✓	✓
Deinitializer	✓		
Instance methods	✓	✓	✓
Class (type) methods	✓	✓	✓
Stored instance properties	✓	✓	
Stored class (type) properties		✓	✓
Computed properties	✓	✓	✓
Lazy properties	✓	✓	✓
Property observers	✓	✓	✓
Respond to notifications	✓		
Custom subscripts	✓	✓	✓
Memberwise initializer		✓	

In most cases, those capabilities that structures have work in the same way that they do for classes, as follows:

Computable Properties. These are properties that are accessed with dot syntax like ordinary stored properties, but actually result in code being executed to calculate a value rather than simply retrieving it. Reading a property causes a `get` piece of code to be executed, while writing to a property causes a `set` piece of code to be executed. The latter can write to other stored properties in the structure. The syntax is exactly the same as for classes.

Lazy Properties. A lazy property is not initialized until it is actually accessed. Properties are marked as lazy when they involve some effort and delay and aren't necessarily always used. If they never get accessed, the effort and delay does not occur. A property is defined as lazy by simply including the keyword `lazy` before the `var` keyword. The syntax is exactly the same as for classes.

Property Observers. A property observer is a piece of code that is executed either just before or just after a new value for a property is set. Code marked with the `willSet` keyword indicates code that will be executed just *before* a new value is set; code marked with the `didSet` keyword indicates code that will be executed just *after* a new value is set. The syntax is exactly the same as for classes.

Class (Type) Property. A class or type property is a property that is common across all instances of a class or structure, and is not associated with a particular instance. The keyword `static` is used to designate a class or type property for a structure. This comes just before the `var` or `let` in the declaration. (In the case of a class the keyword `class` is expected to be used. As of Swift 1.2 class/type properties do not yet work for classes.)

Class (Type) Method. A class or type method is a method that is performed for the class or structure as a whole, rather than for a particular instance. The keyword `static` is used to designate a class or type method for a structure. This comes just before the `func` keyword in the declaration. (In the case of a class the keyword `class` is used.)

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 14 exercises, go to
understandingswiftprogramming.com/14

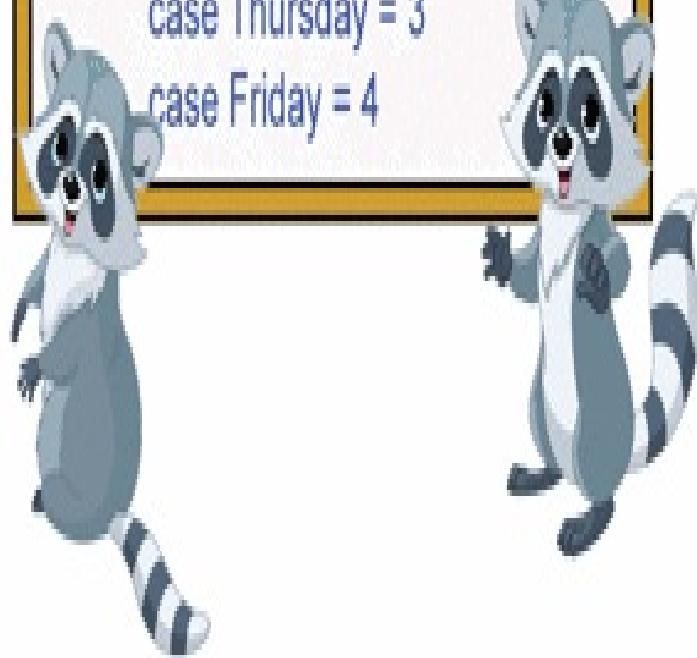
15 Enumerations

Enumerations are **so** confusing.
Having the values be like English
words is nice, but now I don't
understand what raw values are
for.

You sometimes want them in
another form. If the raw values
are integers and are 0, 1, 2, 3,
and so forth, you can use them
as an index to an array.

```
num DayOfTheWeek {  
    case Monday = 0  
    case Tuesday = 1  
    case Wednesday = 2  
    case Thursday = 3  
    case Friday = 4
```

```
let day = DayOfTheWeek.Monday  
let index = day.rawValue  
let event = EventsThisWeek[index]]  
    println(event)
```



Enumerations can be one of the more confusing aspects of Swift. This isn't particularly surprising. Enumerations really combine two capabilities. One capability allows the creation of a custom data type that contains a predefined set of values. The other capability uses this custom data type in ways that are very much like a class. It is the latter capability that causes the confusion, since it seems so different from the basic idea of enumeration, and because programmers new to it have trouble with something that seems to have some of the characteristics of a class (but not some of the expected capabilities such as stored properties) and is overlaid on top of the idea of enumeration values.

In this chapter I will describe only enumerations in their role as predefined sets of values. In a later chapter (Chapter 31, “Enumerations Revisited”) I will describe how enumerations are used in ways that are indeed very much like classes.

An enumeration, according to Apple, “defines a common type for a group of related values and enables you to work with these values in a type-safe way within your code.” Let’s take these one by one.

Defining a Common Type for a Group of Related Values

An enumeration in Swift defines a new (custom) type with a name that refers to a set of possible values. (The word *enumerate* means to “name one by one” or “to specify, as in a list”.)

To define an enumeration, you use the keyword `enum` and follow it by the name of the new type and a list of the allowed values. The list is contained within braces and the keyword `case` appears before each value:

```
enum DayOfTheWeek {  
    case Sunday  
    case Monday  
    case Tuesday  
    case Wednesday  
    case Thursday  
    case Friday  
    case Saturday  
}
```

This data type `DayOfTheWeek` defines all of the possible values allowed for the day of the week: `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, and `Saturday`.

A variable declared with a type of `DayOfTheWeek` that has one of the possible values for `DayOfTheWeek` stored in it can be created like this:

```
var today = DayOfTheWeek.Friday
```

You can also refer to it in a shorthand version using only the reference to the value, if you explicitly declare the type of the variable:

```
var today: DayOfTheWeek = .Friday
```

There's also an alternative syntax that requires only a single `case` statement:

```
enum DayOfTheWeek {  
    case Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,  
    Saturday  
}
```

The values, also known as *members*, or *member values*, represent related concepts. These must be values that will not change during the lifetime of the program, since they are fixed by the `enum` definition. The days of the week or the months of the year are good examples of enumerations, since they do not change.

Working with the Values in a Type-Safe Way

Apple's definition of an enumeration also mentioned *work with these values in a type-safe way*. What I believe that they mean by this is the way that Swift handles enumerations, in contrast with the way it is done in Objective-C. In Objective-C and C, enumerations are actually integers, with each name really just an alias for an integer.

Swift defines enumeration values differently. In the example above, where `today` is set to the value `Friday` (referred to as `.Friday` just as shorthand for `DayOfTheWeek.Friday`), the value *really is* `Friday`. It is not an integer that stands in for `Friday`, nor is it the string “`Friday`”. It is the value `Friday`, which by defining it as an enumeration makes it a symbol that is not much different from `var` or `class` or `func` or `true` or `false`.

Defining a type with values like this makes the code safer. If the day of the week is instead represented by an integer (0 to 6, say), it would be possible for errant code to write an incorrect integer value (say 764) into the variable that stored the value representing the day of the week. This can't be done if the allowable values are specifically defined.

Using enumerated values, which are chosen to be easily understandable by humans, also makes the code more readable. And more readable code is itself safer.

Raw Values

It is sometimes the case that you want to use some representation in addition to the defined values of the enumeration. In particular, you might want to use an integer, for example, to allow you to store something in an array using the integer as an index. Thus we might want to sometimes use the integer value 5 instead of the enumeration member value Friday.

We can do this by defining what are known as *raw values*. These are defined as part of the original definition of the enumeration. For example, if we wanted to include integers as part of our DayOfTheWeek enumeration, we could do it as follows:

```
enum DayOfTheWeek: Int {  
    case Sunday = 0  
    case Monday = 1  
    case Tuesday = 2  
    case Wednesday = 3  
    case Thursday = 4  
    case Friday = 5  
    case Saturday = 6  
}
```

This definition is just like the original except that we have added two things. First, just after the name of the enumeration, we add a colon (“：“) and the type of the raw value. We are using an integer here, but we could also use a string, character, or any of the integer or floating point number types. And second, for each enumeration value that we define, we follow it with an equals sign and a value that will become the raw value.

Each raw value must be unique. Once a raw value for an enumeration member is set, it cannot be changed.

To get the raw value, we access the `rawValue` property of a particular member:

```
var today = DayOfTheWeek.Friday  
print(today.DayOfTheWeek.Friday.rawValue) // Prints: 5
```

We actually didn't have to define all of the integer values the way we did in our definition of an enumeration with raw values. For the special case of an integer, all we need to do is to specify the first value of the integer, and the compiler will take care of the result, incrementing each value:

```
enum DayOfTheWeek:Int {  
    case Sunday = 0  
    case Monday      // Will be 1  
    case Tuesday     // Will be 2  
    case Wednesday   // Will be 3  
    case Thursday    // Will be 4  
    case Friday      // Will be 5  
    case Saturday    // Will be 6  
}
```

Using Enumerations with a Switch Statement

It is quite common to use enumerations with switch statements. This makes it easy to perform different actions depending upon the value of the enumeration. The result is also code that is very readable. An example is:

```
enum DayOfTheWeek {  
    case Sunday  
    case Monday  
    case Tuesday  
    case Wednesday  
    case Thursday  
    case Friday  
    case Saturday  
}  
  
var today = DayOfTheWeek.Friday  
  
switch(today) {  
    case: .Sunday {  
        print("Today is Sunday")  
    }  
    case: .Monday {  
        print("Today is Monday")  
    }  
    case: .Tuesday {  
        print("Today is Tuesday")  
    }  
    case: .Wednesday {  
        print("Today is Wednesday")  
    }  
    case: .Thursday {  
        print("Today is Thursday")  
    }  
}
```

```
}

case: .Friday {
    print("Today is Friday")
}

case: .Saturday {
    print("Today is Saturday")
}
```

No default is provided because the only possible values of `dayToday`, if it has the type `DayOfTheWeek`, are those values covered by the `case` statements in the `switch` statement.

	Classes	Structures	Enumerations
Memory Type	Reference	Value	Value
Has instances	✓	✓	✓
Inheritance	✓		
Type casting	✓		
Conform to protocol	✓	✓	✓
Can be extended	✓	✓	✓
Deinitializer	✓		
Instance methods	✓	✓	✓
Class (type) methods	✓	✓	✓
Stored instance properties	✓	✓	
Stored class (type) properties		✓	✓
Computed properties	✓	✓	✓
Lazy properties	✓	✓	✓
Property observers	✓	✓	✓
Respond to notifications	✓		
Custom subscripts	✓	✓	✓
Memberwise initializer		✓	

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 15 exercises, go to
understandingswiftprogramming.com/15

16 Properties and Class or Type Methods



“This app is running **so** slow. It’s got to be those lazy properties. They don’t want to work any more than my boss’s brother here does.”

An earlier chapter (Chapter 12 on “Classes, Objects, and Inheritance”) described basic properties and how they are used with classes and objects. Two other earlier chapters (Chapters 14 and 15) described structures and enumerations.

There are a number of different kinds of properties—stored properties, computed properties, and lazy properties. Some can have property observers. There are instance and class or type properties. Properties can be used not only with classes, but also with structures and enumerations. But the situation is a little complex—different kinds of properties can be used with different kinds of these types.

The figure below shows what kinds of properties can be used by different types:

	Classes	Structures	Enumerations
Memory Type	Reference	Value	Value
Has instances	✓	✓	✓
Inheritance	✓		
Type casting	✓		
Conform to protocol	✓	✓	✓
Can be extended	✓	✓	✓
Deinitializer	✓		
Instance methods	✓	✓	✓
Class (type) methods	✓	✓	✓
Stored instance properties	✓	✓	
Stored class (type) properties		✓	✓
Computed properties	✓	✓	✓
Lazy properties	✓	✓	✓
Property observers	✓	✓	✓
Respond to notifications	✓		
Custom subscripts	✓	✓	✓
Memberwise initializer		✓	

Thus, stored instance properties can be used by classes and structures, but not enumerations. Stored class (type) properties can be used by structures and enumerations, but not classes. (They are expected to be implemented for classes at some future point.) Computed properties, lazy properties, and property observers are supported by all three types.

In this chapter I will describe how these various properties are used—class (or type) stored properties, computed properties, lazy properties, and property observers.

I will also describe how class or type methods are used, which can be used by all of these types.

I refer to properties or methods that apply to an entire class, structure, or enumeration by calling them “class or type” properties or methods. I’m trying to avoid confusion here. Apple just calls them *type properties* or *type methods*, but I think this is confusing because the word *type* refers to a much broader set of things than just classes, structures, or enumerations. Also, the keywords used are `class` or `static`.

Class or Type Properties

Class (type) properties don't actually work yet for classes. Class/type properties *do* work for structures and enumerations, but if you try to use them with classes, you get an error message "class variables not yet supported". Apple's documentation doesn't say what the correct syntax will be for class properties when used for classes. Therefore I will describe (guess) how I expect that they will work here for classes, and how they do work for the other types.

CLASSES

A *class property*, also known as a *type property*, stores a value for the class as a whole. Such a property will, in the case of a class, presumably be defined with the *class* keyword in front of the *var* or *let* keyword. The following shows an example of a class (type) property as used in a class:

```
class Apple {  
    var weightOfApple = 0.0  
    class var weightHeaviestApple = 0.0  
    // Warning:  
    // 1. Does not yet work for classes  
    // 2. Use of class keyword is a guess  
}
```

The *class* keyword in the third line makes *weightHeaviestApple* a type or class property. This may look a bit like a class nested within another class, but it is not. The idea here is that this property stores something for the class as a whole. This property stores the weight of the heaviest apple in the class. The class also has an instance property that stores the weight of particular apples. (And presumably lots of other instance properties not included here.)

STRUCTURES

The example below defines a structure that works exactly like the class example above. The difference is that the *struct* keyword is used instead of *class* when defining the structure, and the *static* keyword used in front of the variable declaration in the third

line.

```
struct Apple {  
    var weightOfApple = 0.0  
    static var weightHeaviestApple = 0.0  
}
```

ENUMERATIONS

Enumerations also use the keyword `static` for class or type properties.

Computed Properties

The properties that I have described so far store values associated with a particular instance or for the class as a whole and later retrieve them. Such properties are known as *stored properties*. A class can also have what are known as *computed properties*. A computed property is accessed for retrieval just like a stored property, but it neither stores nor retrieves the value of the property from memory. Instead, it computes the values when accessed. The code below shows a class with a single computed property, `weightKg`:

```
class Fruit {  
    var weightInLbs: Float = 0.0  
    var weightInKg: Float {  
        get {  
            return weightInLbs * 2.20462  
        }  
        set (newValue) {  
            weightInLbs = newValue / 2.20462  
        }  
    }  
}
```

The computed property `weightKg` provides the weight of a piece of fruit in kilograms. That weight is actually stored in pounds, in the property `weightLbs`.

We can create an instance of `Fruit` as follows:

```
var pieceOfFruit = Fruit()
```

We can then set its weight to be half a pound:

```
pieceOfFruit.weightInLbs = 0.5
```

If we access the `weightInKg` property, Swift will compute the weight in kilograms by executing the code associated with the `get` keyword, returning the value that was

computed.

```
print(pieceOfFruit.weightInKg) // prints: 1.10231
```

We can also write a value to the computed property:

```
pieceOfFruit.weightInKg = 0.8
```

This will cause the code associated with the `set` keyword to be executed. The parentheses just after the `set` serve to define the name of an input parameter which is set to the value that was written to the computed property. By convention, we have called this `newValue`, although we could have used any legitimate identifier name. The code converts the value in `newValue` to pounds and stores it in the stored property `weightInLbs`.

Computed properties must have a getter, using the keyword `get`. A setter, using the keyword `set`, is optional.

This is an instance computed property, and can be used in classes, structures, and enumerations.

Class or type computed properties also exist, and can be used in classes, structures, and enumerations, but are rarely used.

Lazy Properties

A lazy property is a property the value of which is stored only when it is accessed, rather than being stored when a new instance is initialized.

In cases where it takes some effort to initialize a property, it may be desirable to delay initializing it. The property may have a value that is downloaded over the Internet, or otherwise obtained in an expensive manner. In many cases the property may never be accessed and in such a case this saves the effort (and potential delay) of obtaining the value. To defer initialization until a property is first accessed, put the keyword `lazy` just before the `var` or `let` defining whether the property is a variable or a constant.

The example below shows a property that contains an image, which is obtained by the method `getFruitImage`:

```
lazy let imageOfFruit: UIImage =
getFruitImage("fruitImage.gif")

func getFruitImage(imageFilename String) -> UIImage {
    // Add code here to
    // go to server and get an image for imageCode,
    // given the name of the file
    // then return the image from the function
}
```

Property Observers

A property observer has syntax similar to a method. For a particular property, there are two possible pieces of code. One is executed just *before* a value is stored to the property in question; the other is executed just *after* a value is stored to the property.

A property observer is a piece of code that is executed when the value of a property is about to be, or has just, changed. Property observers obviously only work with properties that are variables, not constants, since the latter never change.

A property observer is included just after the definition of a property in a class (or structure or enumeration) definition. If the code is to be executed just before the value of a property is set, the keyword `willSet` is used. If the code is to be executed just after a property is set, the keyword `didSet` is used.

If `willSet` is used, a constant named `newValue` is defined by the compiler that contains the value that the property will be set to.

If `didSet` is used, a constant named `oldValue` is defined by the compiler that contains the value that the property was previously set to.

For an observer to execute code only after a value has changed, code like the following goes in the definition of the property:

```
var color: String = "red" {  
    didSet {  
        print("The property color was set to a new value; the old  
        value was \(oldValue)")  
    }  
}
```

This prints out the old value of the property, using the `oldValue` constant provided automatically.

If code is to be executed both just before and just after a value has changed, both the `willSet` and `didSet` keywords are used.

```
var color: String = "red" {
    willSet {
        print("The property color will be set to a new value of \
(newValue)")
    }
    didSet {
        print("The property color was set to a new value; the old \
value was \ oldValue")
    }
}
```

You can make up a custom name to replace `oldValue` or `newValue`, if desired, by placing the custom name within parentheses just after the `willSet` or `didSet` keyword:

```
var color: String = "red" {
    willSet (willBeColor) {
        print("The property color will be set to a new value of \
(willBeColor)")
    }
    didSet (wasColor) {
        print("The property color was set to a new value; the old \
value was \ (wasColor)")
    }
}
```

Property observers are not called when a property is initialized, either when it is provided with an initial value or such a value is set with a default, convenience, or memberwise initializer.

Property observers cannot be used with properties that are initialized lazily (with the `lazy` keyword).

Property observers will only work with stored properties, not computed properties. (You can put any code you want to execute when a computed property is accessed in the getter

or setter of the computed property.) If a computed property has been inherited, you can add a property observer to it (since you might not have access to its getter or setter).

Observers as described above can also be used for global and local variables, even though these are not properties. (Global variables are those that are defined outside of any definition of a class, structure, enumeration, function, method, or closure. Local variables are those that are defined within a function, method, or closure.)

Class (Type) Methods

Most methods that you will be using with Swift are instance methods. This means that they are invoked from a particular instance of a class, and if reference is made to an instance property, the particular instance they are invoked from determines which copy of the property is used. Instance methods can also refer to class or type properties.

Class methods, like class properties, are different. A class method, also known as a type method, is invoked from the name of the class, structure, or enumeration. Because it has no concept of what an instance is, it cannot access instance properties, but only class (type) properties.

Because classes do not yet have class properties, I'll use an example of a structure with a class (type) property and a class (type) method, is shown below:

```
struct Apple {  
    var weightOfApple = 0.0  
    static var weightHeaviestApple = 0.0  
  
    static func printWeightOfHeaviestApple() {  
        print("Weight of the heaviest apple is \(  
            weightHeaviestApple)\")  
    }  
}
```



```
Apple.weightHeaviestApple = 1.2  
Apple.printWeightOfHeaviestApple()  
// Prints: Weight of the heaviest apple is 1.2
```

This sets the class/type property and then executes the method that prints its value.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 16 exercises, go to
understandingswiftprogramming.com/16

17 Functions Revisited: First Class Citizens



"Why are functions first class citizens when raccoons can't even vote?"

In Swift, functions are “first class citizens”, meaning that they can be used in the same ways that values and variables can be: they can be assigned to variables and constants, they can be passed as input parameters to functions (and closures), and they can be passed as return values from functions and closures.

The term comes from a comment made in the 1960s about procedures in Algol being “second class citizens” because they could not be used in these ways, unlike values, variables, and other entities in the language. Most of today’s modern languages now have this capability of treating functions as first class citizens. Functions that can be passed in this way and that can use functions as input parameters and return values are often called “higher order functions”.

In this chapter I will first describe how functions can be passed in these ways, and then discuss perhaps the most interesting kind of function as first class citizen, the nested function, which allows you to define one function within another and capture the variable context as you pass the inner function to the outside. That function can then be assigned to a variable on the outside and executed, making use of the environment that was created in the nested function.

Assigning a Function to a Variable

To assign a function to a variable, first define it:

```
func addTwoNumbers (a:Int, second b:Int) -> Int {  
    var c = a + b  
    return c  
}
```

Then assign it by name to a variable:

```
var add = addTwoNumbers
```

The compiler will infer a type for `add`, which will be the type of the function—its signature, the input parameter types, return value, and return types. You can also define it explicitly if you want:

```
var add:(Int,second: Int) -> (Int) = addTwoNumbers
```

You can now execute it with the variable `add`:

```
print(add(2, second:7)) // Prints: 9
```

This isn't terribly exciting, since it is just doing what we could do anyway with the original name of the function. But we can also do this when we obtain a function by calling another function.

Returning a Function from Another Function

We start by putting our `addTwoNumbers` function inside of another function:

```
func giveMeTheAddFunction () -> ((Int, second: Int) ->
(Int)) {
    func addTwoNumbers (a: Int, second b: Int) -> Int {
        var c = a + b
        return c
    }
    return addTwoNumbers
}
```

We then execute that function and assign the result to a variable.

```
var d = giveMeTheAddFunction()
```

We can now add numbers by executing that function:

```
print(d(3, second: 4)) // Prints: 7
```

The first line of the `giveMeTheAddFunction`, which has two return arrows in it, looks rather strange. Does `giveMeTheAddFunction` have two return values? No. The second return arrow is part of the definition of the type that `giveMeTheAddFunction` is returning, which is itself a function that has a return arrow contained in its type.

This is very much like we will do in the last part of this chapter—create a function nested within another function, and then pass that function outside of the outer function. The difference is that with this example we only care about the procedure the function executes—adding two numbers together. We don't need the surrounding environment of variables and the like to be captured.

Passing a Function in an Input Parameter

We can also pass a function to another function as an argument. To do this, we create the function `iDontKnowHowToAdd`.

Like the `giveMeTheAddFunction` above, the function includes the type of another function in its definition—the type of the `addTwoNumbers` function. It's very easy to make a mistake when typing in these kinds of types, and the mistakes aren't necessarily that easy to detect. So we can use a little trick: Defining the type of the function that we are passing with a typealias. We do this as follows:

```
typealias twoIntsInAndOneOut = (Int, second: Int) -> (Int)
```

Now when we need to use the expression `(Int, second: Int) -> (Int)` we just use `twoIntsInAndOneOut` instead:

```
func iDontKnowHowToAdd(f: twoIntsInAndOneOut, a: Int, b: Int) -> Int {
    var c = f(a, b)
    return c
}
```

Using the typealias also makes the code more readable and, frankly, less weird-looking. There are three input parameters, the function `addTwoNumbers` and two integers. We've given a local name to the function `f` with that type (defined by the typealias), and we've also given local names of `a` and `b` to the integers that are the second and third input parameters.

Now, of course, the idea here is that the `iDontKnowHowToAdd` function supposedly cannot add (although this is really a fiction), and so we pass in the `addTwoNumbers` function so that it has the capability to add. We also pass in two integers that we want to be added. The return value is `Int`, the type of the value of the sum of the integers that the `iDontKnowHowToAdd` function will end up returning.

In the actual body of the function, we execute the function we passed in by referring to the local name `f` and use the local names `a` and `b` as input parameters. The sum is assigned to `c`, which is then returned.

To actually run this, we first define the `addTwoNumbers` function:

```
func addTwoNumbers (a: Int, second b: Int) -> Int {  
    var c = a + b  
    return c  
}
```

We then define the `iDontKnowHowToAdd` function:

```
typealias twoIntsInAndOneOut = (Int, second: Int) -> (Int)  
func iDontKnowHowToAdd(f:  
    twoIntsInAndOneOut, a: Int, b: Int) -> Int {  
    var c = f(a, second: b)  
    return c  
}
```

We then execute the `iDontKnowHowToAdd` function with two integers:

```
var m = iDontKnowHowToAdd(addTwoNumbers, a: 2, b: 7)  
print(m) // Prints: 9
```

The parameter names `a` and `b` are required in Swift 2. They should not be used in Swift 1.

Nested Functions and Capturing the Environment

In attempting to understand functions and closures, and particularly the business of capturing surrounding variables and constants, it can be helpful to look at nested functions. Nested functions are when one function is defined inside another function. We actually saw a nested function earlier in the `giveMeTheAddFunction` example. But this did not demonstrate the full power of a nested function, including that of capturing the environment. The inside, or inner, function cannot be directly called by code outside of the outer function, although that function can be passed as a return value to the code outside and then called once it is outside. We'll see an example of this below.

A global function, which is to say a function that is in the global scope—at the highest level of the code and not inside another function—cannot capture the surrounding environment of variables, constants, and their values. Only a function nested within another function can do this.

The example of a nested function shows how a function, which in Swift can be passed around like it is a variable, literal or type, and can be passed as a return value from another function. In addition, the example shows how variables and their values can be captured and used later, even when they have gone out of scope in their original definition, as part of the “closure” mechanism.

This pair of functions keeps track of the elevation of a person who is traveling in the mountains. It has a property, `currentElevation`, that is initially set to 0 (sea level) when the outer function, `initializeElevation`, is initially called. The outer function then returns the inner function, `updateElevation`. The outer function is called only at the beginning and when you want to start over at an elevation of 0.

The returned inner function, `updateElevation`, is then assigned to a variable `f`, which effectively becomes a function and can be executed by referring it using the same syntax as a function, in this case by using its name and a pair of parentheses after the name, with a number within the parentheses to indicate the amount of change (in feet) in elevation, e.g., as `f(100)`. Calling the function `f` in this way will update the property `currentElevation` variable and print out its current value. Calling the function with a new value reflects the traveler making a change in elevation by the amount specified; the property `currentElevation` reflects the overall elevation level that the function keeps track of.

Here's the code:

```
func initializeElevation() -> (Int) -> () {  
    print("Initialize elevation to 0 feet")  
    var currentElevation = 0  
    func updateElevation(updateAmount: Int) {  
        currentElevation += updateAmount  
        print("Update \(updateAmount); currentElevation is now \(currentElevation) feet")  
    }  
    return updateElevation  
}
```

This is mostly just a normal Swift function inside another Swift function. Again, the outer function has two return arrows, with just part of the type of the function being returned.

We can now call the outer function. Because the outer function is returning the inner function as a return value, we are assigning the *inner function* to the variable `f`:

```
var f = initializeElevation()
```

We can now execute the inner function, by calling the function `f` with a number for an elevation change:

```
f(100) // Prints: Update 100; current elevation is now 100 feet
```

And again:

```
f(100) // Prints: Update 100; current elevation is now 200 feet
```

And:

```
f(500) // Prints: Update 500; current elevation is now 700 feet
```

And:

```
f (-300) // Prints: Update -300; current elevation is now 400 feet
```

Now, `f` is not a very good name for this function. If we try to use the name `updateElevation`, it will not work, because an inner nested function cannot be called from outside (except by doing just what we have done, passing it outside as a return value).

We can, however, assign that return value to a variable with the name `updateElevation`, and then we will be calling it with a more descriptive name.

```
var updateElevation = initializeElevation()  
// Prints: Initialize elevation to 0 feet
```

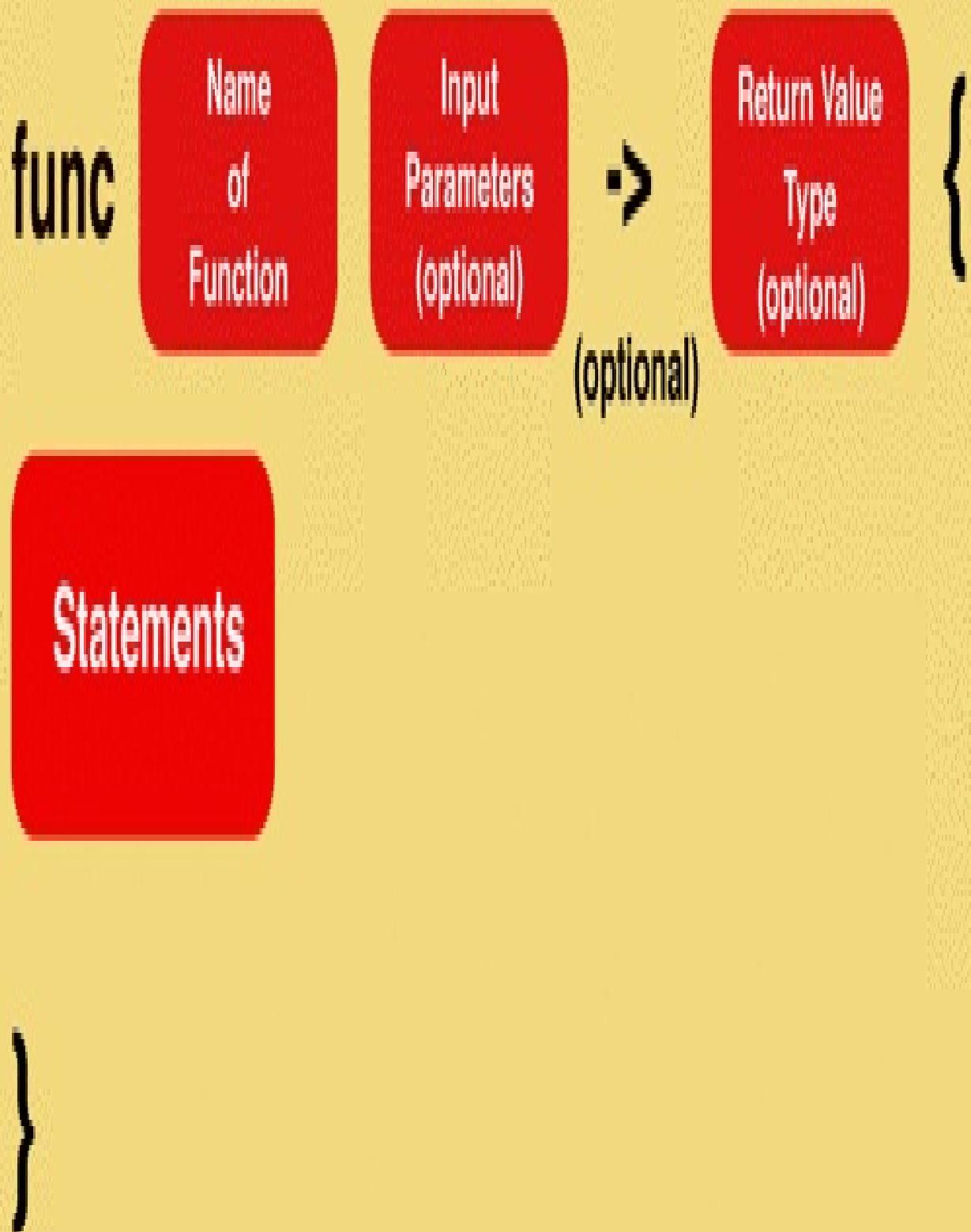
We have initialized the elevation to 0 again.

And now we can use the name `updateElevation` as a function:

```
updateElevation(400)  
// Prints: Update 400; current elevation is now 400 feet
```

The call to `updateElevation` executes the inner function that has been stored in the variable `updateElevation`. This updates the variable `currentElevation`, which, once the outer function has finished executing and returned a value, has gone out of scope and been released and likely has been deallocated from memory. How, then, can the call to `updateElevation` work? Only because the variable and its value have been captured, along with the ability to update that value.

Only the function that has been returned to the global scope can now access the variable `currentElevation`. It cannot otherwise be accessed from the global scope, and the function that created it has by now likely disappeared.



Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 17 exercises, go to
understandingswiftprogramming.com/17

18 Closures and Closure Expressions



"Here it is. It has the code and the variables and constants and their values from the surrounding environment. All this is packaged up so that the code can be run anywhere."

A *closure* is a general term that can refer to either a *function* or a *closure expression*. Both include chunks of code that perform a specific task. Both can, if desired, (but are not required to) accept values as input parameters and return a value that is the result of the processing the code has done.

Closures, in the right context, will capture, or *close upon*, the names and values of variables and constants that are in their scope that the code in them refers to and package these variables and constants together with the code of the closure for use in later execution. It is this capability that they are named for. Closures in Swift are sometimes known as *blocks* and they are the same idea as that of a block in Objective-C. A Swift closure, either a function or a closure expression, can be used in any Objective-C API that will take a block.

What's the difference between a function and a closure expression? Functions have names and a specific, and somewhat rigid, syntax. Closure expressions do not have names and are sometimes known as *anonymous functions* because of this. Closure expressions have their own syntax, different from a function, with that syntax being very flexible and allowing many forms.

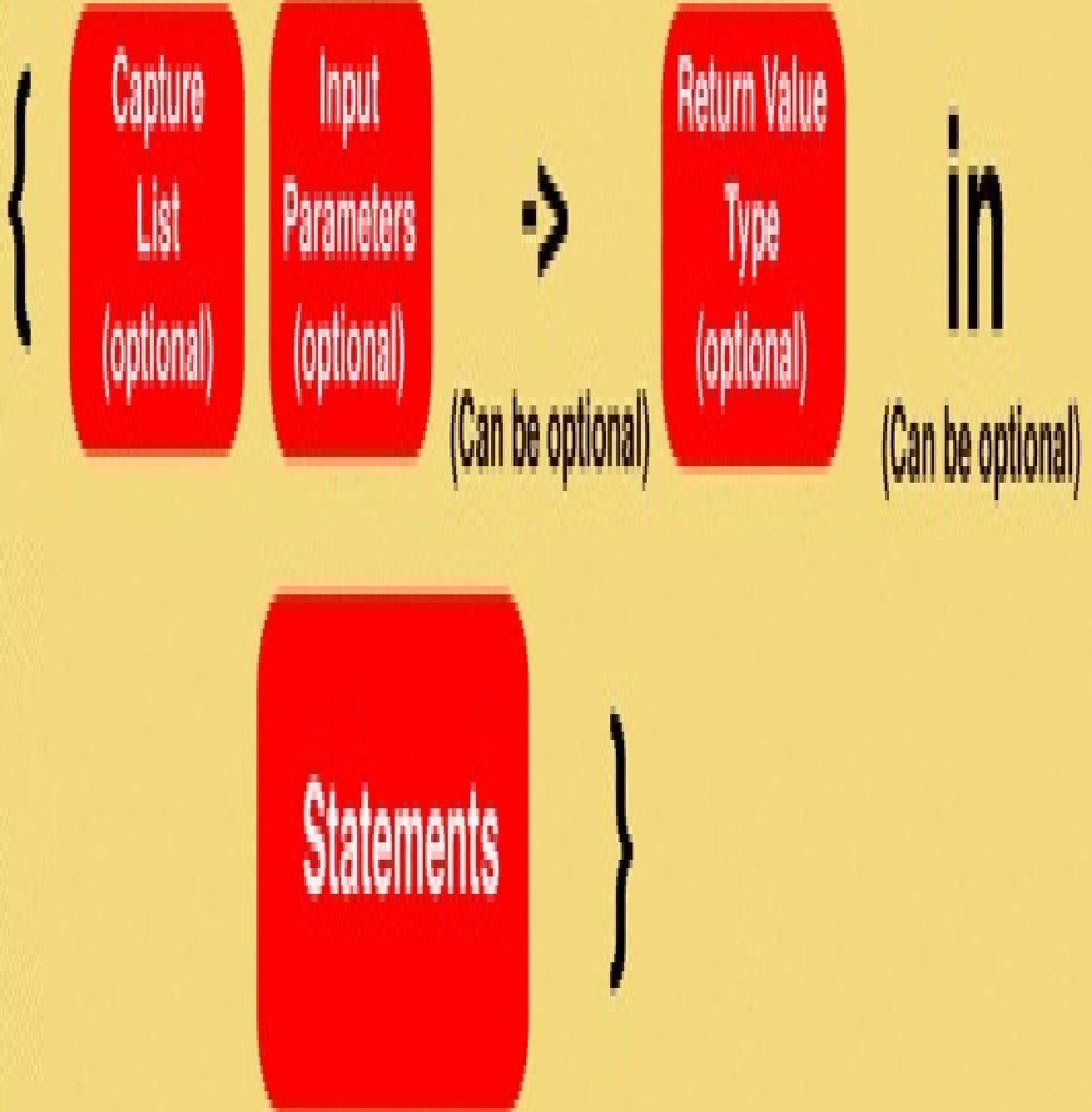
In practice, functions often have substantial amounts of code and a big motivation for creating them is so that the same function can be called in different situations, thus making programming efficient by reusing code. Closure expressions usually have relatively small amounts of code and are typically used only once.

Closure expressions are often used *inline*, meaning that the closure expression allows code to be an argument in a call to a method (often a call to an API). This often makes code more readable and direct—the code is often more easily found because it is close to the API call rather than being buried in a function somewhere else that is called.

There is often confusion about terminology. Although both functions and closure expressions are closures, when developers use the term *closure* they more often than not actually mean *closure expression*. Some developers never use the term *closure expression* but call them *anonymous functions* instead.

Closures, whether functions or closure expressions, are objects and are reference types, meaning that they are stored as part of the heap memory, and it is possible for closures to have reference cycles.

The Syntax of A Closure Expression



The figure above shows the syntax of a closure expression. Nearly every part of it can be optional. This diagram shows the syntax as it is normally used. In some narrow circumstances, even the braces are not required and a conditional expression can be used instead of a statement.

We can compare the full syntax for a Swift function and a Swift closure for the same code, as follows:

```
func addTwoNumbers (a: Int, b: Int) -> Int { // Swift
function

var c = a + b

return c

}
```

This function, which adds together two integers, is called as follows:

```
var d = addTwoNumbers(2, b: 5)
print(d)      // Prints: 7
```

The same chunk of code that goes in a function can also go in a closure expression:

```
var e = { (a: Int, b: Int) -> Int in
var c = a + b
return c
}
```

In this example we have assigned this closure expression to the variable `e`. We can then execute the variable, which will execute the closure expression:

```
var d = e(2, 5)
print(d)      // Prints: 7
```

This does exactly the same thing as the line above that calls the `addTwoNumbers` function:

```
var d = addTwoNumbers(2, b: 5)
print(d)      // Prints: 7
```

If we look closely at the syntax for this closure expression:

```
var e = { (a: Int, b:Int) -> Int in
var c = a + b
return c
}
```

we find that it has almost everything that is shown in the diagram for closure expression syntax: a leading brace, input parameters, a return arrow, a return type, the keyword `in`, statements, and a trailing brace.

One thing that it does not have is a *capture list*. A capture list is a list of references (variables and constants) that might cause what is known as a *memory reference cycle*. The list contains a tag for each reference that indicates how to avoid the reference cycle. Reference cycles can cause memory leaks, in which memory that is no longer needed cannot be properly recycled. See Chapter 19 on “Memory Management” for details on reference cycles and how capture lists can be used to avoid them with closure expressions.

Using a Closure Expression in an Input Parameter of a Function

Perhaps the most common use of a closure expression is using it as an argument in a call to a function. Very often this is a call to an API that is part of the Cocoa Touch or other library in iOS. Here's an example:

The iOS system (and Mac too) has a threading system known as Grand Central Dispatch. In an iOS app, you often want to execute some code on a background thread rather than the main thread of execution so as to not interfere with the flow of processing for the user interface, which runs on the main thread. You don't want the UI to freeze or stutter. Grand Central Dispatch makes it easy to run code on a background thread. You simply put that code in one of several queues (depending upon the priority level that you want the code to run with) and the Grand Central Dispatch system will take care of running it on the appropriate thread or threads.

The code is:

```
let priority = DISPATCH_QUEUE_PRIORITY_DEFAULT
let queue = dispatch_get_global_queue(priority, 0)
dispatch_async(queue, { ()->() in
    print("Hello from the default priority background queue")
})
```

The code here first defines a priority level, then gets a reference to a queue of the desired type (We want a concurrent queue rather than a serial one) and desired priority (We want the default level, a medium level of priority).

The `dispatch_async` API call has two parameters. The first is a reference to the queue that was just created. The second expects a closure with the code to be executed. This can be either a function or a closure expression.

Here we provide it as a closure expression, which has the conventional syntax, indicating that there are no input parameters and no return values—just some code, a single `print` statement.

Defining a Function to Accept a Closure

We have to take some care in defining a function so that it will accept a closure. We cannot take too literally the idea that a closure is merely a “chunk of code”, as Apple likes to say. It is a little more than that.

For example, we can rewrite the `addTwoNumbers` function so that the second input parameter, which normally takes an integer, takes a closure instead.

Here's the function:

```
func addTwoNumbers (a: Int, b: Int) -> Int {  
    var c = a + b  
    return c  
}
```

And how we call it:

```
var d = addTwoNumbers(2, b: 5)  
print(d) // Prints: 7
```

We might think that the only thing we have to do differently is to call it with a closure instead of an integer:

```
var d = addTwoNumbers(2, b: { return 5 }) // Compiler  
error
```

But no. The problem is that the function is typed to accept an `Int` as the second parameter, not a closure. We have to do two things. First, we have to change the type of the second parameter of the function. And second, we have to execute the closure to produce the resulting integer. Below, we've set the type of the closure in the input parameter of the function to `() -> Int`, indicating that it has no input but produces a return value of an `Int`. And we've changed `b` to `b()` in the code inside the function so as to execute the closure:

```
func addTwoNumbersClosure (a: Int, b: ()->Int) -> Int {  
    var c = a + b()  
    return c  
}
```

We can now call it as before:

```
var d = addTwoNumbersClosure(2, b: { return 5 })  
print(d) // Prints: 7
```

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 18 exercises, go to

understandingswiftprogramming.com/18

19 Memory Management



**"Uh, could you put that gun down, please?
It's really making me nervous. And could you
explain again the difference between *weak*
and *unowned*?"**

To get the most efficient use of the limited amount of random access memory (RAM) that machines generally have, computer systems typically use some kind of memory management scheme. This is usually automatic garbage collection, which seeks out memory that is no longer used by a program and makes it available for reuse. Many scripting languages that run on a personal computer simply ignore the issue of managing RAM, assuming that automatic garbage collection will be implemented, and that it will solve the problem. In the case of a personal computer, that is not an unreasonable assumption. PCs generally have substantial RAM. A MacBook Air 2, for example, has a minimum of 4 Gigabytes and can have 8. Automatic garbage collection is typically implemented on personal computers and usually works well.

Mobile devices, however, are a different world. Devices like iPhone 5s and 6s have 1 Gigabyte of RAM. The iPad has recently had 1 Gigabyte as well until the iPad Air, which has 2 Gigabytes. Engineers who work on memory and processor chips generally believe that it is unlikely that iOS devices will have substantially more memory any time soon because the ARM chips used in these devices put both the memory and processor on the same die. Making the memory larger would mean making the chip larger, when it is generally considered desirable to make the chip smaller or use any additional space to make the processor more powerful. (There have, however, been recent rumors claiming that upcoming iPhones will have 2 Gigabytes of RAM.)

Devices like iPhones and iPads have remarkably high-quality media, including high resolution displays and cameras. The iPhone 6 Plus has a resolution of 401 pixels per inch. The iPhone 6 has a camera with a slow-motion version capable of operating at 240 frames per second, which has obvious high requirements for memory. Given that this camera is in a phone that users carry everywhere in their pocket, it is natural to want to use it in apps. Thus apps for these devices will frequently need access to large amounts of memory to hold these images.

Thus, the demand for memory in mobile devices is generally higher than the demand for memory in PCs, while the RAM that is available to mobile devices is typically more limited than that of a PC.

Apple's Use of Reference Counting

Until relatively recently, iOS required programmers to manage blocks of memory manually, using a system known as *reference counting* (explained below). Programmers would allocate memory when an object was created and then write statements to release memory when no longer required. This was not only tedious and time-consuming, but, more importantly, unsafe. A failure to release memory when required meant a *memory leak* (see below). Worse, an attempt to release memory that had already been released caused a crash.

It was thus highly anticipated that Apple would replace manual memory management with something better.



**“Sorry, we recycle everything.
The only group that still needs
garbage collection is the Android
team.”**

The only serious competitor to iOS as a mobile operating system, Android, uses garbage collection to reclaim memory. And before Apple introduced Automatic Reference Counting (ARC) to iOS, many developers were expecting that Apple would use garbage collection, just as it used garbage collection in Mac OS X. That didn't happen. Instead, Apple not only implemented ARC in iOS, it implemented it on OS X, and actually *deprecated* garbage collection on OS X.

Apple explained why in a session at the 2011 World Wide Developer's Conference:

Unfortunately garbage collection has a suboptimal impact on performance. Garbage can build up in your applications and increase the high water mark of your memory usage. And the collector tends to kick in at undeterministic times which can lead to very high CPU usage and stutters in the user experience. And that's why GC has not been acceptable to us on our mobile platforms.

As Drew Crawford has discussed in his paper "Why mobile web apps are slow" (July, 2013, at sealedabstract.com), garbage collection is highly sensitive to the amount of memory available—if the memory available is many times that which your app minimally requires, garbage collection is highly efficient. If memory is less than this, it starts getting pretty inefficient, and your app can run up to 70% slower than it would if manual memory management were used. Crawford says that "in a memory constrained environment garbage collection performance degrades exponentially [with the level of memory constraints]."

Crawford asserted that apps can need up to 6 times their minimal required memory for garbage collection to be efficient. Other analysts have taken issue with this, suggesting that better garbage collection algorithms that have been more recently implemented do not require as much memory.

What seems clear is that neither garbage collection nor reference counting completely solves the problem. As discussed below, it takes some programming effort to use reference counting in a way that is not necessary for garbage collection, to avoid reference cycles (which will be explained below). But it is also clear that programmers have to be generally aware of memory use in apps, particularly when making use of capabilities (like cameras) that use large amounts of memory.

Crawford's summary of the issue:

Here's the point: memory management is hard on mobile. iOS has formed a culture around doing most things manually and trying to make the compiler do some of the easy parts. Android has formed a culture around improving a garbage collector that they try very hard not to use in practice. But either way, everybody spends a lot of time thinking about memory management when they write mobile applications. There's just no substitute for thinking about memory. Like, a lot.

Using Memory in Apps

In designing an app, the general rule is to use memory only when you need it, and to give it up as soon as you no longer need it. The operating system is designed to help apps use memory very efficiently. Memory is allocated in *blocks* to apps when required. When a block is no longer required, it is deallocated and made available for use by other parts of the app or by other apps.

In app design, one has to be generally aware of how much memory is being used by the app. With today's larger memories, doing things like displaying large, high-resolution photographs, even a large number of them that a user might scroll through, won't necessarily use a prohibitive amount of memory. Roughly half of the amount of RAM that a device physically has is available for use by an app if necessary. This means about 512 Megabytes, for example, for recent phones like iPhone 5 to iPhone 6 Plus (that have 1 Gig RAM), and 1 Gigabyte for the iPad Air 2 (has 2 Gig RAM).

You want to insure not only that there is enough memory for what your app is doing at every moment but preferably also that there is memory available for other apps that may be running in the background. If your app uses all of the available memory, these other apps may get pushed out, and there may be a delay in restarting them.

Aside from insuring that there is enough memory for the app, you must particularly make sure that there are no significant *memory leaks*. A memory leak is a situation in which memory is allocated to an object but then never deallocated. This is particularly a problem in cases where the allocation and failure to deallocate happens repeatedly during execution of an app. This can cause large amounts of memory to be allocated to the app but not actually used, often ultimately resulting in a crash.

How Reference Counting Works

As discussed above, iOS devices make use of *reference counting* to manage memory. In reference counting, when a block of memory is allocated, a counter is maintained for that block. When a block of memory is allocated, the allocation is done by some other object, which then maintains a reference to that object, meaning that it is making use of the information in that block. When first allocated, the counter is set to one, meaning that one other object has a reference to it.

Should another object have a need to use information in that block, it will establish its own reference to the block, resulting in the counter for the block being increased by one, to two.

When an object having a reference to an object no longer needs information from it, it releases the object it is pointing to, which causes the reference count to be decreased by one.

When a reference count for an object, that is a block, reaches 0, the operating system will reclaim the memory and make it available for reuse.

Another (perhaps easier) way of looking at reference counting is by tracking *ownership*. When an object and its block of memory is referenced by another object, it is said to become an owner of that object. An object is allowed to have more than one owner. When an object removes a reference, it is said to *give up ownership*. When an object no longer has any owners, the operating system can reclaim its memory for reuse.

Swift makes use of *Automatic Reference Counting*, a system in which the compiler and runtime system take care of nearly all of the details of memory management. When an object is created, the compiler will generate code to cause the operating system to allocate a block of memory for it and set the reference count. When a block of memory is no longer needed (reference count of 0 or no owners), code inserted by the compiler will deallocate that memory.

This is all done automatically by the compiler and operating system. The system works well without any intervention by the programmer, with one exception. That exception involves what is known as a *reference cycle*, also known as a *retain cycle*. A reference cycle prevents automatic reference counting from working properly, and programmers need to ensure that reference cycles are avoided.

How Reference Cycles Cause Memory Leaks

Suppose you have code that creates a new object known as Parent. The Parent object will then create another object known as Child. Now suppose further that the Child object makes use of information in the Parent object, say by accessing one of its properties. There will be a reference from the code that created the new object to Parent, setting the reference counter for Parent to 1, and then there will be an additional reference from the Child object to Parent that will set the reference counter for Parent to 2. (Another way to look at this is that Parent will have two owners, the object that created it and Child.)

At some point, the Parent object will no longer be needed, and the code that created it will remove its reference to Parent, causing Parent to have a reference count of 1. However, it needs a reference count of 0 to be deallocated, and that will not happen. The reason is that Child still maintains a reference to Parent.

The Child object still has a reference count of one. It is owned, in other words, by the Parent object. But the Parent object will only remove its reference to Child, and thus its ownership, if it itself is deallocated and fails to exist. Similarly, the Child object will continue to maintain its reference to Parent, and thus its ownership, as long as it exists. The result is a deadlock situation in which neither the Parent or Child object will be deallocated.

How to Avoid Reference Cycles with `weak` and `unowned`

When Apple developed Automatic Reference Counting, it created what are essentially tags for references. These are often known as *modifiers*, and like other keywords used when declaring a property, are also known as *declaration modifiers*. If, say, one object uses another object (or one of its properties) in an assignment statement, that constitutes a reference. It means that the first object is dependent upon the second object to obtain its value. For that assignment statement to work properly, the second object must still exist in memory.

There are three kinds of tags: `strong`, `weak`, and `unowned`. When a reference is made, that reference is normally tagged as `strong`. This is done automatically as the default—you won't see many `strong` keywords in a Swift program, although a programmer could put them in whenever a strong reference is made, just to remind readers of what kind of reference was being made.

If a strong reference is made to an object, the system will keep that object around. The problem comes, as discussed earlier, when object A has a strong reference to object B, and object B has a strong reference to object A.

To avoid this, one of the other types is used. The most common is `weak`. Thus:

```
weak var length = 56
```

declares a variable that, should it make reference to an object, will make that reference a weak reference. Without the `weak` modifier, the reference would be a strong one. With a strong reference, that object would be kept around, resulting in a reference cycle. A weak reference, however, is no reference at all from the point of view of the memory management system. The object will be kept around only if something else is pointing to it with a strong reference. (In cases where there are potential reference cycles there is normally something else referencing one of the objects that will keep it around as long as it needs to be around.)

There are conventions about which kinds of references are used in different situations.

In general, when two objects have a parent and child relationship, meaning that the parent created the child, the rule is that the reference from parent to child will be strong, while the reference from child to parent will be weak. Since the parent created the child, it needs to keep a strong reference to it to keep that child alive. But since the parent created

the child, the child can reasonably assume that some other object is referencing the parent and keeping it alive. It is thus reasonable for the child's reference to the parent to be weak (or unowned).

Weak versus Unowned Modifiers

The modifiers `weak` and `unowned` mean pretty much the same thing from the viewpoint of avoiding a reference cycle—that the reference is not `strong`. A `strong` reference means that an object has ownership in another object and wants to keep it around. Either a `weak` or an `unowned` reference means that an object deliberately has NO ownership in another.

However, there is a difference between a `weak` and an `unowned` variable. A `weak` variable must be an optional, because the ARC system, after deallocating the object that the variable points to, will set that variable to `nil`.

In contrast, an `unowned` variable must be a non optional, and the ARC system will not attempt to set it to `nil`. It is assumed that an `unowned` variable has a longer lifetime than a `weak` one.

Reference Cycles in Closure Expressions

Closures (whether functions or closure expressions) are reference types and are thus subject to the memory management of heap memory.

The main problem with a closure expression is when it is used with an instance of a class. It is common for a closure expression to be assigned to the property of a class. When this happens, the code in the body of a closure expression can refer to the instance, or a property of an instance, using the keyword `self`. The assignment from the property of the class instance creates a strong reference to the closure expression; the reference to `self` creates a strong reference from the closure expression to the class instance, and there is a reference cycle.

The solution is the use of a *capture list*. A capture list is list of the names of entities that statements in the body of the closure reference that might cause a reference cycle. A typical capture list is:

```
[weak self, unowned self.multiply]
```

The syntax of a capture list consists of a list of memory reference modifiers along with the name of a variable or other entity, separated by commas. This list is surrounded by square brackets. The capture list itself comes right after the leading (left) brace of the closure, and before the input parameters (if any).

Stack versus Heap Memory

Random-access memory in an iOS device is divided into a number of separate sections. Two sections of particular interest are the *stack* and the *heap*.

The stack is a set of fixed-size areas of memory that contain basic context information.

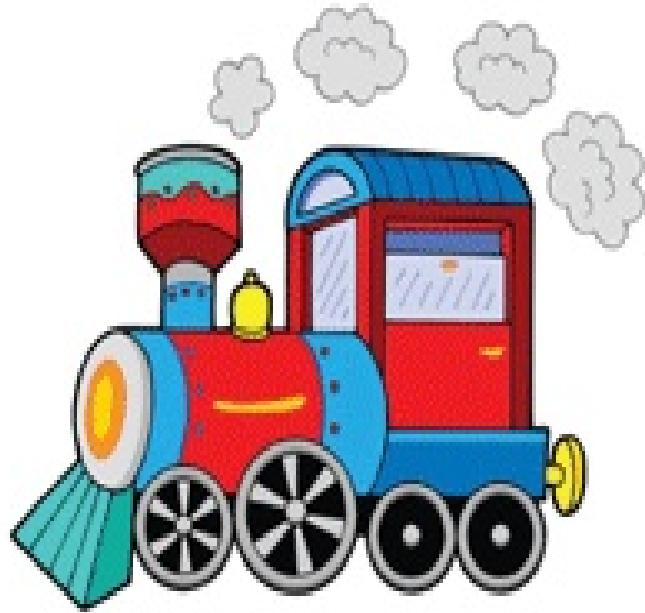
The heap, in contrast, has a very large amount of memory divided into blocks, and in which those blocks are dynamically allocated in response to demand. Thus, when a new object is created, a sufficient amount of memory will be allocated for it, and that memory will continue to be allocated to that object until the object reaches the end of its lifetime and the blocks of memory are deallocated. Allocation and deallocation of memory are performed by code created by the compiler as part of the Automatic Reference Counting system.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 19 exercises, go to
understandingswiftprogramming.com/19

20 The Flow of Control Revisited: Matching with Patterns



What
is that?

It is specialized hardware
for executing switch
statements in Swift. It's
called a *switch engine*.



In an earlier chapter (Chapter 3 on “The Flow of Control”) I described the basic conditional statements in Swift: if, if-else, and switch. The if and if-else statements work like those in C and other languages but are a little cleaner. Switch statements in Swift are slightly different than those in C, a little safer and cleaner because you do not need a break statement after each case to prevent falling through to the next case test.

In addition, switch statements have more sophisticated matching capabilities than I have described so far. There are also two other statements, if-case and for-case (new in Swift 2) that make use of these same matching capabilities. In this chapter I will describe how these switch statements that use matching patterns work.

Matching Against Patterns

In conventional if statements, for loops, and switch statements, matching is simple. A test compares a value against another value or an arithmetic or logical expression. Thus:

```
if x == 5 { print(" x is 5") }

if x > 0 && y == 3 { print("x is greater than zero and y
is equal to 3")
}
```

Swift, however, has more sophisticated matching capabilities, and allows matching against one of a number of patterns that are defined.

Swift also allows the use of where clauses to add an additional constraint to the match.

In addition, Swift allows *value binding*, which sets a temporary variable or constant to the value that is being matched so that that variable or constant can be used in statements within the case clause.

The patterns used in matching are:

Expression Pattern. This is just the conventional expression (e.g., `m==5`) that values have previously been matched against.

Tuple pattern. This matches a tuple that contains values against values or expressions that are represented as tuples.

Wildcard (underscore) pattern. The underscore character (“`_`”) will match anything. It is usually used in matching tuples, where one element of a tuple can be any value, and the case clauses mainly refer to other elements of the tuple.

Enumeration case pattern. This matches a value against the value of an enumeration member (e.g., `DayOfTheWeek.Tuesday`).

Type-casting patterns. This uses an `is`, `as?`, or `as!` operator to perform a type casting operation before the matching is attempted.

The patterns can be mixed together. Thus, the wildcard pattern is often mixed with the tuple pattern.

Just to avoid confusion: These are matching patterns. They have nothing to do with design patterns, such as the singleton, delegate, and similar patterns.

The Basic Switch Statement Syntax

To refresh your memory, here's the basic syntax for a switch statement that was shown in the earlier chapter:

```
var name = "George"

switch name {
    case "Peter":
        print ("The name is Peter")

    case "George":
        print ("The name is George")

    case "Jennie":
        print ("The name is Jennie")

    default:
        print ("No match found")
}
```

Matching Against Expressions

Matching with Swift patterns can be a little more sophisticated than conventional matching, even with just the expression pattern, since ranges can be used.

The expression to be matched must, however, be the same type as the value in the case statement it is being matched against.

An example of a switch statement matching with an integer against a set of ranges is shown below:

```
var m = 7
switch m {
    case 0...5:
        print ("The value is from 0 to 5")
    case 6..<8:
        print ("The value is 6 or 7")
    case 8...12:
        print ("The value is from 8 to 12")
    default:
        print ("The value is something else")
}
```

Using Tuples and Underscores in Switch Statements

Tuples can be used to match cases in switch statements:

```
var tup = (45, "Don Drysdale")
switch(tup) {
    case (35, "Fernando Valenzuela"):
        print("Valenzuela with 35 home runs")
    case (44, "Babe Ruth"):
        print("Ruth with 44 home runs")
    case (45, "Don Drysdale"):
        print("Drysdale with 45 home runs")
    default:
        print("Can't find anything that matches")
}
```

We can also use an underscore to indicate that, in some cases, any value in a particular position will match:

```
var tup = (45, "Don Drysdale")
switch(tup) {
    case (_, "Fernando Valenzuela"):
        print("F. Valenzuela")
    case (_, "Monica Valenzuela"):
        print("M. Valenzuela")
    case (_, "Babe Ruth"):
        print("Ruth")
    case (45, "Don Drysdale"):
        print("Drysdale with 45 home runs")
    default:
        print("Can't find anything that matches")
}
```

In this switch statement we will have a match in the first three cases regardless of the number in the first position of the tuple, while in the fourth case we still require that the first position be 45 for a match.

Matching Against Enumeration Cases

Matching can also be done against the member values of enumerations.

An example is shown below:

```
let e = DayOfTheWeek.Tuesday
switch e {
    case DayOfTheWeek.Sunday:
        print ("It must be Sunday")
    case DayOfTheWeek.Monday:
        print ("It must be Monday")
    case DayOfTheWeek.Tuesday:
        print ("It must be Tuesday")
    case DayOfTheWeek.Wednesday:
        print ("It must be Wednesday")
    case DayOfTheWeek.Thursday:
        print ("It must be Thursday")
    case DayOfTheWeek.Friday:
        print ("It must be Friday")
    case DayOfTheWeek.Saturday:
        print ("It must be Saturday")
}
```

There is no default case because the enumeration member values (all of the possible days of the week) are exhaustive.

Value Binding

It is possible to assign values to variables and constants within a case. This is known as *value binding* and is similar to what is done in if statements when unwrapping optionals. The following example is just like the first example in this chapter, but with a couple of changes in the first case. First, there is no “Peter” to match against, with `let x` being substituted. And second, when this case matches, it now prints out “The name is `\(x)`” rather than “The name is Peter”.

What is happening is that because there is a `let x` value binding instead of a value to match against, the first case will always match. A value binding acts like an underscore character and matches anything. The constant `x` is now set to the value of `name`. And then, when the `print` is executed, the value of `x` will be printed. So the switch statement prints out “The name is Peter” as before. (The example here uses a constant for `x` because there is no need for a variable, the value being set only once. Variables can also be used. But note that the scope of the constant or variable is limited to only the case clause where the constant or variable was declared.)

```
var name = "Peter"
switch name {
    case let x:
        print ("The name is \(x)")

    case "George":
        print ("The name is George")

    case "Jennie":
        print ("The name is Jennie")

    default:
        print ("No match found")
}
```

Where Clauses

Replacing the value to be matched with the `let x` of a value binding means that we no longer have a way to match the case with the original value that the switch statement was attempting to find a match for. We can do this matching, however, with a `where` clause, as shown below. An expression like `where x == "Peter"` is added. This means that the code in a case will only be executed if the original value matches what is being tested for. What this means in this particular case is that only the second case will be executed, and what will be printed out is “The name is 2 George”.

```
var name = "George"

switch name {
    case let x where x == "Peter":
        print ("The name is 1 \$(x)")

    case let x where x == "George":
        print ("The name is 2 \$(x)")

    case let x where x == "Jennie":
        print ("The name is 3 \$(x)")

    default:
        print ("No match found")
}
```

The examples I’ve shown for value binding are very simple and don’t necessarily need the complexity shown. Often, however, value binding is used when matching with tuples, where things get a little more complex and it can make more sense to use value bindings and `where` clauses.

In the value binding example there was actually an overlap across some of the cases, with the first case matching anything. This is legitimate. What will happen is that the first case statement that matches will execute, and the others will be ignored.

If-Case Statements

The capability of matching against patterns that has been described above for switch statements can also be used in the new Swift 2 statement, if-case:

```
let m = 5
if case 1...6 == m {
    print("m is within the range 1 to 6")
}
```

This is a very simple example. The value of the if-case statement is more often seen with more complex situations. Some programmers have used (with Swift 1) a switch statement with a single case (and the overhead of a default case, since switch statements must be exhaustive) because they wanted to use the more advanced pattern matching capabilities in switch. With Swift 2 they can use the simpler if-case statement instead.

For-Case Loops

Matching against patterns can also allegedly be done with the new Swift 2 statement, for-case.

Unfortunately, this capability does not work using the description in the Apple documentation, as of Swift 7.0 beta 2. Some people have been able to make it work with different syntax, but it is not clear whether Apple will change the compiler or change the documentation, so I haven't described it here.

When this is resolved I will post a description on the understandingswiftprogramming.com web site.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 20 exercises, go to

understandingswiftprogramming.com/20

21 Error Handling

San Jose Giants
vs Modesto Nuts
June 12



"If you spend the off-season throwing errors for the Android team, you'll throw out your arm. And you'll never make it to the majors."

In Swift 2 a sophisticated and well-designed system for handling errors has been introduced.

Swift was criticized early on for not having a try-catch error capability like many other languages. (The criticism is somewhat ironic because try-catch is not widely used in Objective-C, with the simpler `NSError` system typically used instead, when it is used at all.)

Even so, the Swift team has responded by producing an error handling system that is similar to the common try-catch schemes in other languages, but with significant differences. For example, code that can throw an error (exception) must be placed in a method that has the keyword `throws` as part of its declared syntax (and type signature). And such methods are called with the keyword `try`. Both of these help to clearly indicate what part of the code is throwing the error.

Handling Errors in Swift 1

In Swift 1, errors have been handled in a number of ways. One approach was to use the `NSError` object that is commonly used in Objective-C for error handling: When an API, in particular, was called, an `NSError` object was created (often called `err`), set to `nil`, and passed to the API. If the task was successful, the `NSError` object would remain `nil`. If there was an error, a description of it would be placed in the `NSError` object. The first step after the API call was accomplished was to test the `NSError` object for `nil`, and, if it was not, deal with the particular error.

Some developers used the enumeration capability in Swift to handle errors in a different way: An enumeration, typically called `Result`, was defined as follows:

```
Enum Result <T, E> {
    Case Success(T)
    Case Failure(E)
}
```

A task was then done with one of the enumeration values returned when it had completed. (That is, `Result.Success` or `Result.Failure`.) In the case of success, the type `T` contains some result of the success. In the case of failure, the type `E` contains a description of the particular error encountered. (`E` might be an `NSError` object).

In some cases, developers using these techniques initially reacted negatively to the new Swift 2 approach, indicating that the techniques they were using were fine. However, after some experimentation coding the new approach, these developers decided that the new approach resulted in safer, more readable code.

Swift 2's Approach

We can see how Swift's error handling works with a simple example. For example, an app might attempt to access a file on a remote server, and encounter one of the following errors:

1. The app cannot access the Internet.
2. The file on the remote server cannot be found.
3. The file has been successfully downloaded, but there is something wrong with the file.

I'll keep this simple with just these three errors, but obviously a lot more can go wrong in this situation.

These are recoverable errors (they don't crash the app) that most commonly will result in a message to the user of the app indicating the error so that user can respond. In other cases the recovery might involve just code. And in the case of an unrecoverable error the error handling code might terminate the app, hopefully with an informative message to the user.

Defining Errors as Enumeration Member Values

The first step toward implementing Swift’s error handling is to define an enumeration that includes member values for each of the possible errors:

```
enum FileError: ErrorType {  
    case cannotAccessInternet  
    case cannotAccessFile  
    case fileValidityError  
}
```

Note the `ErrorType` in the first line of the definition of the enumeration. This indicates that the enumeration conforms to the `ErrorType` protocol.

Executing Error-Prone Code in a Function/Method

In Swift, the code that is to be executed that might result in an error is placed into a function or method. The function or method is then called as part of a do-try-catch sequence.

The function or method that is executing the might-fail code has the keyword `throws` included in its definition, just after the input parameters. Thus:

```
func getFile(filename: String) throws -> String {  
    // Code to go to a remote server and  
    // get the contents of a file goes here  
}
```

The keyword `throws` indicates that the function or method has the ability to throw an error. “Throw an error” means that the function or method, when an error is found, executes a statement consisting of the keyword `throw` together with an enumeration value indicating the specific error. This causes the flow of control to immediately leave the function or method and to be processed by one or more catch statements.

Thus, for example, if the code in a function `getFile` determines that it cannot access the Internet, it might execute the following statement:

```
throw FileError.cannotAccessInternet
```

If the `throws` keyword is not included in the definition of a function or method, that function or method cannot throw an error.

The method can use if-else statements to throw errors:

```
func getFile(filename: String) throws -> String {  
    // Code to be executed that might produce an error goes here  
    // Before attempting to download let's make sure we have an  
    // an Internet connection; the app has a Boolean variable that maintains
```

```
// this information

if !gotInternetConnection {
    throw FileError.cannotAccessInternet
}

else {
    // Here we attempt to download a file from a server
    // by providing an iOS API with a URL
        // the file ends up as a string with fileContents
        // but if the file could not be found fileContents is nil
        // Likely an HTTP 404 but has been transformed.

    if fileContents == nil {
        throw FileError.cannotAccessFile
    }

    else {
        // Here we (after unwrapping it, it is an optional)
            // try to do something with the file, it is likely
            // JSON and we want to convert it to something easier
            // to deal with, but that conversion can fail
                if formatError {

        throw FileError.fileValidityError
    }

    // Success!
}

else {
    return "Success!"
    print("File downloaded OK and is in correct format")
}

// Do more stuff with the file
}

}

}

}
```

Using Guard Statements

The code above is correct but is a bit of a mess. This mess can be cleaned up with a guard statement, new in Swift 2.

It is often convenient to use guard statements rather than if-else statements. The guard statement works like if-else statement, but has a different syntax and flow. It looks like this:

```
guard != 0 else { throw Error.SpecficError }
print("x is not 0")
```

The guard statement is basically an if-else statement with the logic inverted. If the “if” condition is true, it does not execute an associated statement that is contained within curly braces. Instead, control passes to the statement just after the end of the “else” part of the guard statement. Note that there is no pair of curly braces analogous to those for the “if” statement. This makes sense, given where control is passed. It also means that any variables or constants created after the guard keyword will remain in scope after the end of the guard statement. And it makes for a nice, clean flow for the code.

The guard statement requires that the else part of the statement transfer control out of the method. (A throw will do that.) The else part of the statement is thus, obviously, required.

As I have discussed earlier, the pattern that guard statements make easy to create is sometimes called the “Bouncer Pattern”. The idea is to quickly get rid of the bad cases (like a bouncer for a bar does), then you can focus on what needs to be done for the good cases.

The following code does the same thing as the code shown earlier, but uses guard statements.

```
func getFile(filename: String) throws -> String {
    // Code to be executed that might produce an error goes here
    // Before attempting to download let us make sure we have an
    // an internet connection; the app has a Boolean variable that maintains
    // this information
```

```

guard gotInternetConnection else {
    throw FileError.cannotAccessInternet
}

// Here we attempt to download a file from a server
    // by providing an iOS API with a URL
    // the file ends up as a string with fileContents
    // but if the file could not be found fileContents is nil
    // Likely an HTTP 404 but has been transformed.

guard fileContents != nil
else {
    throw FileError.cannotAccessFile
}

// Here we (after unwrapping it, it is an optional)
    // try to do something with the file, it is likely
    // JSON and we want to convert it to something easier
    // to deal with, but that conversion can fail

guard !formatError else {
    throw FileError.fileValidityError
}

// Success!
return "Success!"

print("File downloaded OK and is in correct format")

// Do more stuff with the file
}

```

This is much cleaner than the previous code that used a sequence of nested if-else statements.

The Do-Try-Catch Sequence

When the function or method that contains the code in question is to be executed, it is done so using the keyword `try` in front of the function call:

```
try getFile("filename")
```

This call to the function or method is part of a do-try-catch sequence, as follows:

```
do {
    try getFile("filename")
        print("File retrieved with no errors")
    } catch FileError.cannotAccessInternet
    {
        print("Cannot access Internet—Make sure you have an
        Internet connection. ")
    }
    catch FileError.cannotAccessFile
    {
        print("Cannot access requested file on remote server—Make
        sure you have provided the correct name for the file.")
    }
    catch FileAccessError.fileValidityError
    {
        print("There is something wrong with the specified file—
        Make sure that this file is correct.")
    }
}
```

Note that there is only one `try` statement, but multiple `catch` statements. This is different from languages like Java, which use `try-catch` statements in pairs, with only one `catch` for each `try`.

If the `getFile` call succeeds with no throwing of an error, any code following that call will be executed, such as the `print` statement shown in the example.

However, if an error is thrown, statements after the call will not be executed, and control will pass immediately to the appropriate catch statement.

It is common for the catch clauses to simply use the same enumeration value that was used in the throw statement. However, the catch clauses can use the same sophisticated matching capabilities that switch statements have if desired.

If you are sure that a function or method will not execute code that results in an error, you can use the `try!` keyword instead of `try`. This will disable the throwing of any errors.

Defer

The `defer` keyword allows you to define code that will be executed just before the flow of execution leaves the current scope. (It doesn't matter how the current scope was exited —by returning from a function or method, generating an error/exception, or just reaching a right curly brace.)

Thus:

```
defer {    close("filename") }
```

is a common use, since you usually want to close a file once you've read from it, or attempted to read from it, regardless of what happened.

This code (analogous to the `finally` statement in languages like JavaScript), makes sure that this code is executed, regardless of whether an error has been thrown or not.

It is allowed to have multiple `defer` statements, each with a block of code that you absolutely need to get executed, whatever happens.

The `defer` statement makes it easy to put code that is related close together. It helps avoid errors, since it is no longer necessary to put cleanup code (or calls to it) in multiple branches, depending upon what happens.

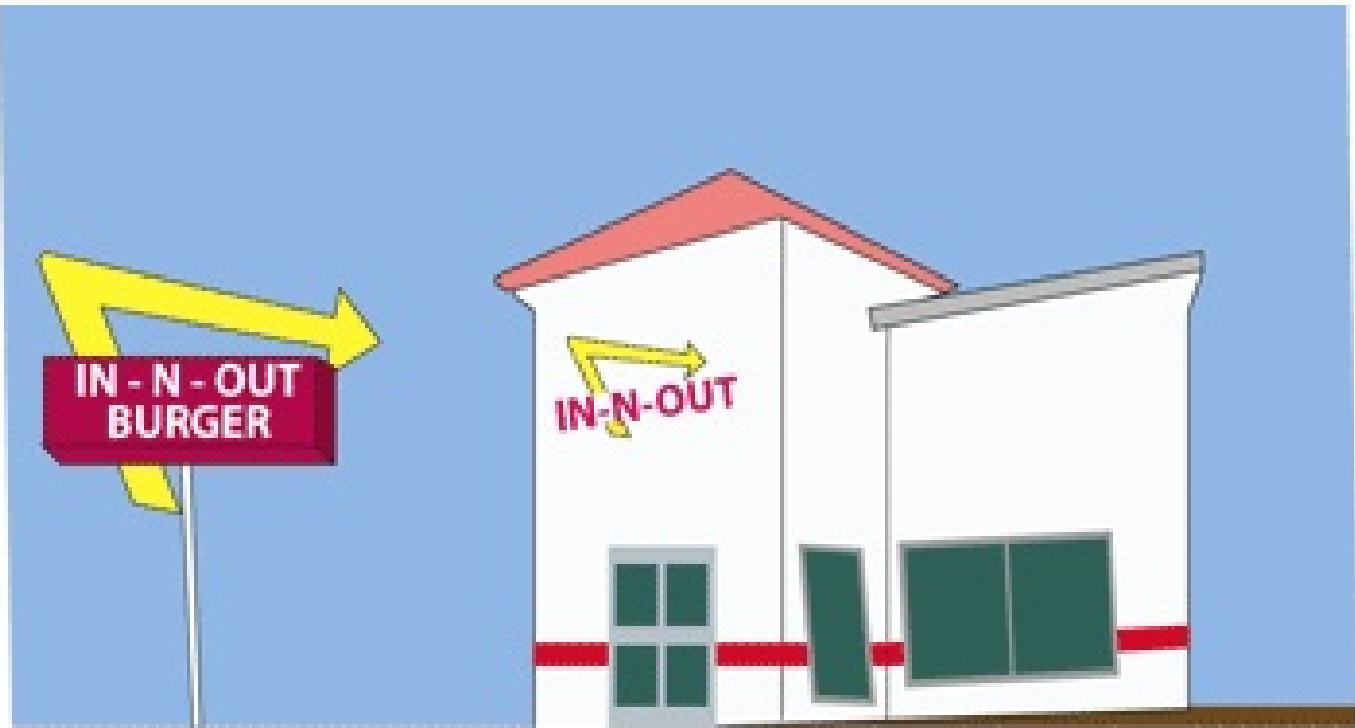
Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 21 exercises, go to

understandingswiftprogramming.com/21

22 Functions Revisited Again: Input Parameters



"I can't figure out why our programmers are using so many `inout` parameters in their function calls. It's the coders who stay up all night drinking coffee and eating burgers who are doing it. I suspect that they are barely awake when they are writing code. It's a real mystery."

Earlier chapters introduced functions and methods and described their basic syntax (motivated by the syntax of Objective-C), that uses named parameters. (In Swift 2 named parameters are used in both functions and methods. In Swift 1 named parameters are only used in methods.)

Functions and methods also have many possible variations allowed for input parameters. The purpose of the present chapter is to describe these variations.

This chapter will cover five different possible variations for input parameters. They are:

1. The use of external parameter names, which are the default for both functions and methods in Swift 2.
2. Variable parameters. By default, input parameters are constants. By declaring them as variables, they can be modified within the function, although they are not available outside the function.
3. In-out parameters, which allow external variables to be passed by reference to the function, modified within it, and returned.
4. Default parameter values. Default values can be specified for input parameters within the input parameter parentheses.
5. Variadic parameters. A variable number of parameters is allowed for one type of the input parameters.

An example of the most basic form of a Swift function is shown below:

```
func addTwoNumbersWithFirst (first: Int, second: Int) ->
Int {
    var c = first + second
    return c
}
```

```
var d = addTwoNumbersWithFirst(2, second: 5)
```

A function is allowed to have no parameters at all, but must still, in that case, have parentheses, even though they are empty. (This requirement is unlike that for the return arrow and type for the return value, which can both be omitted entirely if there is no return value.)

```
func functionNoInputsOrReturn() {
    print("Hello")
}

var e = functionNoInputsOrReturn()
```

Variations in External Parameter Names

The syntax for the example of a basic function presented above has two input parameters, and the input parameter syntax declares two constants, `a` and `b` to be parameters. These are *local parameter names*, and their purpose is to allow the parameters to be referenced within the body of the function.

```
func addTwoNumbers (a: Int, b: Int) -> Int {  
    var c = a + b  
    return c  
}
```

In Swift 2, these local parameter names are used as the default external parameter names for calling functions and methods:

```
var d = addTwoNumbers(2, b: 5)
```

This doesn't make a lot of sense from the viewpoint of the calling statement, however. We can fix this by changing the name of the function and adding an external parameter name for the second parameter:

```
func addTwoNumbersWithFirst (a: Int, second b: Int) ->  
Int {  
    var c = a + b  
    return c  
}
```

which is called with:

```
var d = addTwoNumbersWithFirst(2, second: 5)
```

The local name for the second parameter is still `b`, but now there is an external name, `second`.

Note that the external parameter names don't mean anything within the body of the function.

It's possible to define an external name for the first parameter of a function or method, if you really want to:

```
func addTwoNumbers (first first: Int, second: Int) ->
Int {
    var c = first + second
    return c
}
```

This is called with:

```
var d = addTwoNumbers (first: 2, second: 5)
```

In Swift 1, you can do the following. You use the name just once, but put a “#” character just before the name:

```
func addTwoNumbers (#first: Int, # second: Int) -> Int {
    var c = first + second
    return c
}
```

However, the hash symbol does not work for this purpose in Swift 2. You have to specify the name of the first parameter twice. In the case of the second (and further) parameters, you need specify it only once.

It should be recognized that the parameter names and values must be in the order specified in the function definition. Just because you are using a name for each parameter does not mean that you can put them in any order.

In Swift 2, you can declare a function so that external parameter names are not required, using an underscore character:

```
func addTwoNumbers (a: Int, _ b: Int) -> Int {  
    var c = a + b  
    return c  
}
```

This is called with:

```
var d = addTwoNumbers (2, 5)
```

In the case of the first parameter, the parameter name in the call is not required by default.
In the case of the second parameter, the underscore indicates that a parameter name is not required (and cannot be used.)

Making an Input Parameter a Variable

It is not possible, in the default case, to modify the value of an input parameter of a function from within that function. An input parameter is by default a constant. However, the input parameter can be modified if it is declared a variable by using the `var` keyword before it in the definition of the function:

```
func addTwoNumbers (var a: Int, b: Int) -> Int {  
    a = a + 5 // Now allowed to modify a  
    var c = a + b  
    return c  
}
```

Thus, the input parameter with the name `a` can now be modified.

This can also be done if an external parameter name is used:

```
func addTwoNumbers (var first a: Int, b: Int) -> Int {  
    a = a + 5 // Now allowed to modify a  
    var c = a + b  
    return c  
}
```

In-Out Parameters

Normally, a function cannot change the value of a variable that is passed to a function as an input parameter, at least when accessing it as an input parameter. And it is not a good idea for a function to be modifying variables that are in the scope of its calling statement —that breaks the usual assumption of loose coupling and modularity that object-oriented programming is supposed to be based on.

If, however, we want to do this we can do it in a clear and explicit way by declaring a parameter as an *in-out parameter*.

Thus, if our `addTwoNumbers` function is called as follows:

```
var m = 5
var d = addTwoNumbers(2, b: m)
```

The `addTwoNumbers` function cannot change the value of `m`.

However, if we, in the `addTwoNumbers` function, add the keyword `inout` before the name of an input parameter, that parameter becomes an in-out parameter. The variable appearing in the calling function corresponding to the in-out parameter can now be modified by the function.

```
func addTwoNumbers ( a: Int, inout b: Int) -> Int {
    b = 7
    var c = a + b
    return c
}
```

The way that we call it is slightly different. We use an ampersand in front of the variable name for the in-out parameter because we are passing it by reference (rather than making a copy):

```
var m = 5
```

```
var d = addTwoNumbers(2, b: &m)
```

In this situation, we've changed the value of `b` (which was 5) to 7. And so the function will return the value of 9, rather than 7. But beyond this, because we have declared `b` as an in-out parameter, the variable (outside the function) that was in the calling statement, `m`, will also have its value changed to 7.

```
print(m) // Prints: 7
```

Setting Parameters to Default Values

It is possible to set a default value for an input parameter, by appending an equals sign and a value to the definition of an input parameter:

```
func addTwoNumbers (a: Int, b: Int = 7) -> Int {  
    var c = a + b  
    return c  
}  
  
var d = addTwoNumbers(1)  
print(d) // Prints: 8
```

Parameters that have default values should be put at the end of the list of parameters. If an input parameter does not have an external name, Swift will assume one based on the local name so that it can be clear which parameter is being included, should the call provide a value for a parameter that has a default supplied in the function definition.

Variadic Parameters

A variadic parameter has a variable number of input values.

A variadic parameter is indicated by specifying an ellipsis (three periods, or dots) after the input type.

I have modified the `addTwoNumbers` function to create a function named `addAllNumbers` that can take any number of input values:

```
func addAllNumbers (numbers: Int...) -> Int {  
    var c = 0  
    for number in numbers {  
        c = c + number  
    }  
    return c  
}  
  
var d = addAllNumbers(1, 2, 3, 4)  
print(d) // Prints: 10
```

The original version of this function specified its input parameters as `(a: Int, b: Int)`. In the new function, instead of listing specific input parameters, the name `numbers` refers to an array. The type is specified with three periods following it to indicate that this is a variadic parameter. All of the input parameter values that are part of a variadic parameter must be of the same type.

In the body of the function, individual parameters are referred to by addressing elements in the array, starting with an index of 0. Or the entire array can be iterated through, as shown here.

If there are only two numbers, the function still makes sense.

```
var d = addAllNumbers(2, 5) // Result is 7
```

But there can be any number of input parameters:

```
var d = addAllNumbers(2) // Result is 2  
var d = addAllNumbers(2, 7, 8, 34, 54) // Result is 105  
var d = addAllNumbers() // Result is 0
```

There can be only one variadic parameter for a given function. If there are other input parameters, the variadic parameter must come last.

Input Parameter Syntax for Functions

Variations:

External parameter names

Variable parameters

In-out parameters

Default parameter values

Variadic parameters

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 22 exercises, go to

understandingswiftprogramming.com/22

23 Variations in Closure Expression Syntax

In an earlier chapter (Chapter 18 on Closures and Closure Expressions) I discussed how closures, functions, and closure expressions were related and how closure expressions were used.

You saw the full closure expression syntax:

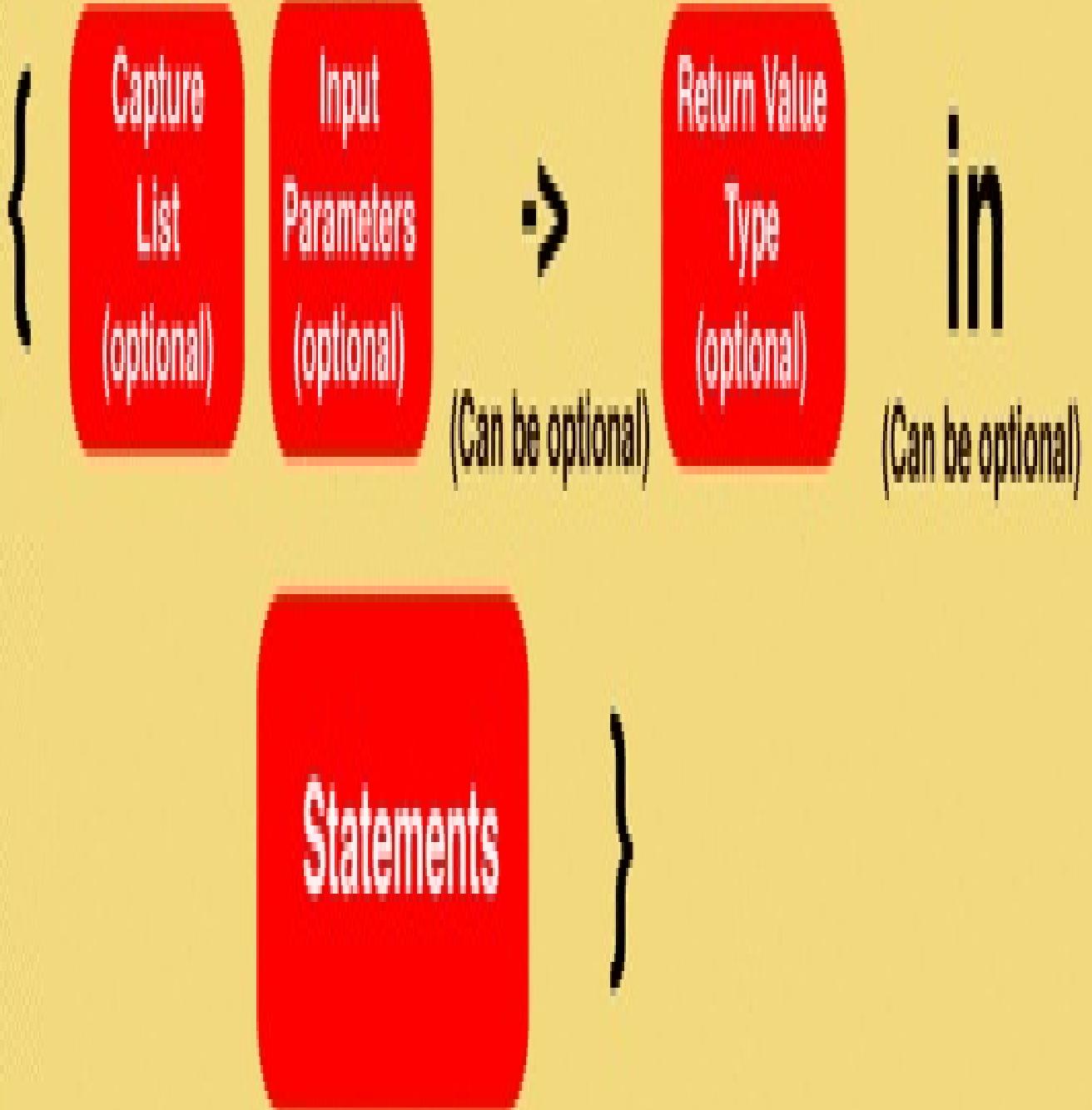
```
var d = { (a: Int, b: Int) -> Int in
  var c = a + b
  return c
}

var m = d(3, 4)
print (m) // Prints: 7
```

As well as the simplest:

```
var a = { print("Hello") }
a() // Prints: Hello
```

And this illustration that describes the syntax:



I said at that time that closures were very flexible in allowing a wide variety of syntax variations. The purpose of this chapter is to describe these variations and provide some hint about how the Swift compiler goes about accepting these variations.

Apple has put a lot of effort into a compiler that is quite intelligent in the way that it parses expressions for closures, and allows the programmer a lot of latitude.

This is mostly, but not necessarily entirely, a good thing. Even though it is possible to leave out much or even nearly all of the syntax for a particular closure, you should think about whether you should. Will the resulting code be readable? What makes perfect sense (and saves some typing) when you are doing a lot of coding with closures may not make much sense six months later when you have been working on something else for some time. Not to mention co-workers who may not have your level of skill in Swift who may end up working on your code.

The standard example—used in the Apple documentation—for describing the flexibility of Swift closure expression syntax is based on the Swift sorting function. This is an interesting example because it not only demonstrates how to get from the full closure syntax to the minimal form, but to an unusually freakish minimum of only a single character, a “<”, telling the sort function what order it should sort things in.

Sorting An Array with a Closure Expression with Full Syntax

We begin with an array of strings that each represent the name of one of five states:

```
var states =  
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
```

And we want to use a global Swift function known as `sorted` to sort them. The function `sorted` is a slightly more complex version of `sort`, which allows more control over how the sorting is done. The `sort` function just accepts the array to be sorted as its only input value. The `sorted` function accepts the array to be sorted, plus a second input parameter that is designed for either a closure expression or a function.

The code to sort with a closure expression is:

```
var statesSorted = sorted(states,  
{ (item1: String, item2: String) -> Bool in  
return item1>item2  
})
```

The global function `sorted` has two input parameters. The first is `states`, which is the array we just assigned that has names of states. The second argument is a closure expression. The closure expression is surrounded by the usual braces and begins with input parameters:

```
(item1: String, item2: String)
```

As we saw from the earlier chapter, it can get a little confusing when a function has what seems to be an “input parameter”, but then that parameter turns out to be a closure or function, that itself has inputs and outputs. So this “input parameter” actually puts *out* values? Well, no. But the closure that gets supplied to the function `sorted` has inputs and outputs, and the function `sorted` does supply values to the closure.

The values that the function supplies to the closure are a pair of strings in a particular order. And what it wants in response is a simple Boolean value that indicates whether the strings have been provided in the correct order (`true`) or not (`false`).

Whenever the sorting function needs guidance about what order to sort (how it does this and how many times it does it depends on the sorting algorithms the function uses internally) it calls the closure expression with an ordered pair of strings (the names of the states, presumably) and then looks at the Boolean value of the output.

The closure expression has the expected return arrow (`->`), return type (`Bool`), and `in` keyword.

It also has a return statement that has the expression `item1>item2`. The `>` operator is actually the critical thing here—how it works in comparing whether one string is “greater than” another is what determines how the sorting is done.

When I ran this code I got the following:

```
"Iowa", "Florida", "Delaware", "Connecticut", "Alaska"
```

So assuming that we want alphabetical order, our sorting works but it is backwards. So we change the “greater than” character to a “less than” symbol:

```
var statesSorted = sorted(states,
{ (item1:String,item2:String) -> Bool in
return item1<item2
})
```

Which yielded:

```
"Alaska", "Connecticut", "Delaware", "Florida", "Iowa"
```

The desired result.

Successively Simplifying the Syntax

We can now go through a succession of simplifications of the syntax.

Input Parameter Types Not Required. In general, when providing a closure as a parameter in a function, it is not required to indicate the type. The function knows what type it will accept; the compiler can infer the type from what is returned (at least at this stage.) We can thus drop the input parameter types:

```
states =
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
statesSorted = sorted(states,
{ (item1, item2) -> Bool in
return item1 < item2
})
```

Input Parameter Parentheses Not Required. The parentheses around the parameters are also not required when providing a parameter in a function. Given the remainder of the closure expression, it is clear to the compiler what is going on, and the parentheses around the input parameters can be dropped. Thus, the following is correct syntax:

```
states =
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
statesSorted = sorted(states, {
item1, item2 -> Bool in
return item1 < item2
})
```

Return Keyword Not Required if Single Expression. If a closure contains only a single expression, and a type is defined for a return value, then there is no need for a `return` keyword, because it is clear that a return value must be provided and it is clear which expression (there being only one) should be executed to calculate such a return value. Thus, the following is correct:

```
states =
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
```

```
statesSorted = sorted(states, {
    item1, item2 -> Bool in
    item1 < item2
})
```

Trailing Closure Syntax. Normally, the input parameters for a function are contained within a pair of parentheses. However, in the special case in which an input parameter comes last (that is, is the “trailing” parameter), there is an alternative syntax allowed: The right (closing) parenthesis comes just at the end of the next-to-last input parameter states, and the left brace of the closure comes just after the closing parenthesis. (Blank spaces are allowed after the parenthesis). Thus, the following is correct syntax:

```
states =
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
statesSorted = sorted(states) { item1, item2 -> Bool in
    item1 < item2
}
```

Some functions (not this one) have only a single parameter, that for the closure, and thus the parentheses with a trailing closure would be empty. In such a case, the parentheses can be dropped.

Return Arrow and Type Not Required. It’s clear from the input type expected by the sorted function that a Bool should be returned. We can drop the return arrow and the type and the compiler will figure it out:

```
states =
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]
statesSorted = sorted(states) { item1, item2 in
    item1 < item2
}
```

Shorthand Input Parameters and Dropping in Keyword. Rather than providing a name for each input parameter that is then used in the closure statements to refer to a particular input parameter, it is permissible to use instead a shorthand symbol to refer to each input parameter. The first input parameter is referred to by the symbol \$0, while the second input parameter is referred to by the symbol \$1, etc.). (This is also called an *automatic*

argument name.) We can at the same time drop the `in` keyword. Thus, the following is correct syntax:

```
states =  
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]  
statesSorted = sorted(states) { $0 < $1 }
```

Dropping Everything But a Single Comparison Operator. In the ultimate example of simplification, we drop the trailing closure syntax and revert to the conventional syntax, but pass only a single comparison operator to the function.

```
states =  
["Delaware", "Alaska", "Iowa", "Florida", "Connecticut"]  
statesSorted = sorted(states, <)
```

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 23 exercises, go to

understandingswiftprogramming.com/23

24 Custom Subscripts

It is often the case that subscripts are useful for quickly and easily accessing data that is stored in the form of an array, string, or dictionary. Thus, in the example below we can quickly access the second item in the array `airplanes` with the expression `airplanes[1]`:

```
let airplanes = ["Boeing 777", "Airbus 320", "Beechcraft  
Bonanza"]  
print(airplanes[1])
```

In this case the second statement would print out “Airbus 320”.

(As in most languages the index to an array starts at 0.)

A subscript is just an identifier that is contained within square brackets. The value used in a subscript is often an integer, as seen above, but it can also be a string, and a string is often used with a dictionary. In the case of a dictionary the value used in the subscript is known as a *key*.

Using a subscript is just a slightly more convenient way to access information. In the code above I used `airplanes[1]` to access “Airbus 320”, but I could also have written a function that allowed the same access by, say, `getAnAirplane(1)`.

Usually, you can put data directly into arrays or dictionaries and access it with subscripts. But there are situations in which it is desirable to create your own custom subscripts, which is easy to do.

I’ve tried to make up an example that makes some sense that just uses a single subscript so it is simple.

Suppose we are very interested in everyone’s Top Ten Movies, and so we create a class called `EveryonesTopTenMovies`. Each instance of the class `EveryonesTopTenMovies` represents a person. We have a little bit of interest in who these people are, so we do have a class definition that looks like this that includes properties that store people’s names and phone numbers:

```

class EveryonesTopTenMovies {
    var name: String = ""
    var phoneNumber: String = ""
    var movies:[String] = []
}

```

We can create an instance of this class for a person and set their name and phone number as follows:

```

let johnsTopTenMovies = EveryonesTopTenMovies()
johnsTopTenMovies.name = "John Cooper"
johnsTopTenMovies.phoneNumber = "415 328-1876"

```

Now obviously we can set their movie choices by using the property `movie`, but the idea here is to use subscripts more directly. So we add the subscript code to our class definition:

```

class EveryonesTopTenMovies {
    var name: String = ""
    var phoneNumber: String = ""
    var movies: [String] = []
    init() {
        for var i = 0; i < 11; i++ {
            self.movies.append("No movie set for item \\" + (i+1) + "\"")
        }
    }

    subscript (index: Int) -> String {
        get {
            if(index < 1 || index > 11) {
                return "No 0th or 11th ranked movie!"
            }
        }
    }
}

```

```

else {
    return movies[index-1]
}

}

set(newValue) {
    if(index < 1 || index > 11) {
        print("No 0th or 11th ranked movie!")
    }
    else {
        movies[index-1] = newValue
        // was nameOfMovie
    }
}
}
}

```

Note that the subscript definition is very much like a function or method definition. The main differences are that it uses the keyword `subscript` rather than `func`, the fact that it does not have a name, and that it has code that gets executed as either a getter or a setter, depending upon how the subscript is used. But the subscript definition does have input parameters with the same formatting rules as a function, and it has a return arrow and a type for the return value.

Another way to look at a subscript is that it works very much like a computed property.

As before, we create a new instance of the class `EveryonesTopTenMovies`:

```
let johnsTopTenMovies = EveryonesTopTenMovies()
```

Store a name into it:

```
johnsTopTenMovies.name = "John Cooper"
```

And some movies:

```
johnsTopTenMovies[1] = "Batman"  
johnsTopTenMovies[2] = "Erin Brockovitch"
```

As a result of our defining our own class with subscript access, we can be finicky about things. Here “Batman” is John Cooper’s No. 1 most favorite movie. And we used a 1, not a 0, to access it. It’s John’s 1st most favorite movie, not his 0th!

```
print ("John's most favorite movie is\  
(johnsTopTenMovies[1])")
```

Custom subscripts can be used for structures and enumerations as well as classes.

A class (or structure or enumeration) can use operator overloading for subscripts in the same way that it does for operator overloading with a function.

Syntax for Custom Subscripts

subscript **Symbol** **Input
Parameters** → **Return
Value** {

get {

Statements

 }

set **New
Value** {

Statements

 }

}

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 24 exercises, go to

understandingswiftprogramming.com/24

25 Operators Revisited: Overloading and Custom Operators

In Swift, custom operators can be defined. If we would like to use a particular symbol to cause the execution of a particular piece of code, we can define the symbol as a custom operator and the necessary related operands.

Swift can also do *operator overloading*, in which the same operator is defined to work with more than one group of operands. The pattern of types of the operators in a particular situation determines which definition of the operator will be used for that particular situation.

Creating a Custom Operator

A custom operator in Swift is created in two steps. First, the operator is declared, which specifies the character or characters used for the operator and some parameters. And second, a function is implemented for the operator that performs the actual operation that the operator represents.

Suppose we'd like to implement a square root operator that uses the Unicode square root symbol $\sqrt{}$. There's a square root function already existing in Swift, `sqrt`, but we think it's nicer to use the symbol.

The operator declaration requires the symbol, the keyword `operator`, and whether the operator is to be a prefix, infix, or postfix operator. It also can specify the precedence and associativity (parsing direction or none). In the case of a unary operator, or operator that takes only one operand, we don't need to specify the precedence or associativity.

We first need to import the `Foundation` framework so that the `sqrt` function works:

```
import Foundation // Needed for sqrt function
```

(Alternatively, `UIKit` can be used, which includes `Foundation`.)

We then use the following line of code to declare the operator:

```
prefix operator √{ }
```

(The Unicode square root symbol $\sqrt{}$ can be entered on a Macintosh by pressing the V key with the option key held down.)

The function that implements the operator is as follows:

```
prefix operator √{ }

prefix func √(x: Double) -> Double {
    return sqrt(x)
}
```

```
print(√5) // Prints: 2.236067
```

(If you are entering this in the interactive REPL, you must enter the operator declaration just before the function definition, with no delay.)

The `prefix` keyword is necessary to indicate that this implements the prefix form of the `√` operator. We could also define another function in the same way but with the `postfix` keyword. This would take the square root of an operand that came before the `√` operator in a statement.

Like any function, it has a name, and this is just the symbol for the operator. The rest of the implementation is just like an ordinary function: A single input parameter is defined of type `Double` for `x`, and a return value is defined of type `Double`. The code in the function makes a call to the Swift square root function, using the value from the input parameter, and then returns the result.

Syntax for Operator Overloading and Custom Operators

1. Define the operator symbol and position:

Position

operator

Symbol

()

2. Define the function to be run, using the symbol as the name, and add the position:

Position

func

Symbol

Input
Parameters

->

Return
Value

{

Statements

}

Operator Overloading

To add an additional definition of an operator—and thus overload it—you only have to provide a definition in the same way that was just shown for creating a custom operator.

The difficult part of dealing with operator overloading isn't writing the code. It is deciding what operators you want to overload and how the functions are defined. You don't want to overlap definitions or be ambiguous about them. They should also make sense (e.g., see Matt Thompson's comment about the addition operator in Chapter 4).

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 25 exercises, go to

understandingswiftprogramming.com/25

26 Optional Values Revisited

In an earlier chapter (Chapter 9 on Optional Values) I described the basics of optional values. Optional values are a way of ensuring that variables that can have no value—that is, that can be set to `nil`—are handled with a high degree of safety. This is done by defining an optional version of a data type if a variable is going to contain `nil`. If a non optional (ordinary variable) is set to `nil`, a runtime error will result. I also said that a value assigned to an optional data type will be wrapped, and that it must be unwrapped before it can be used. I then described some different ways of unwrapping the value, including the preferred method of optional binding.

In this chapter we revisit optional values, and discuss a few more capabilities—the use of implicitly unwrapped optionals, the use of the `nil` coalescing operator, the use of optional chaining, and optional binding for multiple values.

Implicitly Unwrapped Optionals

An alternative to the basic wrapped optional is an *implicitly unwrapped optional*. This is used only when you are absolutely sure that a variable has a value. It is often used in situations such as a property defined in a class as a `nil`, but actually initialized with a value as part of an `init()`, perhaps when an instance is created with a parameter that sets the property. Care is then taken to not thereafter set the property to `nil`.

It is also common for the Cocoa Touch API to return optionals that are guaranteed to have a value, and these can be declared as implicitly unwrapped optionals.

A variable is defined as an implicitly unwrapped optional by using type annotation and providing an exclamation point (“`!`”) just after the name of the type:

```
var n: Int! = 5  
var p = n
```

Here is a very simplified example—the variable `n` is declared as an integer implicitly unwrapped optional with a value of 5 assigned. When the value is then assigned to the variable `p`, it is not necessary (or allowed) to do anything to unwrap it.

If, however, an implicitly unwrapped optional is accessed that does not have a value, there will be a runtime error.

A more complicated example is shown below:

```
class bird {  
    var species: String! = nil  
    init(species: String) {  
        self.species = species  
    }  
    let aParticularBird = bird(species: "Mallard Duck")
```

In this case we have a class that stores information about particular birds, namely, the

name of their species. We define a property, `species`, as a variable with a type of an implicitly unwrapped optional string, with a value of `nil`. The initializer accepts a parameter when a new instance is created and sets the value of the property `species` (accessed as `self.species`) to the value contained in the parameter named `species`.

What happens in the last line is that a new instance is created, and the property `species` is first initialized to be `nil`. When the initializer is executed, however, the property is then set to the value of “Mallard Duck”.

If this property is accessed, unwrapping is not required:

```
print("The species is \$(aParticularBird.species)")
```

If it is later discovered that the species name is not correct, it is not allowed to simply set the value back to `nil` (for unknown). If the correct species name is not available the instance must be deleted.

Nil Coalescing Operator

Another approach to optionals ensures that a variable has a value, either that contained in an optional variable or a predefined default value. It is quite safe because unwrapping is done automatically and there is no possibility of a runtime error.

This is done with a *nil coalescing operator*. This operator is a double question mark and is used with an optional variable and a default value.

```
var errorMessageFromDisk: String? = "Bad File Sector"  
let defaultMessage = "Something Bad Happened"  
  
let errorMessage = errorMessageFromDisk ??  
defaultErrorMessage
```

Here the first line declares and sets an optional variable with an error message. (We're pretending that we got this as the result of attempting to read a disk.) This is what might reasonably come from a file system. However, the variable storing the message is declared as an optional, because the file system might not always know the specifics of the error, in which case it would return a nil.

A default error message is defined, and then the nil coalescing operator is actually used.

In the case shown, errorMessage would be set to "Bad File Sector" after it was unwrapped from the optional version. This is because the left hand side of the ?? nil coalescing operator has a value. However, suppose the variable errorMessageFromDisk was nil. In such a case, errorMessage would be set to the default message, "Something Bad Happened", by the nil coalescing operation.

This is equivalent to the following more verbose, and less safe, statements:

```
var errorMessageFromDisk: String? = "Bad File Sector"  
let defaultMessage = "Something Bad Happened"  
  
if errorMessageFromDisk == nil {
```

```
let errorMessage = defaultMessage
else {
let errorMessage = errorMessageFromDisk!
}
```

The name of the operator is a little obscure. “Coalesce” means to “come together”, “combine”, or “form one mass or whole”. Thus presumably the coalescing operator allows the original optional value and the value that gets set if the original value is `nil` to “come together” and form a new variable or constant that contains the information that is relevant, given the situation. Of course, lots of operations do this sort of thing. The name isn’t really very enlightening about what it does. (It’s hard to name operators.)

Optional Chaining

In situations in which there is more than one optional in a sequence of property relationships to unwrap, it takes less code and is more readable to use *optional chaining* for the unwrapping of optionals.

This accomplishes the same thing as optional binding. For optional chaining to have any effect, all of the values that are marked as optionals in the chain must in fact have values. If any of them are `nil`, the optional chaining statement does nothing.

To refresh your memory about how optional binding works, we can see it here:

```
var breedOfDog: String? = "Border Collie"
```

We have an optional with a value; we now have to unwrap it:

```
if let breed = breedOfDog {  
    print("Breed of dog is \(breed)")  
}  
  
else {  
    print("breedOfDog has no value")  
}
```

This works fine if there is just one value to test for `nil`.

Now consider a more complex situation. Suppose we have a person who owns a dog, and `breedOfDog` is a property of the class `Dog`.

```
class Person {  
    var dog: Dog? = nil  
}  
  
class Dog {  
    var breedOfDog: String? = nil  
}
```

```
var george: Person? = Person()  
george!.dog = Dog()  
george!.dog!.breedOfDog = "Border Collie"
```

We've created an instance of a `Person`, and assigned it to a variable `george` that is an optional `Person` (`Person?`). We've also created an instance of a `Dog`, and assigned it to a property of the variable `george`. And we have created a value for a property of `Dog` where the instance of `Dog` is now contained in a property of `george`.

All of these variables and properties have values, once all the code has been executed. However, they are all declared as optional versions of whatever type they are, and at any point after these values are assigned any or all of them could have `nil` assigned to them. Thus we need to take care in unwrapping them. Note here that unwrapping is just as necessary when using an optional on the left side of an assignment statement as it is on the right side. In the code above I have used forced unwrapping, which is a bad idea. I should use optional binding.

Before unwrapping the optional value of `breedOfDog`, we have to make sure that the other variables in the dot syntax chain have values.

The preferred way to do this is with optional chaining. For the example above, this requires:

```
if let breed = george?.dog?.breedOfDog {  
    print("If everything has a value, breedOfDog is \(breed)")  
}
```

The question mark at the end of each variable, known as the *chaining operator*, indicates that the property or variable in question is an optional, and that the value should be unwrapped if it exists. If the value does not exist then the optional chaining should stop immediately. This is the main difference between using forced unwrapping (with a “!”) and optional chaining (with a “?”). With forced unwrapping, if any of the optional values is `nil`, you will get a runtime error. With optional chaining, the statement will simply be ignored.

The type of a value that is returned from an optional chaining statement is itself an optional.

If a particular variable or property in the chain is not an optional, it does not have a question mark at the end of it, and does not need to be unwrapped. Its value is assumed to exist because otherwise the system would have generated a runtime error.

Optional chaining is common for relationship chains in apps themselves, and especially common when a relationship chain is retrieved from an API like Cocoa Touch.

Optional chaining can be used to write to optional values as well as to read from them, and they can be used for methods and subscripts as well as properties.

Optional Binding for Multiple Values

In situations in which there is more than one optional value that needs to be unwrapped at one time, this can be done, beginning in Swift 1.2, compactly with a simplified syntax.

Suppose that `a`, `b`, and `c` are all declared as optionals and have all been assigned a true value. They can be unwrapped by the code below. First, the optional declarations:

```
var a: Int? = 5
var b: Int? = 6
var c: Int? = 7
var d: Int? = 8
```

Then the code:

```
if let a = a, b = b, c = c, d = d {
    print("All of the optionals a, b, c, and d have values")
}
```

The syntax above does the same thing as the more wordy:

```
if let a = a {
    if let b = b {
        if let c = c {
            if let d = d {
                print("All of the optionals a, b, c, and d have values")
            }
        }
    }
}
```

It's also possible to add a `where` clause to do an additional test beyond making sure that

all of the optionals have values:

```
var a:Int? = 5
var b:Int? = 6
var c:Int? = 7
var d:Int? = 8
var m = 5
if let a = a, b = b, c = c, d = d where m > 4 {
    print("M is > 4 and all of the optionals a, b, c, and d
have values")
}
```

Optionals Syntax

x: Int? Declaring a variable as an optional

x! Forced unwrapping an optional (used on a variable that has a value)

Testing an optional for nil

```
if x!= nil {  
    print("The value of x is \(x!)")  
} else {  
    print("x has no value") }
```

Optional binding

```
if let x = x {  
    print("The value is \(x)")  
} else {  
    print("There is no value") }
```

x: Int! Implicitly Unwrapped Optional

?? Nil Coalescing Operator

x? Optional Chaining Operator

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 26 exercises, go to

understandingswiftprogramming.com/26

27 Access Control

Swift includes a capability for controlling access to particular pieces of code and particular types, properties, methods, and the like.

This is most commonly used for frameworks that you create that are used (by importing the framework module) by your apps.

Developing your own framework is a common practice if you have code that you want to use in more than one app. And if it is generally useful you may want to provide it to other developers.

In defining a framework, it is frequently the case that you want users of the framework to be able to access only certain entities (e.g., classes, methods and properties) that you define as the official API for the framework. You typically don't want developers using the framework to be able to access other entities that you want to be able to change without giving notice. This is done by controlling access. Typically, the official API is designated as `public` while other capabilities are designated as a level less than `public`.

Access is controlled based on two factors. One factor is where the code you have imported and are attempting to run came from. The other factor is the level of access allowed for a specific type depending upon tagging using three defined tags: `public`, `internal`, and `private`.

Where the Code Came From

When you import code into a program, that code comes from somewhere. It might come from a different module than the module where your program is, it might come from the same module (but a different file), or it might come from the same module and the same file where your program is.

The table below shows what access level is required (as specified in the code for particular entities) to gain access to code, depending upon where the code came from. A checkmark means that access is allowed for that particular access level—public, internal, or private.

Access Control Level Specified in Code

Where the
Code Came
From

Public Internal Private

From a Different
Module



From Same Module /
Different File



From Same
Module / Same File



There are three situations.

First, if you are running code imported from a different module, the level of the entity you are trying to access in that code must be `public`. If the level is `internal` or `private`, you will not be able to access it. (See below for how this level is determined.)

Second, if you are running code in the same module but a different file as your app, the level of the entity you are trying to access in the code being imported must be either `public` or `internal`. If the level is `private`, you will not be able to access it.

And third, if you are running code in the same file as your app, access is always allowed, regardless of the level of access defined in the code.

How the Code is Tagged

Every entity (meaning type, method, or property) has an level of access control level associated with it. This is determined by what can often be a fairly complex set of rules that can be different for specific types.

However, a few general rules will suffice for most developers in most situations. First, if you are building an app, rather than a framework, you will be writing code that is in one of the files in the module that is the target. If you do nothing in this code, all of the code will be tagged as having an access level of `internal` (the default), and that code will run regardless of where it is as long as it is in the same module. So you will not have to worry about access levels. In building your app you may use standard frameworks that you import. If they have been properly defined, the classes, methods, and properties that you access will be part of the official defined API for that framework, and those entities should have the access level of `public` and the code in your app will be able to access them.

If you build a framework, or need to use more restrictive access levels in your app, you may need to understand some of the more detailed rules about access control, and you should consult the Apple documentation.

A general rule about access is that the access level for an entity cannot be less restrictive than the access level of the *most restrictive* access level of the components for the entity. The previous sentence is pretty hard to parse, so to make it clearer, see the following example.

First, to understand the example, let's say that you've defined a custom class and created an instance of that class and then called a function:

```
private class Trout {  
    var species = "rainbow"  
}  
  
func printSpecies () {  
    print("Species is \(aParticularTrout.species)")  
}
```

This is called with:

```
var aParticularTrout = Trout()  
  
aParticularTrout.printSpecies()
```

So, we have defined a class with an access level of `private`, and then a function with an access level of `internal` (since it is not explicitly specified and the default access level is `internal`.)

The access level of the function `printSpecies` might be assumed to be `internal` (since its type is not specified and `internal` is the default), but note that it uses an input parameter with a type of `Trout`. And we have explicitly tagged the custom class `Trout` as having an access level of `private`. The level `public`, remember, is the least restrictive (it will always run), while `private` is the most restrictive. Because the input parameter for `printSpecies` is very restrictive (`private`), the access level for `printSpecies` will not be `internal` but `private`.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 27 exercises, go to

understandingswiftprogramming.com/27

28 Generic Programming

EnEnSwift, like many other modern languages, has capabilities that support *generic programming*. Generic programming allows a programmer to write code that works for more than one data type.

The Problem

The need for generic programming in Swift actually results from Swift's high degree of type safety. In every piece of code, we are very aware of the data type of variables, constants, and the like. However, forcing variables and constants to be a particular type makes it difficult to write code that works for more than one type. We can see the problem in the examples below.

Suppose we have two integers stored in variables:

```
var integer1 = 5  
var integer2 = 7
```

And we'd actually like to swap the values, so that `integer1` becomes 7 and `integer2` becomes 5.

We can write a function to do this:

```
func swapIntegerValues(inout a: Int, inout b: Int) {  
    var temporaryValue = a  
    a = b  
    b = temporaryValue  
}
```

And execute it with:

```
swapIntegerValues(&integer1, b: &integer2)  
print("integer1=\\"(integer1) and integer2=\\"(integer2)")  
// Prints: integer1=7 and integer2=5
```

The only thing a little unusual about this is that instead of passing the values to a function and getting one or more return values, we are passing them as “inout” values. The “`&`” in front of the variable names in the function call, and the `inout` keyword in the input parameter definition in the function, indicates that these are inout values, meaning that the global variables `integer1` and `integer2` are passed as parameters, accessed, and can even be

changed, within the function.

Now suppose we want to do the same thing with a pair of floating point numbers:

```
var floating1 = 5.1
var floating2 = 9.2
```

Our function for swapping integers won't work—it only works with the type of `Int`. We'd normally have to write another function:

```
func swapFloatingValues(inout a: Double, inout b: Double)
{
    var temporaryValue = a
    a = b
    b = temporaryValue
}

swapFloatingValues(&floating1, b: &floating2)
print("floating1=\\"(floating1) and floating2=\\"(floating2)")
// Prints: floating1=9.2 and floating2=5.1
```

This gets annoying, even for this simple function, if we want to use it with various of other types—say `Float` or `String`. For every type, we need a different function. If we had a complicated function with lots of different input values, each with several different types, it would get to be rather time-consuming to write functions to handle all the possibilities.

The Generic Solution

Generic programming allows us to write a single, generic function that has a generic type, represented with the symbol `T`:

```
func swapGenericValues<T> (inout a: T, inout b: T) {  
    var temporaryValue = a  
    a = b  
    b = temporaryValue  
}
```

It is used, and called, in exactly the same way as for the function that uses integer types only:

```
var integer1 = 5  
var integer2 = 7  
swapGenericValues(&integer1, b: &integer2)  
print("integer1=\\"(integer1) and integer2=\\"(integer2)")  
// Prints: integer1=7 and integer2=5
```

But we can now use it with other types as well.

To use generic programming to define a function, we make two changes. First, we add, just after the function name, one or more parameters contained within angle brackets. In this case, there is a single `T`, known as a *placeholder type*, that represents a type that is specified only generically. In this case, it can be any type. And then, wherever an actual type name is specified in the code in the function, we use the placeholder type, in this case `T`, instead. An upper case `T` is used by convention, especially when only one type is used, but any legitimate identifier name can be used. It is also possible to specify more than one type parameter, and in that case it is more common to use more meaningful names as identifiers. Camel case notation is typically used, with the first letter of the identifier in upper case. The placeholder types within angle brackets are known as *type parameters*.

The rest of this chapter describes some of the different ways that generic programming can be used in Swift. In the example above the generic type can stand in for *all* possible types. This is actually unusual. Normally, limits are placed on what types are allowable. These

limits are known as *constraints*, and they are specified by either indicating the names of protocols that the types must conform to, or the names of classes that the types must be subclasses of.

As we saw above, generic programming can be used with functions. But it can also be used with classes, structures, and enumerations.

Generic Functions and Constraints

We can look at a different function to see an example where we need constraints:

```
func compareTheseValues<T> (a: T, b: T) {  
    if a == b {  
        print("The two values are equal")  
    }  
    else {  
        print("The two values are NOT equal")  
    }  
}
```

We can call this with the following:

```
compareTheseValues(5, b:7)  
// Prints: error: cannot invoke '==' with an argument list of type '(T, T)
```

In fact, this function as described will not work. We get an error message, because the double equals sign comparison operator won't work with just *any* type. We have to constrain the allowable types:

```
func compareTheseValues<T: Equatable> (a: T, b: T) {  
    if a == b {  
        print("The two values are equal")  
    }  
    else {  
        print("The two values are NOT equal")  
    }  
}  
  
compareTheseValues(5, b: 7)  
// Prints: The two values are NOT equal
```

This version of the function works. In this case, we specified the placeholder type `T` with a colon after it and then the name of a protocol, `Equatable`. This specifies that the function will only work for types that conform to the `Equatable` protocol. This is our constraint. We could have alternatively specified the name of a class, which would mean that the function would only work for types that are subclasses of the class that was specified.

The `Equatable` protocol requires types that conform to it to implement two operators, the `==` operator and the `!=` operator.

Multiple Placeholder Types and Names

In the examples above we just used a single placeholder type, and called it T:

```
<T>  
<T: Equatable>
```

The use of T is arbitrary and is just a convention to use when you have a single placeholder type. If you have two, its common to use T and U. If there are more than two, it is common to use meaningful names instead of T and U.

```
<T, U>  
<TreeLength, TreeDiameter, TreeSpecies>
```

It is customary to use camel case, with the first letter in upper case.

Generic Classes and the Like

In some situations generic functions aren't enough—you want to use a class, structure, or enumeration. You might want to define a variable in a class, say, in generic terms, and you might want the variable to be accessible by an initializer and/or method and refer to it in generic terms in that initializer or method.

Generic types are defined in classes, structures, and enumerations in exactly the same way that they are for functions. Placeholder types for one or more generic types go between angle brackets just after the name of the class, structure, or enumeration. They can be referred to in variables, constants, initializers, or methods. The scope of the placeholder is the entire class, structure, or enumeration. So if you define a class, structure, or enumeration as having a particular placeholder, such as T, you can use the placeholder T within a method of the class, structure, or enumeration without specifically defining it for that method.

The following example shows a structure with a definition that uses a placeholder type:

```
struct Queue<T> {
    var queue = [T]()
    mutating func put(item: T) {
        queue.append(item)
    }
    mutating func get() -> T? {
        var n = queue.count
        if n > 0 {
            var itemFromQueue = queue[0]
            for var i = 0; i < n-1; i++ {
                queue[i] = queue[i+1]
            }
            queue.removeAtIndex(n-1)
        }
        return itemFromQueue
    }
    else {
        return nil // If no item
    }
}
```

```
    }  
}  
}
```

We can put a series of integers into the queue as follows:

```
var queue = Queue<Int>()  
queue.put(5)  
queue.put(8)  
queue.put(2)
```

And get it out as follows:

```
print(queue.get())  
// Prints: Optional(5)
```

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 28 exercises, go to

understandingswiftprogramming.com/28

29 Classes Revisited Again: Initializers

An earlier chapter (Chapter 12 on “Classes, Objects, and Inheritance”) described how an instance of a class, that is, an object, is created. When creating a new instance of a class, all variables and constants defined in that class must be assigned an initial value.

(An exception is optionals, which will be set to `nil` if not otherwise initialized. It’s still a good idea, however, to also set initial values to optionals, for good code readability.)

A property that has its initial value set when it is declared has a *default property value*. Apple recommends doing it this way if the value is always the same for any instantiation. This approach is generally simpler and cleaner.

Thus, if we have a simple class `Fruit`, the following will define the class, declare the property `skinColor`, and set an initial value for `skinColor`:

```
class Fruit {  
    var skinColor = "red"  
}
```

If we then create an instance of the class `Fruit`, the initial value will be set to “red”.

```
let pieceOfFruit = Fruit()  
print(pieceOfFruit.skinColor) // Prints: red
```

Initializers

In the examples shown in the earlier chapter variables and constants were usually initialized this way. In one case, we used a simple version of an *initializer*, a method-like piece of code in a class that begins with `init()` and is executed automatically as part of creating a new instance of a class. This looks like the following:

```
class Fruit {  
    var skinColor: String  
  
    init () {  
        self.skinColor = "red" // Using self is preferred but not required until there  
        // is ambiguity  
    }  
}
```

A new instance of the class `Fruit` is created, setting the initial value of the property `skinColor` to “red”. This does the same thing as the previous code that set the value in the declaration of the variable. The Swift compiler will enforce the requirement that an initial value be set, but it is just as happy to have it done by an initializer as in a property declaration.

However, when an initializer is used, the property declaration must have its type set by a type annotation. The compiler cannot infer the type if the value is set in an initializer.

In the above example the property `skinColor` is referred to by `self.skinColor`, with `self` referring to the current class. Just referring to `skinColor` itself, without the `self`, will work. Using `self`, however, is considered a better practice because there are situations in which there is ambiguity and using `self` is a good habit.

The initializer can appear anywhere in the class definition, but it is convenient to place it between the declarations of properties and the definitions of methods.

In this chapter we will consider initializers in much more detail. They can get quite complicated, especially when a class inherits from one or more superclasses, since inherited properties must also be initialized.

Designated Initializers

The initializer described above is known as a *designated initializer*. Unless initialization is only done by setting initial values in the same statement that declares a property, each class must have at least one designated initializer.

A designated initializer has the keyword `init` along with (optional) initial parameters and some code. It is not actually a method, but looks and acts very much like an instance method. There are some differences: The keyword `func` is not used, and initializers do not return a value. A designated initializer is called automatically when a new instance of a class is created. You cannot call a designated initializer in ordinary code, although you can call one from another designated initializer or from another type of initializer that we will discuss a bit later, known as a convenience initializer.

The initializers described in this chapter work only for classes. A similar process happens for structures, but does not work quite the same way. Enumerations are initialized in still a different way, with each enumeration member having its own initializer, or set of initializers, included in the case clause for that member. See the chapters on structures and enumerations for details.

We can use a designated initializer to allow the statement creating a new object to pass values that are set to properties of the class. This is shown below:

```
class Fruit {  
    var skinColor: String  
    var fleshColor = "white"  
  
    init(skinColor: String) {  
        self.skinColor = skinColor  
    }  
}
```

If a class is defined as above, a new instance of the class can be created with the following:

```
let aFruit = Fruit(skinColor: "green")
```

```
print(aFruit.skinColor) // Prints: "green"  
print(aFruit.fleshColor) // Prints: "white"
```

This is a little confusing because we are using the same name for the argument that we used for the property. However, this is the usual practice, and the use of `self.skinColor` allows us to reference the property, while `skinColor` itself refers to the argument passed in the initializer.

This sets the value of the property `skinColor` to the value indicated in the statement creating the new instance, by executing the initializer and passing it the `skinColor` parameter set to the value `green`.

We can also define a class that gives us the ability to either create a new instance with a default value for `skinColor` or to set it specifically by the statement creating the instance. The class definition is:

```
class Fruit {  
    var skinColor: String  
    init() {  
        self.skinColor = "red"  
    }  
    init(skinColor:String) {  
        self.skinColor = skinColor  
    }  
}
```

There are now two initializers included in the class definition. One has no input parameters, the other has a single input parameter labeled `skinColor` of type `String`.

If we want to just create a new instance and execute the designated initializer with no input arguments, we do the following:

```
var aFruit = Fruit()
```

This will create an instance of the class `Fruit` and set its `skinColor` property to "red".

Alternatively, we can create a new instance and execute the designated initializer that allows us to pass a “green” value for the parameter `skinColor`:

```
var aFruit = Fruit(skinColor: "green")
```

We did this earlier; the only difference is that our class definition now has both kinds of initializers in it, and the system had to determine which initializer to execute. This of course uses what is effectively function overloading, matching the type(s) of input parameters of the initializer to allow calling the correct one. (This might be called “initializer overloading.”)

A more realistic `Fruit` class, as we saw in earlier chapters, has more properties. An example is this:

```
class Fruit {  
    var countryGrownIn: String  
    var skinColor: String  
    var fleshColor: String  
    var hasSeeds: Bool?  
    var weightLbs: Double  
  
    init (countryGrownIn: String, skinColor: String, fleshColor: String, hasSeeds: Bool, weightLbs: Double) {  
        self.countryGrownIn = "United States"  
        self.skinColor = skinColor  
        self.fleshColor = fleshColor  
        self.hasSeeds = hasSeeds  
        self.weightLbs = weightLbs  
    }  
}
```

To create an instance of `Fruit`:

```
aFruit = Fruit(countryGrownIn: "United States", skinColor: "green", fleshColor: "white", hasSeeds: true, weightLbs: 0.6)
```

While this gives us a lot of control over the setting of the properties of an instance, we usually don't need such a detailed level of control and would often prefer something easier. That's where convenience initializers are for.

Convenience Initializers

A *convenience initializer* simplifies code by defining default values for many or most of the properties of the class before calling a designated initializer. A convenience initializer is optional, and it must call another initializer (convenience or designated) in the same class. Eventually, a designated initializer in that class must be executed.

Initializers in classes cannot be called by ordinary code. However, convenience initializers can call designated initializers, and, as we will see later, designated initializers can call designated initializers in superclasses.

Convenience initializers, like designated initializers, are never called directly except by other initializers, and use the keyword `init`. A convenience initializer has same syntax as a designated initializer, except that it has the keyword `convenience` just before the `init` keyword. If there is more than one convenience initializer defined in the class, a particular convenience initializer is called as a result of matching the parameter names and types in the statement creating an instance of a class.

The class `Fruit` defined below shows how a convenience initializer is used. This is like the class `Fruit` we saw before, except that it has five different properties rather than just one:

```
class Fruit {  
    var countryGrownIn:String  
    var skinColor:String  
    var fleshColor:String  
    var hasSeeds:Bool  
    var weightLbs: Double  
  
    init(countryGrownIn: String, skinColor: String,  
        fleshColor: String, hasSeeds: Bool, weightLbs: Double) {  
        self.countryGrownIn = countryGrownIn  
        self.skinColor = skinColor  
        self.fleshColor = fleshColor  
        self.hasSeeds = hasSeeds  
        self.weightLbs = weightLbs  
    }  
}
```

```
}

convenience init(skinColor: String, weightLbs: Double) {
    self.init(countryGrownIn: "United States",
skinColor: skinColor, fleshColor: "white", hasSeeds: true,
weightLbs: weightLbs)
}

}
```

Let's say that for our particular application, most of the time three of the five properties defined in the class are normally the same: "United States" for countryGrownIn, "white" for fleshColor, and true for hasSeeds. The properties that are typically different for different pieces of fruit are skinColor (which might be "red", "green", "yellow", and others) and weightLbs, the weight in pounds. We create a designated initializer that initializes all of the properties in the class. But most of the time, when we create a new instance, we won't define the values of all five properties in our statement that creates the new object. Instead, we'll create a new instance with just two parameters, as shown below:

```
var aPieceOfFruit = Fruit(skinColor: "green", weightLbs:
0.54)
```

This creates a new instance but does not call the designated initializer because the parameter names and types do not match. Instead, the initializer that matches the parameter names and types is called, which turns out to be the convenience initializer. That convenience initializer, in turn, calls the designated initializer. In doing so, it passes along the values for skinColor ("green") and weightLbs (0.54), and then specifies default values for the other three properties: countryGrownIn, fleshColor and hasSeeds.

The mixture of using a designated initializer and a convenience initializer allows the programmer full control to in setting the properties of a new instance when creating it when desired, but makes it easy to create a new instance in the more common cases when only one or two parameters need to be set to something other than a default value.

Some General Rules About Initializers

One exception to the rule that variables must be provided with an initial value is in the case of optional types. Optionals may be initialized, but they are not required to be. If they are not initialized they will be assigned a value of `nil`, that is, no value.

Changes to properties made by initializers do not trigger the execution of property observers.

Constants, which normally can be assigned a value only once, can be changed by initializers, even if they have already been assigned a default value.

Initialization and Inheritance

When a class is part of an inheritance hierarchy, Swift has mechanisms and rules that insure that all properties that are defined in that inheritance hierarchy are set to initial values.

In the case of a class that inherits from one or more superclasses, it may be necessary for that class to provide an initializer that makes a call to the initializer of its immediate superclass, and perhaps to modify the values of properties that it has inherited as well as set the initial values of the properties that it is defined in its own class.

The programmer has a choice here. If the stored properties that are defined in a new subclass are set to initial values as part of their declarations, and if the subclass has no initializers, then the subclass will inherit initializers from its superclass(es) and the programmer does not have to do anything further.

The other alternative, the more common case when the new subclass has one or more initializers, requires that a call be made to an initializer in the subclass's immediate superclass. If there are further superclasses, this will trigger a chain of calls to initializers that eventually reaches the highest level in the hierarchy, the base class.

This is done in two phases, to insure that a class in the hierarchy does not improperly overwrite properties that were initialized by another class.

The phases are:

Phase One. In this phase, successive calls are made to initializers in every class in the hierarchy that defines one or more stored properties. This begins at the level of the new subclass, which initializes its stored properties and then makes a call to an initializer of its immediate superclass. That initializer will call an initializer of its immediate superclass, continuing until it reaches the topmost level. A designated initializer in every class in the hierarchy must be executed. (Convenience initializers can be called, but they must, directly or indirectly, execute a designated initializer in the same class.) The initializer must initialize all properties that have been introduced by its class (that have not been initialized in their declaration) before calling the superclass.

Phase Two. In this phase, the initializer for the new class can modify the values of any inherited properties, which already have had initial values set by an initializer in a superclass.

What this means for a designated initializer in a new subclass is that it has code that does three things, in the following order:

First, it sets initial values for all properties that do not have initial values set as part of their declaration. (Optionals do not have to be set since they will be set to `nil` automatically.)

Second, it makes a call to an initializer in its immediate superclass.

Third, if desired, it modifies the values of properties that have been inherited.

If you do not do this in the order indicated, the compiler will complain.

We can see an example of this with a pair of classes, `Fruit` and `Banana`:

```
class Fruit {  
    var skinColor:String  
    var countryGrownIn: String  
    init() {  
        self.skinColor = "green"  
        self.countryGrownIn = "United States"  
    }  
}  
  
class Banana: Fruit {  
    var weightInLbs: Double  
    init(weightInLbs: Double) {  
        self.weightInLbs = weightInLbs  
        super.init()  
        self.skinColor = "yellow"  
        print(weightInLbs)          // Prints: 0.42  
        print(self.skinColor)       // Prints: yellow
```

```
print(self.countryGrownIn)    // Prints: United States
}
}

let aBanana = Banana(weightInLbs: 0.42)
```

We create an instance of the class `Banana`, providing the weight of our particular banana, 0.42 pounds. This causes execution of the designated initializer for the `Banana` class and passes it the value of 0.42.

The initializer follows the defined sequence. First, it initializes the one property that was newly defined in the `Banana` class: `weightInLbs`. Next, it makes a call to its superclass, `Fruit`. The superclass executes its initializer, setting the default values to its properties `skinColor` and `countryGrownIn`.

If there was an additional superclass in the hierarchy, its initializer would be called here, but since there is not, the superclass initializer is done.

The instance of the `Banana` class has inherited two properties from `Fruit`: `skinColor` and `countryGrownIn`. The `Banana` class initializes `skinColor` to “yellow” (changing its value from the value set by the superclass). It does not change the value of the other inherited property, `countryGrownIn`.

We can see the values that get printed by the `Banana` initializer when it is done. The class has introduced a single new property, `weightInLbs`, and set it as part of the call creating the new instance. It has inherited a property, `skinColor`, and changed its value. And it has inherited another property, `countryGrownIn`, and kept its value.

Overriding Initializers. It is possible to override an inherited initializer. In the case of a designated initializer, the keyword `init` is preceded by the keyword `override`. In the case of a convenience initializer, it simply uses the normal syntax, without the `override` keyword. A convenience initializer that is overriding an inherited initializer must use the same names and types that were used in the initializer it is overriding.

Required Initializers. If an initializer is marked with the keyword `required`, it will be inherited in any subclasses. It is relatively rare for initializers to be inherited, and this keyword ensures that particular initializers will in fact be inherited.

Deinitialization and Inheritance

Deinitializers are inherited, if they exist, by subclasses. When an object is being destroyed, a deinitializer in its class will be called first if it exists. Then any existing deinitializers from superclasses will be called.

Failable Initializers

It is possible for an initializer to fail in certain circumstances. In cases where an initializer might fail, there must be code that determines whether the attempt to create the new instance has succeeded or failed and that takes appropriate action.

Such “failable initializers” can be used in classes, structures, and enumerations.

If an initializer can fail, it should contain an initializer using the `init` keyword that contains code to deal with the failure. The keyword `init` in such a situation should be followed by a question mark.

Why would an attempt to create an instance fail? There are several possibilities.

Some instances require certain resources that, at a given time, might not be available. For example, an instance might be an image (e.g., a Cocoa Touch `UIImage` type) that must be read from the device’s file system, or downloaded from a remote server. In either case, that image might not be available, for whatever reason.

Other instances might be created using parameters supplied by the statement that calls for the instance to be created. It may be desirable to check these values to see if they are valid. If one or more is not valid, the instance cannot be created, and the initializer must fail.

The following class definition defines a class named `Apple` that allows instances to be created that sets the values of certain properties when it is created:

```
class Apple {  
    var skinColor: String = "red"  
    var weightInLbs: Double = 0.5  
    init?(skinColor: String, weightInLbs: Double) {  
        guard (skinColor == "red" || skinColor == "yellow" ||  
            skinColor == "green") else { return nil }  
        self.skinColor = skinColor  
        guard weightInLbs <= 3.0 else { return nil }  
        self.weightInLbs = weightInLbs
```

```
}
```

```
}
```

An instance of Apple is created as follows:

```
let anApple = Apple(skinColor: "red", weightInLbs: 0.8)
```

This will successfully create an instance of the class `Apple` that has a skin color of “red” and a weight (in pounds) of 0.8.

However, the initializer has some validity checks when creating an instance. Only apples with a skin color of “red”, “yellow”, or ‘green’ are allowed to be created. In addition, the weight of the apple must be no more than 3 pounds.

If we attempt to create a purple apple, it will fail:

```
let anApple = Apple(skinColor: "purple", weightInLbs: 0.8)
print(anApple)      // prints: nil
```

As part of the validity check in the initializer code, if the validity check fails the statement `return nil` is executed. This causes an exit from the initializer. Note that initializers do not actually return a value, and so this statement of `return nil` is not actually an attempt to return a value: it is just a signal to indicate that the initializer has failed. However, the constant `anApple` will be set to `nil` in this case. In this very specific situation, the initializer effectively “returns” a value. (If the initializer succeeds, no `return` statement is executed.)

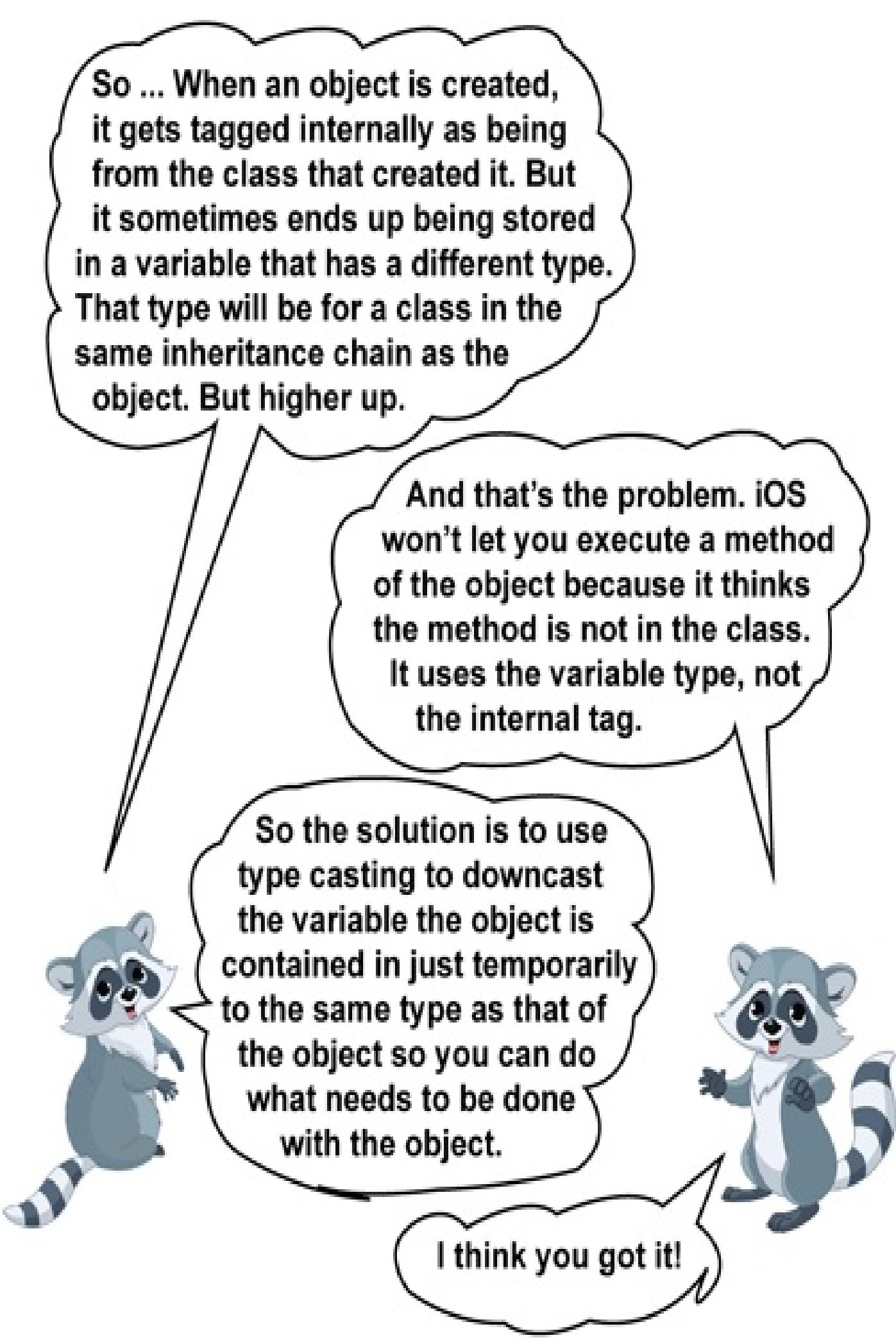
Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 29 exercises, go to

understandingswiftprogramming.com/29

30 Type Checking and Type Casting



So ... When an object is created, it gets tagged internally as being from the class that created it. But it sometimes ends up being stored in a variable that has a different type. That type will be for a class in the same inheritance chain as the object. But higher up.

And that's the problem. iOS won't let you execute a method of the object because it thinks the method is not in the class. It uses the variable type, not the internal tag.

So the solution is to use type casting to downcast the variable the object is contained in just temporarily to the same type as that of the object so you can do what needs to be done with the object.

I think you got it!

Swift provides capabilities for both determining the type of a value and for causing the system to temporarily interpret the type of a variable or constant as that of another type. These capabilities are known as *type checking* and *type casting*.

Why Type Checking and Type Casting are Needed

Suppose we have the following classes:

```
class Animal {  
}  
  
class Fish: Animal {  
    func printFish() {  
        print("I am a fish")  
    }  
}  
  
class Bird: Animal {  
    func printBird() {  
        print("I am a bird")  
    }  
}
```

We can create an instance of each of the Fish and Bird classes:

```
var aBird = Bird()  
var aFish = Fish()
```

And we can also create an array called `twoAnimals` and store both of these instances in the array:

```
var twoAnimals = [aBird, aFish]
```

What we have are two instances, one of the class `Bird` and the other of the class `Fish`.

Although arrays must contain values of the “same type”, Swift’s adherence to the subtyping (subtype polymorphism) principle, plus a very accommodating compiler, results

in a rather liberal interpretation of this rule.

The instances of `aBird` and `aFish` are not the same type (`Bird` and `Fish`). But what the compiler will do is to tag the array `twoAnimals` with the type `Animal`. Types of `Bird` and `Fish` are allowed to be contained in an array of type `Animal`, because both are subclasses of `Animal`. (This is the subtyping principle.)

Now, suppose we want to look in detail at each of the instances. We might assign one of them to the variable `obj`:

```
var obj = twoAnimals[0]
```

It's important to recognize just what we are dealing with here. Although the array `twoAnimals` has a type of `Animal`, the actual objects within the array still have their original type. (This is the type of the class that created these as instances of that class.) When we do the assign to the variable `obj`, that variable will be inferred to be a type of `Animal`. But, internally, the object in `obj` will still have its original type of `Bird`.

Type Checking with `is`

We can test this with the type checking keyword `is`:

```
var obj = twoAnimals[0] // Get the bird instance
print(obj is Bird) // Prints: true
print(obj is Fish) // Prints: false
```

The type checking keyword will check the internal type associated with the instance, not the type associated with the variable, which is `Animal`.

When Internal Types and the Types of Variables Containing Them are Different

The variable `obj` contains an instance of the class `Bird`, and that class has a method called `printBird`. We want to execute that method. We could try the following:

```
obj.printBird  
// Error: 'Animal' does not have a member named 'printBird()'
```

This gets an error. What's the problem? It is that the variable `obj` has a type of `Animal`. And the class `Animal` does not have a method named `printBird`. It's not enough for the internal representation of the instance to have the correct class name. The variable that it is contained in must also have the correct class name.

Type Casting with as, as? and as!

The solution to this is one of the so-called type casting operators, as, as? and as!

I say “so-called” because these operators do not change the type of a variable in the same way that they do in many other languages.

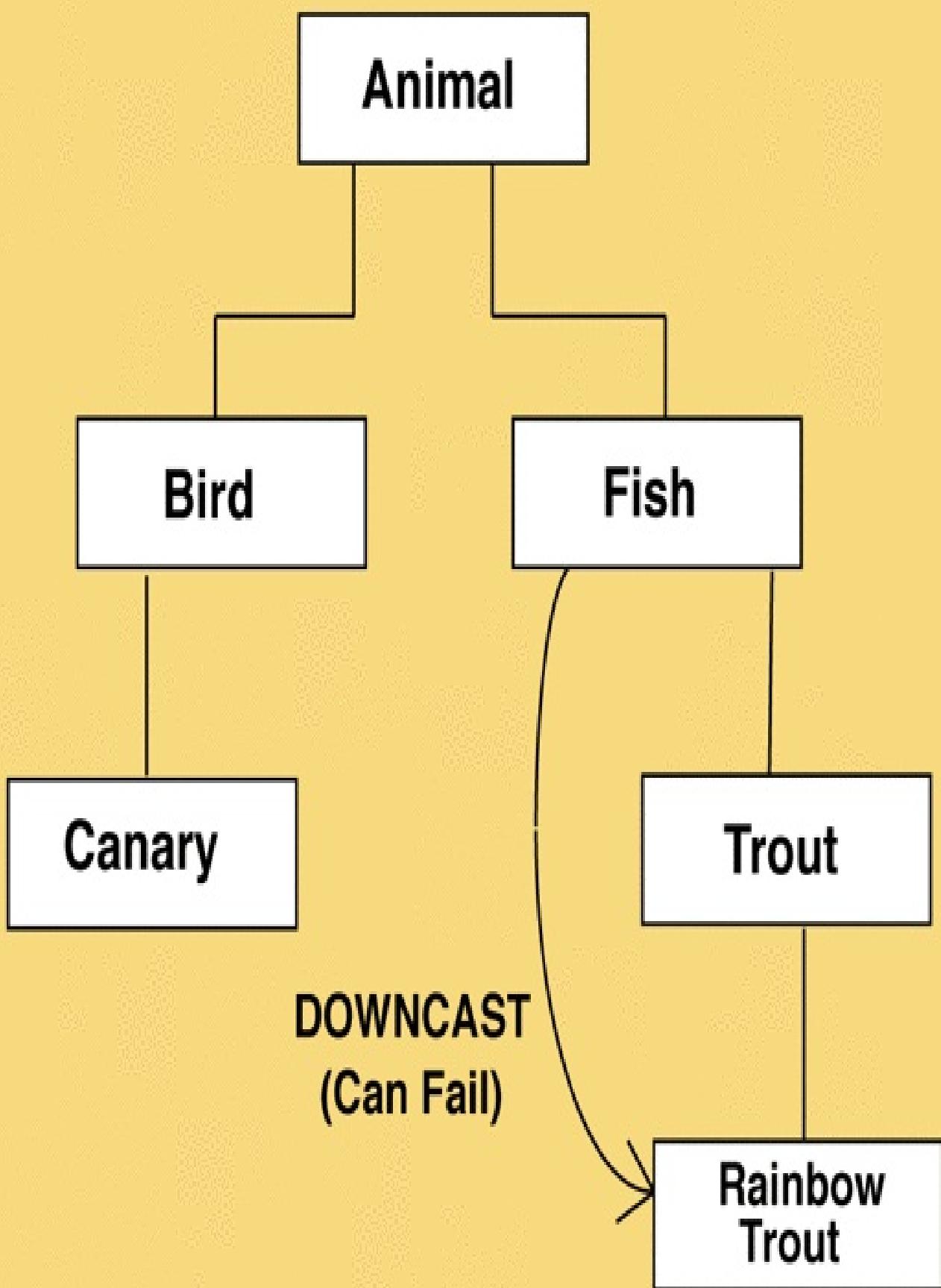
In Swift type casting is more tentative. It’s as if the keyword `as!` means “Please accept this as if it was this particular type.”

Here `obj` contains, as before, an instance of the `Bird` class. But we have, with the `as!` keyword, effectively (but temporarily) made the system accept `obj` as being of type `Bird`. This allows the `printBird` method to be executed.

Casting a variable with a type like `Animal` to be a type like `Bird` (a subclass of `Animal`) is known as *downcasting*. (See the section later on “What’s Up and What’s Down?”)

It is possible for downcasting to fail. If you try to downcast a variable type to a subclass that does not exist, the downcast will fail. The exclamation point in the `as!` keyword is intended to remind programmers of this fact. If you try to downcast to a class that does not exist, you will get a runtime error.

If there is any possibility that the subclass you want to downcast to might not exist, it is safer to use `as?` rather than `as!` This keyword will cause the operation to return a `nil` if the downcast fails, which you can then check for.



What's Up and What's Down?

In the bizarre world of computer science, a “tree” has only a single root and is upside-down, with the root at the top and branches pointing down and leaves at the bottom. Object oriented programming uses the tree metaphor as its basis for understanding classes and subclass relationships. A base class is thus at the top, with its subclasses being branches and eventually leaves below. Given this, a “downcast” means converting an instance of a given class to have (temporarily) have the type of one of its subclasses. (An “upcast”, the reverse of this, is possible in Swift but is not normally done.)

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 30 exercises, go to
understandingswiftprogramming.com/30

31 Enumerations Revisited

1.

I think I understand raw values now. But associated values seem even stranger.

2.

The barCode enumeration shown on the left whiteboard has a single String "property" for the oldStyleCode member, and four integer "properties" for the QR code member.

Associated values are like properties. Except they are not for the whole class, just for specific members.

```
enum barCode {  
    case.oldStyleCode(String)  
    case qrCodes(Int, Int, Int, Int)  
}
```



Optionals are Defined by an Enumeration

```
enum (A) {  
    case Some (A)  
    case None  
}
```



Type is stored as a associated value

We saw in an earlier chapter how an enumeration can be created—a new named data type that can contain only one of a set of predefined values that are specified in the definition of that enumeration. Thus, we can create a custom enumeration named `DayOfTheWeek` and define it as having possible values (known as *members*) like `Monday`, `Tuesday`, `Wednesday`, etc. that represent all of the days of the week. We can then define a variable as:

```
var today: DayOfTheWeek = .Friday
```

We also saw that other values, such as integers, can be attached to each of the enumeration members. Thus the member named `Friday` might have as an alternative value the integer 5. These are known as *raw values*.

This is a nice but not very surprising extension of the way that enumerations normally work in C and similar languages.

Clearing Up Some Confusion About Instances, Properties, and Enumerations

We now get to a very different role of an enumeration—its role as a custom type that works much like a class and that has instances.

We can see graphically how enumerations fit in with classes and structures by looking at the table below that shows the different characteristics of classes, structures, and enumerations:

	Classes	Structures	Enumerations
Memory Type	Reference	Value	Value
Has instances	✓	✓	✓
Inheritance	✓		
Type casting	✓		
Conform to protocol	✓	✓	✓
Can be extended	✓	✓	✓
Deinitializer	✓		
Instance methods	✓	✓	✓
Class (type) methods	✓	✓	✓
Stored instance properties	✓	✓	
Stored class (type) properties		✓	✓
Computed properties	✓	✓	✓
Lazy properties	✓	✓	✓
Property observers	✓	✓	✓
Respond to notifications	✓		
Custom subscripts	✓	✓	✓
Memberwise initializer		✓	

The big things are inheritance (type casting goes along with it), using reference type memory (deinitializer goes along with this), and having stored instance properties.

Structures and enumerations don't have inheritance nor use reference type memory, and are in this way quite different than classes. But a structure is in a sense a simplified class, and uses instances in the same way as classes, which is a major similarity.

Enumerations don't have stored properties, which makes them seemingly very different from structures and classes.

What might be surprising, though, is that enumerations, like the other types, can create instances, even though they do not have stored properties. And although they do not have stored properties, they do have something else that acts very much like a stored property.

A stored property is Swift's version of what in object-oriented programming has traditionally been called an *instance variable*. An instance variable stores information for a particular instance, and the data in the set of instance variables defined by the class is what makes one instance different from another. So how can enumerations have instances if they have no stored properties?

Or, even if they can create instances, how can these different instances be useful if they don't have values in stored properties to differentiate one from another?

Associated Values are the Instance Variables/Stored Properties of Enumerations

The answer to this is that enumerations have something new, known as an *associated value*, that plays the role of an instance variable or Swift property for an enumeration.

Associated values work rather like Swift properties but each is defined in a way that makes it useful for specific enumeration members.

Let's take the example that we used before, of an enumeration that represents a particular day of the week.

Now suppose that we want to capture some information, but the information we want to capture depends on the day of the week.

On weekdays we catch a commuter train and we are interested in tracking how accurately the trains arrive at the Embarcadero station in San Francisco (we're concerned about being late for work). So we have an app that, when we tap a button that says we have arrived, captures the time.

However, on Saturdays and Sundays, we walk around the park in our suburban city, where we are fascinated by our suspicion that every time a duck dives to the bottom of Lake Elizabeth, it comes back up after a delay that seems quite constant for each duck, even though different ducks take different amounts of time. (Presumably the ducks are digging for food.)

So on Mondays, for example, we capture a 6-character string that represents the hour, minute, and second of our arrival in San Francisco (e.g., "084501").

On Saturdays and Sundays, however, we capture a different kind of information. For each instance of a duck sighting, we capture the type of duck (a string like "Mallard, male") and then we capture a floating point number which represents the amount of time in seconds that the duck was under water. The ducks always come up within 60 seconds, but we want much more precision than we cared about in the case of the commuter train.

The definition of the enumeration would be:

```
enum DayOfTheWeek {  
    case Sunday (String, Double)
```

```
case Monday (String)
case Tuesday (String)
case Wednesday (String)
case Thursday (String)
case Friday (String)
case Saturday (String, Double)
}
```

If the day is Monday, we might create an instance representing our trip into San Francisco with the following:

```
var today: DayOfTheWeek = .Monday("084501")
```

This instance captures the value of the enumeration, plus the associated value, that saves information as if it were a single stored property.

If the day is Saturday, we might create an instance representing a single observation of a duck diving in the lake that captures the type of duck and how long the duck was underwater:

```
var today: DayOfTheWeek = .Saturday("Mallard, Female",
8.4327)
```

Here we capture two pieces of data, and store them into what are effectively two different stored properties, although we don't call them that, and they only work with instances set to enumeration values that have the proper associated values.

On weekdays we just create one instance per day. On weekends we create as many instances as we have observations of ducks diving.

If we have an instance, we can get the information from it by using a switch statement:

```
var today: DayOfTheWeek = .Saturday(("Mallard, Female",
8.4327)

switch today {
```

```
case let .Sunday (whichDuck, timeUnderwater) :  
print("Duck \u2028(whichDuck) took \u2028(timeUnderwater) secs");  
case let .Monday (timeArriving) :  
print("Train arrived Mon at \u2028(timeArriving)")  
case let .Tuesday (timeArriving) :  
print("Train arrived Tues at \u2028(timeArriving)")  
case let .Wednesday (timeArriving) :  
print("Train arrived Wed at \u2028(timeArriving)")  
case let .Thursday (timeArriving) :  
print("Train arrived Thurs at \u2028(timeArriving)")  
case let .Friday (timeArriving) :  
print("Train arrived Fri at \u2028(timeArriving)")  
case let .Saturday (whichDuck, timeUnderwater) :  
print("Duck \u2028(whichDuck) took \u2028(timeUnderwater) secs")  
}
```

Examples of the Use of Associated Values

It may be helpful in visualizing how associated values are used to know about other examples of how they have been used, including those in the real world.

ENUMERATIONS FOR DEFINING OPTIONALS

An enumeration, including an associated value, is used to implement the optional type in Swift. The definition of an optional is as follows:

```
Enum Optional {  
    case Some(String)  
    case None  
}
```

An optional has a value of either `.None` or `.Some`. If the value is `.None`, there is no associated value, and the optional is effectively a `nil`. If the value is `.Some`, the wrapped optional value is stored in the associated value for `.Some` and can be retrieved as part of the unwrapping process.

For example, suppose we create an optional string type and assign a value to it, causing the value to be wrapped:

```
var message: String? = "Syntax error"
```

We can later unwrap it with a switch statement:

```
switch message {  
    case .Some(let stringValue):  
        print("The unwrapped value is \(stringValue)")  
        // Prints "The unwrapped value is Syntax error"  
    case .None:  
        print("Variable not unwrapped, it is a nil")  
}
```

ENUMERATIONS FOR NETWORK QR CODES

The Apple documentation on enumerations uses as an example the collection of bar code numbers using two different kinds of bar codes—the older UPC-A format, and the newer two-dimensional QR code style format. The enumeration definition looks like this:

```
enum Barcode {  
    case UPCA(Int, Int, Int, Int)  
    case QRCode(String)  
}
```

If the bar code being read is in the UPCA format, it is captured in four pieces as integers. If the bar code is in the QR Code format, it is captured as a single long string.

This is like having four different integer stored properties associated with the UPCA member, but only a single string stored property associated with the QR Code member.

ENUMERATIONS FOR NETWORK ADDRESSES

One book on Swift (*Swift Pocket Reference*, by Anthony Gray), describes the use of enumerations to store network addresses. One type of network address, an Ethernet address, is stored in MAC form as a string (it is actually six 2-digit hex values separated by colons), while another type, IPv4, is stored as four 8-bit unsigned values. The enumeration definition looks like this:

```
enum NetworkAddress {  
    case MAC(String)  
    case IPv4(UInt8, UInt8, UInt8, UInt8)  
}
```

An instance of an address in the Ethernet form might be:

```
var ethernetAddress =  
NetworkAddress.MAC("00:DD:AA:BE:EA:00")
```

An instance of an address in the IPv4 form might be:

```
var ipAddress = NetworkAddress.ipv4(192, 168, 0, 1)
```

ENUMERATIONS FOR CONNECTION STATUS

Another Swift book (*Swift for the Really Impatient*, by Matt Henderson and Dave Wood), describes an enumeration to store information related to a wireless device's current connection to a network. The enumeration definition is:

```
enum NetworkConnection {  
    case NotConnected  
    case WiFi(String, Int)  
    case Cellular(Int)  
}
```

The enumeration members are `NotConnected`, `WiFi`, or `Cellular`. In the latter two cases, the delay in accessing a server via the network is recorded, as an integer, in seconds. In the case of `WiFi`, the name of the `WiFi` network is also recorded.

Value Versus Reference Types in Memory

Like structures, but unlike classes, enumerations are value types, meaning that when a reference is made to them a copy is made of the information stored in memory related to the enumeration. Classes, in contrast, are reference types, meaning that they store only a single copy of the data associated with each instance of a class in memory. This goes into the heap memory (See Chapter 19 on “Memory Management” for a discussion of the differences between heap and stack memory.)

Choosing Between an Enumeration, a Structure and a Class

You would only choose to use an enumeration when the data you are dealing with fits naturally into the categories that you will use as members of the enumeration.

The decision about whether to use an enumeration in its role as a type that has instances, with associated values and methods, is a different decision than whether to use an enumeration in its more basic role. It's certainly possible to define an enumeration and then define a class or structure that uses the enumeration's values (perhaps defining a property for the value of the enumeration) and then using properties of a class or structure rather than associated values of an enumeration.

The decision is usually based mostly on the convenience of the data structures, but if one or more of the associated values has a value (e.g., a very large image or a complex type) that is expensive or unwieldy to copy, it may be desirable to use a class even if the data structure is a little more cumbersome.

Using Methods With Enumerations

Instance and also “class” or “type” methods can be used with enumerations. Class/type methods can be used with class/type stored properties as these kinds of properties do exist for enumerations. However, instance methods must work with associated values rather than stored properties (since enumerations don’t have stored instance properties).

Thus if we create an instance of one of our DayOfTheWeek enumerations that resulted from a particular day Saturday and an observation of a particular duck:

```
var particularDay = DayOfTheWeek.Saturday("Mallard", 12.4)
```

Given an instance of DayOfTheWeek, we would extract the desired information from it using a switch statement:

```
switch particularDay {  
    case .Monday(let timeArrived):  
        print("Time arrived:\(timeArrived)")  
    case .Saturday(let typeOfDuck, let timeUnderwater):  
        print("Type of duck:\(typeOfDuck) Time underwater:\(timeUnderwater)")  
}
```

This obviously just covers Monday and Saturday, but you get the idea.

This extracts the values associated with the enumeration that matches and sets them to one or more constants (e.g., typeOfDuck and timeUnderwater in the case of .Saturday.) That constant can then be accessed in a print statement. (This uses a value binding in a switch statement, which is described in Chapter 20, “The Flow of Control Revisited”.)

Computed and Lazy Properties and Property Observers With Enumerations

Enumerations can have computed properties, lazy properties, and property observers to the extent that they make sense given their lack of stored properties. These have the same syntax as those described for classes.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 31 exercises, go to

understandingswiftprogramming.com/31

32 Protocols



Silicon Valley



BREAKING NEWS

SecState gaffe leads to nuclear Armegaddon

Putin: “If Ukraine can’t be part
of Russia, it should not exist.”

LIVE



**“Wolf, this whole thing could have
been prevented if the Secretary of
State was as good at observing
diplomatic protocol as his teenage
daughter is at making sure that her
Swift code conforms to its protocols.”**

A *protocol* is essentially a set of rules to be used in a particular situation. The term is often used in diplomacy and in medicine. A news report in October, 2014 stated that the first Ebola virus infection in the United States was caused by a “breach in safety protocol.”

Some example text from one of the Ebola protocols: (just skim this quickly):

Ensure that a trained observer watches closely each donning and each doffing procedure of personal protective equipment, and provides supervisory assurance that donning and doffing protocols are followed.

The removal of used personal protective equipment is a high-risk process that requires a structured procedure, a trained observer, and a designated area for removal to ensure protection.

Personal protective equipment must be removed slowly and deliberately in the correct sequence to reduce the possibility of self-contamination or other exposure to Ebola virus. CDC recommends facilities use a powered air-purifying respirator.

A powered air-purifying respirator should have a full face shield, helmet, or headpiece. Any reusable helmet or headpiece must be covered with a single-use (disposable) hood that extends to the shoulders and fully covers the neck and is compatible with the selected powered air-purifying respirator.

This is just a tiny piece of a list of rules that is many pages long. What the protocols say is “If you are going to treat an Ebola patient, here is how you must do it.”

The protocols for protecting against Ebola are very detailed and rigid—they have to be to avoid the spread of a deadly disease.

Diplomatic protocol is detailed and rigid so as to avoid initiating an embarrassing “incident”, or even starting a war.

In programming, the concept is mainly used to allow groups of programmers to more easily use and reuse each other’s code. If a programmer is building an API for iOS, he or she can define a protocol that says to the app programmers, “If you want to use this API,

here is how you must do it.” Typically this means “You must have methods with these names and input parameter types and return values that have these types” or “Your class must have properties with these names and types.”

You might also use protocols in your own code even if it just for your own use, if you want to build frameworks or just standard methods that you can use in multiple apps so as to reuse code more effectively and make it easier to maintain.

Some developers claim that protocols are a sort of poor man’s version of inheritance (usually offered up in discussions of Swift having only single inheritance, with the assertion that protocols are a form of multiple inheritance.) Until Swift 2, this was stretching it, because protocols didn’t actually cause methods and properties to be included in code. They only required programmers to put in something that fits certain syntactical requirements. And you can’t create an inheritance chain with protocols.

With Swift 2, classes, structures, and enumerations can obtain default implementations of methods and properties from protocols, and they can do so from more than one protocol at the same time. So protocols do start looking a bit like multiple inheritance.

Protocols have traditionally primarily promoted reuse of code by enforcing discipline. They can require that certain methods that have particular signatures be in the code. Such methods can be those that are outside of the existing inheritance chain for a class, or used in structures and enumerations (that don’t have inheritance).

The concept of a protocol is similar to that of an interface, and in fact the Java programming language uses the term *interface* to define the rules of their version of a protocol. A Java programmer will then create a class that *implements* the interface, taking care to provide an implementation for those methods and other aspects of code that are specified as required by the interface. A Swift protocol does much the same thing.

It is often said that a protocol (or Java interface) forms a *contract* or agreement between the different pieces of code on how to work together.

In most cases while building an app, the protocols that you will be using are part of the iOS API.

Conforming to and Using a Protocol

Suppose you want to use table views in your app, which are a sort of elaborate vertical menu that allows you to scroll through information and select a menu item.

Table views require the app to conform to two protocols to use them, known as the `UITableViewDataSource` and `UITableViewDelegate` protocols. (If you Google, say, `UITableViewDataSource` Protocol Reference, you'll get a full description of all of the methods that are included in this protocol).

Among the methods in the `UITableViewDataSource` protocol that is required is a method known as `numberOfRowsInSection`.

The menu items in table views are known as *rows*, and if you like you can divide the rows into different groups, known as *sections*. Most table views just have one section, and the example here will have just one section.

When you use a table view, you will create a subclass of `UITableViewController`.

```
class Fruit: UITableViewController, UITableViewDataSource,  
UITableViewDelegate {  
  
    // Methods in class  
  
}
```

When a class is defined, you must specify what protocols that the class will *conform to*. This means that the class agrees to follow the rules specified in the protocol. If the protocol requires a certain method with a certain signature, the class must implement it. If the protocol specifies as *optional* a certain method with a certain signature, the class may implement it but is not required to. (Optional methods aren't quite what they might seem: The iOS API has many APIs with methods that are marked as optional, but are de facto required, because the API wouldn't actually work or be useful without them.) (Just to avoid confusion: Optional methods in protocols have *nothing* to do with Swift optional values.)

In the class definition, you specify the protocols you agree to conform to by listing them after a colon following the name of the class you have defined, in the same way that you would list the name of a superclass. This may seem confusing, but the compiler knows which names represent classes and which represent protocols. The one requirement here is

that if there is a superclass, its name must come first.

If you specify a particular protocol in your class definition, you must include any methods and properties required by that protocol, or you will get an error message from the Xcode development environment.

So let's say that you want to have a table view that lists different kinds of fruits, along with their photo. And, pretty obviously, one of the things that you need to tell the API that will be creating your table view is how many menu items (fruits), that is, rows, will be in your table view.

The `UITableViewDataSource` protocol requires that you implement a method known as `numberOfRowsInSection`. In the typical menu, there is only one section, known as section 0, and you'll implement a method to tell the API that (not described here).

Then you will implement a method to tell the API how many rows need to be in the table. It will look something like this:

```
override func tableView (tableView: UITableView,  
numberOfRowsInSection section: Int) -> Int {  
    return 12  
}
```

This method is called by the operating system, once the app code has requested a table view. This aspect of a method required by a protocol for an API is a little odd, because it seems backwards. Usually, your program calls a method. Here, the operating system is calling a method, and your job is to implement that method.

When the operating system calls the method you are implementing, `tableView:numberOfRowsInSection:section`, it will pass some parameters, including an integer labeled `section`. Since this table view has only one section, this parameter can be ignored. The only thing the method needs to do is to return an integer that reflects the number of rows needed for the table view, which for simplicity I have fixed to be 12, assuming that there are 12 different fruits and thus 12 rows.

(This is an oversimplification; the actual number you will return will be computed in a real situation according to what you need.)

Declaring A Protocol

You can declare your own protocol using the `protocol` keyword. For example, a protocol that requires the method `getWeightOfPlant:` might be defined as follows:

```
protocol PlantProtocol {  
    func getWeightOfPlant(aPlant: Plant) -> Double  
}
```

The definition of the protocol includes the signature for any required functions—the keyword `func`, the name `getWeightOfPlant`, the input parameter(s), return arrow, and return type.

Declaring Protocols that Have Properties

Protocols can also define properties. These can be either stored or computed; the protocol makes no distinction, although it does specify whether the protocol can only be read or can both be read and written to.

The following protocol specifies a read-write property:

```
protocol PlantProtocol {  
    var a: Float { get set }  
}
```

The following protocol specifies a read-only property:

```
protocol PlantProtocol {  
    var b: Int { get }  
}
```

Declaring Protocols with Optional Methods/Properties

If a method or property is included in a protocol, it is required. However, it is often desirable to make use of a protocol but to not *require* particular methods or properties be used by a particular class that conforms to the protocol. They can be used or not, depending upon what is required for the particular app.

The following protocol definition specifies a method and a property that are both optional. (Again, this has nothing to do with Swift optional values.) A third property is also specified, but it is still required.

```
@objc protocol CalculationProtocol {  
    optional func doACalculation(c Int) -> Double  
  
    optional var a: Float { get set }  
  
    var b: Int { get }  
}
```

To be able to mark a method or property as optional, you must put the `@objc` attribute in front of the `protocol` keyword. This normally specifies that a Swift class, method, or property can be accessed from Objective-C. But it is also used for another purpose—indicating that a protocol definition has a definition of something that is optional.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 32 exercises, go to

understandingswiftprogramming.com/32

33 Extensions

Extensions in Swift allow you to add new functionality to an existing class. (Extensions also work with structures and enumerations.)

Extensions can only add new functionality. They cannot change the behavior of, that is, override, existing functionality. In other words, you can add a new method to an existing class, but you cannot modify the code of an existing method of that class.

It is not possible to use an extension to add a stored property to a class or structure (and enumerations don't have stored properties).

In an environment in which you are dealing with a lot of code with many programmers, it is often considered a good practice to allow the extension of code libraries but not to allow their modification.

Extensions also make it possible to modify classes even when you do not have the source code, so it's possible to extend even basic Swift and other iOS classes.

What can be extended? You can add instance methods, class (type) methods, computed properties, initializers, custom subscripts, and nested types. You can also specify that an existing type now conforms to a protocol.

How do you do an extension? It's pretty simple. You use the same syntax that you would if you were putting the code in the original, just prefaced by the `extension` keyword and the name of the class (or structure or enumeration) you are extending and with braces surrounding the new code. So if you want to extend the class `Fruit`:

```
extension Fruit {  
    // New code—method, initializer, computed property, etc  
}
```

There is no name for the extension.

If you don't have the source code for a class and want to add something to it you don't

have any choice but to use an extension. If you do have the source code, then it's a choice of what makes sense. If you have a package of code that you want to keep well-maintained you might prefer to add things individually for special cases using extensions rather than changing the package of code.

Adding a Method

If I have an existing class named `Fruit` and I want to add a new method to it called `showValueOfHasSeeds`, I can do it as follows:

```
extension Fruit {  
    func showValueOfHasSeeds () {  
        if self.hasSeeds {  
            print("This particular fruit HAS seeds")  
        }  
        else {  
            print("This particular fruit does NOT HAVE seeds")  
        }  
    }  
}
```

This method just prints something different depending upon the value of a property of the class `Fruit`. The class `Fruit` obviously must already have the stored property `hasSeeds` defined.

This is an instance method, and is called as follows:

```
let aParticularPieceOfFruit = Fruit()  
aParticularPieceOfFruit.showValueOfHasSeeds () // Prints: This  
particular fruit does NOT HAVE seeds
```

Instance methods can be added to structures and enumerations as well as classes. If a method is added to a structure or enumeration that changes a property, it must have the `mutating` keyword in front of the `func` keyword.

The class method that was added with the extension can be executed as follows:

```
Fruit.showNumberOfFruits () // Prints: 27
```

These examples show how to add an instance and a class (type) method to a class. Adding these kinds of methods to structures and enumerations is done in exactly the same way.

Adding a Computed Property

A computed property, whether it is an instance or a class (type) computed property, involves adding the code that gets executed when the computed property is accessed.

```
extension Fruit {  
    var weightInKg:Float {  
        get {  
            return weightInLbs*2.20462  
        }  
        set (newValue) {  
            weightInLbs = newValue/2.20462  
        }  
    }  
}
```

This is an instance computed property that calculates the weight of a piece of fruit in kilograms, based on the value of the property `weightInLbs`, which actually stores the value of the weight.

```
let aPieceOfFruit = Fruit()  
aPieceOfFruit.weightInLbs = 0.5  
print(aPieceOfFruit.weightInKg) // Prints: 1.10231
```

This creates an instance of the class `Fruit`, then sets the value of its weight, expressed in pounds.

The code in the `set` part of the computed property definition is executed when the `print` statement references the `weightInKg` property, which causes the weight in kilograms to be calculated and then printed.

The code can also allow setting the weight of the piece of fruit when expressed in kilograms:

```
let aPieceOfFruit = Fruit()  
aPieceOfFruit.weightInKg = 1.0
```

This would cause the code in the `set` part of the computed property definition to be executed, which converts the value and stores it in the property `weightInLbs`.

A class or type computed property is done in the same way, except that a *class* or *static* keyword is added, depending upon whether the original type is a class, structure, or enumeration.

Adding an Initializer

Initializers can be added to a class or structure, but there are limits. Only a convenience initializer can be added to a class, not a designated initializer. Initializers can be added to structures under appropriate conditions.

Adding a Custom Subscript

Custom subscripts can be added that did not exist before. This is done in just the same way that a custom subscript would be added in the original code. They can be added to classes, structures, and enumerations.

Adding Nested Types

Nested types can be added that did not exist before. This is done in just the same way that a nested type would be added in the original code. They can be added to classes, structures, and enumerations.

Indicating Conformance with a Protocol

An extension can indicate that a class (or structure or enumeration) conforms to a particular protocol. This might be the case as a result of adding new functionality that makes the class now conform to a given protocol. It might also happen that an existing class (or structure or enumeration) actually did have the functionality to conform to a given protocol, but the class/structure/enumeration definition did not indicate this conformance.

Protocol Extensions

New in Swift 2, implementations of methods and stored properties can be added to protocols by using an extension. See Chapter 34 on “Protocol Oriented Programming.”

What Can Be Extended?

Stored properties

Instance methods ✓

Class (type) methods ✓

Computed instance properties ✓

Computed class (type) properties ✓

Initializers ✓

Custom subscripts ✓

Nested types ✓

Conformance with protocol ✓

Protocols (methods, properties) ✓

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 33 exercises, go to

understandingswiftprogramming.com/33

34 Protocol Oriented Programming



**“You wanted a banana but what you
got was a gorilla holding the banana
and the entire jungle.”**

-- Joe Armstrong

At the June, 2015 Apple World Wide Developer's conference, the most popular presentation was Dave Abrahams' talk on "Protocol Oriented Programming."

Abrahams revealed that Swift had been designed from the beginning—with its lightweight alternatives to classes such as structures and enumerations, and its emphasis on protocols—to be what he called "the first protocol oriented programming language." This description was met with broad applause. Abrahams described protocols as being "at the heart of" the design of the language.

Protocol oriented programming is something that the Apple people seemingly hope will be a very big deal for Swift, for Swift's potential emergence as a more generally used language, and for programming methodology in general.

Swift 2 includes a capability for "protocol extensions," which allows methods and properties to be implemented in extensions to protocols and in this way obtained and used by classes, structures, and enumerations. Since a class, structure, or enumeration can conform to more than one protocol, and thus obtain methods and properties from multiple protocols, protocol oriented programming in Swift seems to be in this one sense more powerful than class-based inheritance, which (in Swift) only allows inheritance from one superclass.

In this chapter I will first describe some of the criticisms that have been made against classes and inheritance. I will then discuss how protocols might do things better. I'll describe various ways of using protocol extensions and the so-called Self requirement used with protocols. I'll finish with a discussion of some of the implications of protocol oriented programming.

What's Wrong with Classes and Inheritance?

Object-oriented programming has become so dominant and so widely practiced that it is easy to believe that the approach must be the best that software technology has to offer. In fact, however, there's been a lot of skepticism expressed by people with rather distinguished backgrounds. Much of this has been directed at classes and inheritance. Alan Kay has been quoted as saying “I invented the idea of object-oriented programming, and I can tell you that I did not have C++ in mind”, and “Java is the most distressing thing to hit computing since MS-DOS.”

Edsger Dijkstra, a Dutch academic and Turing award winner, was highly critical of object-oriented programming. (The quote that has circulated most widely, however, that “Object oriented programming is an exceptionally bad idea which could only have originated in California,” is apparently bogus.)

Of course, it is common for programmers to believe that “the language I use is great, but yours is garbage.” What is perhaps more interesting and credible are statements from practitioners of object-oriented programming, such as the authors of the well-known book *Design Patterns*, written by the so-called Gang of Four. They and others warn about the overuse and abuse of inheritance, and one of the principles they suggest is “favor composition over inheritance.” In other words, use the design pattern of composition instead of inheritance unless you really need inheritance. Composition is the use of classes in combination without any hierarchical relationship between them. Thus, a class that needs to use a method defined in another class will create an instance of that class (or access a class method directly), execute a method from that other class, and use the result. Often, a class will define a property that holds an instance of another class to allow it to quickly and easily use the methods of that other class. Both inheritance and composition are intended to avoid the necessity of duplicating methods (which makes maintenance a nightmare) by allowing a class to use methods contained in another class.

What is specifically wrong with inheritance? There are three general criticisms:

1. Inheritance too often isn't a good way to represent reality, and is inflexible over time.
2. Inheritance, as usually implemented, involves an automatic sharing of data that is unsafe and/or inefficient.
3. Inheritance is too intrusive, in a paradigm that claims to avoid tight coupling among different objects.

I'll discuss each of these in some detail:

NOT A GOOD, OR STABLE, REPRESENTATION

The use of classes and inheritance in object-oriented systems was originally considered a good idea largely because objects and their relationships were supposedly good models of the real world. And classes and inheritance were thought to be a good way to represent the real world in part because of the apparent success of two kinds of categorization and classification systems: the taxonomic system in biology, and the classification systems used for books in libraries.

Biological systems, of course, have anomalies. There are penguins (birds that cannot fly), bats (mammals that *can* fly), and whales (mammals that live in the ocean.) These cases, though annoying, could presumably be handled in software classes by overriding inherited properties and methods that did not apply to a particular class.

Evolution, however, proceeds slowly. Software requirements, in contrast, often change very quickly. And one of the biggest problems with inheritance is its inflexibility. We can see the effect of change quite clearly by looking at the Dewey Decimal system category for Religion (from Clay Shirky's paper "Ontology is Overrated"):

200: Religion

- 210 Natural theology
- 220 Bible
- 230 Christian theology
- 240 Christian moral & devotional theology
- 250 Christian orders & local church
- 260 Christian social theology
- 270 Christian church history
- 280 Christian sects & denominations
- 290 Other religions

Readers in the early 20th century West (when this system was created) were apparently not much interested in reading about non-Christian religions, which of course had far more adherents than Christianity. In the early 21st century, when the world is smaller, information about other religions is more accessible, and terrorists are crashing airplanes into buildings in the name of Islam, Western readers do have an interest.

However, it is not clear that software developers defining classes and class hierarchies are any better at predicting the future than librarians.

The library systems also have an inherent bias that has nothing to do with their apparent claim to be classifying general knowledge. Here's the Library of Congress system for History, again from Shirky:

D: History (general)

DA: Great Britain

DB: Austria

DC: France

DD: Germany

DE: Mediterranean

DF: Greece

DG: Italy

DH: Low Countries

DJ: Netherlands

DK: Former Soviet Union

DL: Scandinavia

DP: Iberian Peninsula

DQ: Switzerland

DR: Balkan Peninsula

DS: Asia

DT: Africa

DU: Oceania

DX: Gypsies

The Netherlands is at the same level as all of Asia and all of Africa.

As Shirky points out, the system doesn't reflect general knowledge, but only the number of books on the shelf at the Library of Congress for a particular topic.

And this "book on a shelf" observation points to another problem—the assumption that the particular piece of knowledge being categorized can be done so with a single

dimension of categorization. This is a convenient assumption for libraries that (until recently) kept all their knowledge in the form of books on a shelf. A classifier picked the best classification and put the book on a particular shelf based on that single classification. Using more than one primary classification makes no sense, because a physical book can be on only one shelf. If a book is on the history of art, the classifier has to decide whether it is more about history or more about art, and put it on the appropriate shelf. (The system does have secondary classifications, and an art history book on the history shelf could be found for those looking for art, with additional effort, by such a cross-reference.)

In the digital world, however, as Shirky puts it, “there is no shelf.” The implications are, for Shirky, are that there is no need for professional classifiers, for a standard (inheritance-based) classification system, or even for categories. It can all be done with links and tags, added by individuals.

Software, of course, and most of the real world it attempts to model, never had a shelf, and thinking that library classification systems, and hierarchies, are a good basis for representing knowledge generally, is a bad idea.

AUTOMATIC SHARING OF DATA

Another problem with inheritance is that a class and its superclasses all have references to the single copy of data kept in the heap part of the memory. Methods from these different classes can access this data essentially simultaneously, leading to the typical difficulties with such simultaneous access. While locks and similar threading mechanisms can prevent these issues, they add complexity and are difficult to implement in a safe and reliable manner. This often increases processing overhead and results in bugs.

INHERITANCE TOO INTRUSIVE

It is perhaps ironic that a common complaint about inheritance is that superclasses often excessively intrude upon their subclasses. The reason why this is ironic is the near-paranoid devotion to encapsulation and security voiced by so many commentators on object-oriented programming and embedded in many languages (although more recent language implementations seem to be more relaxed about this issue), and the insistence upon encapsulation and inheritance as two of the supposed fundamental principles of object-oriented programming.

The more complete “gorilla holding the banana” quote from the cartoon at the beginning of this chapter is:

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

—Joe Armstrong

“Loose coupling” between objects, (or, more generally, parts of a program) is considered highly desirable as part of the general devotion to the principle of encapsulation. However, inheritance often makes it difficult to have loose coupling between objects that are instances of related classes because inheritance, and, particularly, chains of inheritance, often results in large numbers of properties and methods being inherited that substantially affect that object in a way that is difficult for the programmer to predict or keep track of. This is a frequent source of bugs.

Can Protocols Do it Better?

The protocol oriented programming approach is based on the assertion that in many situations, a protocol, along with a structure or enumeration, can work as well or better as a class would in the same situation, and without the negative side effects.

As part of the Swift design, structures and enumerations were developed that could do most of what classes could do.

We can see this most clearly with structures, which are very much like classes but with two critical differences: They do not have inheritance (and its baggage), and they are value types. That is, new instances of structures copy data, rather than reference it, so that there are none of the problems of automatic sharing of data.

Structures, like classes, encapsulate information and behavior in the form of properties and methods. Like classes, they can create instances. They provide a mechanism for controlling access to their code. They provide a namespace—code provided in the methods of a structure is accessed through the structure, avoiding collisions in naming. Structures provide a foundation for abstraction, and, like classes, are extensible.

Enumerations, perhaps less transparently, provide these same capabilities. They have the same ability to define methods and create instances, and in fact have something like stored properties, in the form of “associated values”. They too are value types rather than reference types and thus avoid the automatic sharing problem.

The capability of protocol extensions to define implementations for methods and properties that can then be used by whatever structures and enumerations (and, indeed, classes) that specify that they conform to those protocols provides a mechanism that is similar to inheritance but that acts in a much more restrained manner. It is more flexible than inheritance, because a given instance-creating type (that is, a class, structure, or enumeration) can specify conformance to more than one protocol. And a given protocol, and its methods and properties, can be specified by more than one instance-creating type without regard to where it is in any knowledge representation. Protocols work much more like Clay Shirky’s idea of having “no shelf” and “no categories”, allowing knowledge to be represented in a more flexible, less rigid manner. Using protocols rather than inheritance is the difference between going into a restaurant and ordering strictly a la carte versus getting their No Substitutions Allowed combo special.

Because the decision about what protocols a new instance-creating type will conform to is made at a more local and individualized level, a programmer is much less likely to get a

big surprise (and bug) from unexpectedly obtaining a method or property than would be the case with a large, and long, inheritance chain.

Thus, the use of structures (or enumerations) along with protocols provides the main things that classes provide with a flexible form of knowledge representation that in many situations will be superior than classes, without automatic sharing of data or the excessive intrusiveness of inheritance.

Protocols vs Protocol Extensions

There is perhaps some confusion between what defining a protocol can do and what defining a “protocol extension” can do. The latter uses the keyword `extension` and the name of a previously defined protocol.

A protocol definition specifies a name and defines what properties and methods are required if an instance creating type (class, structure, or enumeration) is to conform to that protocol. Defining a required method means providing the signature for that method, including its name and the types of input parameters and the return value. Defining a required property means defining the name and type of that property and whether it is a read-only and read-write property. It is also possible to indicate conformance of a protocol to a previously defined protocol or protocols, which results in properties and methods being inherited, much like class inheritance works.

A protocol extension (that is, an extension to a previously-defined protocol) could always indicate additional methods that were required for conformance to a protocol. What is new in Swift 2 is that a protocol extension can now include the implementation of one or more required methods and properties.

A Basic Protocol Extension

Suppose we have defined the following protocol:

```
protocol TypicalSquirrel {  
    var nameOfIndividual: String { get }  
    var nameOfSpecies: String { get }  
    var skinHasFur: Bool { get }  
    var breathesAir: Bool { get }  
    var whyYouShouldNotPetThem: String { get }  
  
    func sayWhetherICanFly()  
}
```

This defines some interesting things we might want to know about a particular squirrel. We require that each individual creature (instance) have a name, and that a name be provided for the species that it is a member of. A `{ get }`, or a `{ get set }`, is required when properties are defined in protocols and indicates whether the property is read only (`{ get }`) or read-write (`{ get set }`). We also require that the instance indicate whether the squirrel has fur or not, whether or not it breathes air, and a reason for why you should not pet them.

The protocol also requires that a function be implemented, `sayWhetherICanFly()`.

We don't want to bother defining the values for `skinHasFur` and `breathesAir` every time we define a squirrel, because they are always true. Still, at some point our program might need these values. We also don't want to bother implementing the method `sayWhetherICanFly`, since we rarely encounter squirrels that can fly. They almost always can't.

Thus, we provide an extension to the `TypicalSquirrel` protocol as follows:

```
extension TypicalSquirrel {  
    var skinHasFur: Bool { return true }  
    var breathesAir: Bool { return true }
```

```

var whyYouShouldNotPetThem: String { return "They might
have fleas with the plague" }

func sayWhetherICanFly() {
    print("I am a typical squirrel and I can walk and climb
trees but not fly.")
}

}

```

Why don't we just use a class and inherit information and behavior from a big class like `Mammal`? Well, we could—but the general approach of protocol oriented programming is based on a bit of suspicion about whether we might get in trouble if we do this. Are there things about mammals—perhaps implemented in some third-party code written by people we don't completely trust—that might not be true of our squirrels? It is safer to use smaller components that, because they are intended to be more specific, are more likely to be trustable.

So we then define a structure that allows us to create instances of squirrels, named `Squirrel`, and we adopt the `TypicalSquirrel` protocol so that we can obtain the default characteristics and behavior of typical squirrels:

```

struct Squirrel: TypicalSquirrel {
    var nameOfIndividual: String = ""
    var nameOfSpecies: String = ""
}

```

We include the properties `nameOfIndividual` and `nameOfSpecies` in this structure because the protocol requires them. We also provide an initial value. We don't include either for properties that are already defined in the protocol extension. We don't include the method that is defined in the protocol extension.

We can then create a representation of an individual squirrel named “Pete” (an instance of the structure `Squirrel`):

```

var pete = Squirrel(nameOfIndividual: "Pete",
nameOfSpecies: "Gray Squirrel")

```

And we can print out some of his characteristics:

```
print(pete.nameOfSpecies) // Prints: Gray Squirrel  
print(pete.skinHasFur) // Prints: true  
print(pete.breathesAir) // Prints: true  
print(pete.whyYouShouldNotPetThem)  
// Prints: They might have fleas with the plague
```

And execute the one method that indicates some of his behavior:

```
pete.sayWhetherICanFly()  
// Prints: I am a typical squirrel and I can walk and climb trees but not fly.
```

What to Do When Squirrels Can Fly

This is fine. But say we drive up to the mountains. And in a remote area, high up, we meet another squirrel, George. And he is a Northern Flying Squirrel.

He's pretty rare. In any particular place, it is likely that there will be only one species of flying squirrels. It's thus not worth creating a protocol just for them. So perhaps we just create a different structure we call `FlyingSquirrel`. We also use the protocol `TypicalSquirrel`, but we effectively override a few things. We provide a different value for `whyYouShouldNotPetThem` than the protocol extension provides. And we provide a new method that overrides the implementation of the method `sayWhetherICanFly`.

```
struct FlyingSquirrel: TypicalSquirrel {  
    var nameOfIndividual: String = ""  
    var nameOfSpecies: String = ""  
    var whyYouShouldNotPetThem: String = "They are not  
friendly and have sharp teeth."  
  
    init (nameOfIndividual: String, nameOfSpecies: String) {  
        self.nameOfIndividual = nameOfIndividual  
        self.nameOfSpecies = nameOfSpecies  
    }  
  
    func sayWhetherICanFly() {  
        print("I am a flying squirrel and I can walk and climb  
trees and also fly, or at least glide.")  
    }  
}
```

Note that we do not include the keyword `override` with our new implementation of that method. (We would need this if we were using a class and overriding a method that was inherited from a superclass, but not with protocols.)

We can then create an instance of `Flying Squirrel` named “George”:

```
var george = FlyingSquirrel(nameOfIndividual: "George",  
nameOfSpecies: "Northern Flying Squirrel")
```

And print out some of his information:

```
print(george.nameOfSpecies) // Prints: Northern Flying Squirrel  
print(george.skinHasFur) // Prints: true  
print(george.breathesAir) // Prints: true  
print(george.whyYouShouldNotPetThem)  
// Prints: They are not friendly and have sharp teeth
```

And execute the one method that indicates some of his behavior:

```
george.sayWhetherICanFly()  
// Prints: I am a flying squirrel and I can walk and climb trees and also fly, or at least  
glide.
```

This is a book about programming, not biology, so I won't get into the issue of whether flying squirrels actually fly. But see the animation of a flying squirrel coming in for a landing and decide for yourself at:

<http://understandingswiftprogramming.com/flyingsquirrel>

Note that we could have, as an alternative, created an additional protocol and protocol extension named `TypicalFlyingSquirrel`, and done things in the way that we did them for `Squirrel`. But you don't want to create zillions of protocols.

Using Protocols for Conditional Implementations

Now let's say that you are a biologist working in Monterey, California, and spend most of your time at the shore of the ocean, or on a boat. For most people in most places, if they see a mammal, it will be one that lives on land. Here the situation is reversed: even on the beach, the mammals you see, aside from humans, will likely be those that live in the ocean. Here we might be especially suspicious of the idea of using a class and inheriting stuff from a very large, complicated class like all mammals.

So we will use instead a protocol `MarineMammal`. And because some marine mammals (e.g., seals and sea lions) can walk on land, we'll define a second protocol `CanWalkOnLand`.

We will use a protocol extension to define some behavior, but that behavior depends on whether a structure conforms to both protocols.

We start by defining the `MarineMammal` protocol:

```
protocol MarineMammal {  
    var nameOfIndividual: String { get }  
    var nameOfSpecies: String { get }  
    var skinHasFur: Bool { get }  
    var breathesAir: Bool { get }  
}
```

We need a protocol extension for some basic things about marine mammals that we don't want to set every time:

```
extension MarineMammal {  
    var skinHasFur: Bool { return false }  
    var breathesAir: Bool { return true }  
}
```

We then define the protocol `CanWalkOnLand`:

```
protocol CanWalkOnLand {  
    func iCanWalkOnLand()  
}
```

The protocol `CanWalkOnLand` requires that an instance-creating type implement the function `iCanWalkOnLand()`, but does not actually provide an implementation for that function.

We then implement a protocol extension that provides an implementation for the `iCanWalkOnLand()` method, but that is only implemented if the instance-creating type we are using conforms to both the `MarineMammal` protocol and also the `CanWalkOnLand` protocol:

```
extension MarineMammal where Self: CanWalkOnLand {  
    func iCanWalkOnLand() {  
        print("I can walk on land.")  
    }  
}
```

We can define a structure for whales:

```
struct Whale: MarineMammal {  
    var nameOfIndividual: String = ""  
    var nameOfSpecies: String = ""  
}
```

Since whales cannot walk on land, it conforms only to the protocol `MarineMammal`. The function `iCanWalkOnLand` cannot be accessed.

```
var cynthia = Whale(nameOfIndividual: "Cynthia",  
nameOfSpecies: "California Gray Whale")  
print(cynthia.nameOfSpecies) // Prints: "California Gray Whale"  
cynthia.iCanWalkOnLand() // Compiler Error
```

We can now define a structure for seals and sea lions:

```
struct SealsAndSeaLions: MarineMammal, CanWalkOnLand {  
    var nameOfIndividual: String = ""  
    var nameOfSpecies: String = ""  
}
```

This conforms to both the `MarineMammal` and `CanWalkOnLand` protocols. And because of the conditional protocol extension that implements the method `iCanWalkOnLand()`, that method will be available:

```
var sammy = SealsAndSeaLions(nameOfIndividual: "Sammy",  
nameOfSpecies: "Harbor Seal")  
  
print(sammy.nameOfSpecies) // Prints: Harbor Seal  
sammy.iCanWalkOnLand()  
// Prints: "I can walk on land."
```

Adopting Multiple Protocols That Have Duplicate Methods

What happens if you use extensions to protocols that result in multiple implementations of the same method?

For example, we might define protocols for `Mammal` and `MarineMammal`, and require that a method be provided in both cases, the method `sayCanIWALKOnLand`.

We then provide protocol extensions that implement the method for each protocol.

In the case of `Mammal`, the method is:

```
func sayCanIWALKOnLand () {  
    print("I am a Mammal and I CAN walk on land")  
}
```

In the case of `MarineMammal`, the method is:

```
func sayCanIWALKOnLand () {  
    print("I am a Marine Mammal and I CANNOT walk on land")  
}
```

If you then define a structure, say for `HarborSeal`, create an instance, and try to call this method, what will happen? Which method will be executed?

The answer is that Swift (based on Swift 2 beta 2) will not allow you to create a structure that adopts multiple protocols that each have a function with the same name. The compiler gives you an error.

```
protocol Mammal {  
    func sayCanIWALKOnLand()  
}  
  
protocol MarineMammal {  
    func sayCanIWALKOnLand()
```

```

}

extension Mammal {
func sayCanIWALKOnLand() {
print("I am a mammal and I CAN walk on land")
}
}

extension MarineMammal {
func sayCanIWALKOnLand() {
print("I am a marine mammal and I CANNOT walk on land")
}
}

struct HarborSeal: Mammal, MarineMammal {
}

var seal = HarborSeal()
seal.sayCanIWALKOnLand()

```

This is the sequence of code. It allows `Mammal` and `MarineMammal` to be defined and to have extensions that define the function `sayCanIWALKOnLand`, which is reasonable. But it blows up on the definition of the structure `HarborSeal` with the following error message: multiple matching functions named ‘`sayCanIWALKOnLand`’.

Protocol oriented
programming is nice, but it
still has a lot of object stuff
in it, so it should really be
called “Protocol object
oriented programming”.

I don't think so.

Protocol Oriented Programming
POP

Object Oriented Programming
OOP

Protocol Object Oriented Programming
POOP



Extending Behavior of the Swift Standard Library

A frequent use of protocol extensions will likely be to provide methods that are added to the Swift standard library, and perhaps also third-party libraries.

Extensions to specific types can be added in Swift 1 by using simple extensions. For example, in standard Swift it is not possible to access a String using a subscript containing an integer. The reason is that Swift strings use Unicode and a single character can be up to four bytes long, such as an emoji or logographic (e.g., Chinese) character.

However, if you know that the strings you are dealing with contain only ordinary ASCII characters, you can use an extension to String to add the capability of accessing a character via a subscript using an integer.

In Swift 2, you can use protocol extensions to add capabilities like this. They allow you to add capabilities not to just a single type, but to multiple types at once. Thus, for example, by using a protocol extension specifying the protocol `Collection Type`, you can add a new capability to arrays, dictionaries, and sets (which all conform to the `CollectionType` protocol.)

The Self Requirement for Protocols

In an example shown by Abrahams to argue for the superiority of protocols and structures over classes in certain situations, he has a function that determines what order two values should be placed in. Such a function is commonly needed for sorting an array or for using binary search with an array. It is difficult to provide a general solution for such ordering because the correct order can depend upon the type of the values and there is often more than one reasonable way to do ordering. (The integer 5 is always greater than the integer 4, but what order should the strings “05” and “4” be in?)

In Abrahams’ preferred solution, making use of a protocol and a structure, he uses the following code:

```
protocol Ordered {  
    func precedes(other: Self) -> Bool  
}  
  
struct Number : Ordered {  
    var value: Double = 0  
    func precedes(other: Number) -> Bool {  
        return self.value < other.value  
    }  
}
```

In the protocol, it specifies that for the instance creating type (a structure in this case) that conforms to the protocol, a function is required with the given signature, which includes an input parameter with a type of `Self`. This means that the input parameter must be of the same type as the structure that is being defined, `Number`.

(In Swift, a reference to “self” (all lower case) refers to the current instance of an instance creating type such as a class or structure. A reference to `Self` (first letter only in upper case) refers to that instance creating type. Note that `Self` can also be used in where clauses that have nothing to do with setting the `Self` requirement that is discussed here, but just refer to the current type.)

This reference to `Self` goes beyond simply indicating the type of the input parameter.

Every protocol either is subject to or is not subject to what is known as the “Self requirement.” This is achieved in one of two ways.

The first way is by having that protocol specify `Self` as a type in a signature that is required by that protocol.

The second way is for that protocol to conform to another protocol that itself either conforms to the `Self` requirement or to a protocol that does. (That is, conformance to the `Self` requirement can be passed along indirectly like an inheritance chain.)

As an example, any protocol that conforms to the `Equatable` protocol will itself have a `Self` requirement.

Whether a class, structure, or enumeration is subject to a `Self` requirement has important implications for how that class, structure, or enumeration's code gets executed, how types it uses must be specified, and what types are allowed in collections that code may operate on. Execution of code with a `Self` requirement versus execution without are two different worlds. The differences include:

Static or dynamic dispatch. A class, structure, or enumeration that is not subject to the `Self` requirement is dispatched dynamically, meaning that the mapping between the reference to the name of a method and the code that executes it is determined at runtime. In contrast, code subject to a `Self` requirement is dispatched statically, meaning that the mapping is determined at compile time. Static dispatch is desirable because it often allows the compiler to perform optimizations that it would otherwise be unable to do.

Generic Type Symbols Required. A class, structure, or enumeration that is not subject to a `Self` requirement can use the normal types (e.g., `Int`, `Double`, `Float`, `String`) in its methods and properties. However, if it is, that instance-creating type must specify all types by generic symbols.

Homogeneous Collections. Another requirement imposed on classes, structures, and enumerations that are subject to the `Self` requirement is that collections must be homogeneous. That is, an array, set, or dictionary must contain elements that are of the same actual type, not the more liberal “same type” allowed by subtyping (the subtype polymorphism principle.) This may require that elements in collections be downcast before they are used in code.

Implications of Protocol-Oriented Programming

Swift has provided us with two new ways of using object-oriented programming without inheritance. The first, in the initial design of Swift, was structures and enumerations, which do most of what classes can do except for inheritance, and are based on copying rather than referencing memory.

Now, in Swift 2, protocol extensions allow implementations of methods and properties to be part of protocols, and allow structures and enumerations, as well as classes, to obtain these implementations automatically if they conform to the protocol. This allows something like inheritance, even multiple inheritance, but without inheritance chains, based on a flat rather than hierarchical model of knowledge.

This gives us quite a few choices. The principle of “favor composition over inheritance” provided in the *Design Patterns* book is still valid, but composition can now be realized with structures and enumerations as well as classes. Implementations can be made part of protocols if it makes sense for those implementations to be used in more than one instance-creating type.

We don’t yet know what the implications of protocol-oriented programming will be. If Apple really believes in protocol oriented programming, perhaps there will be an effort to rewrite Cocoa Touch with new APIs that are not only more oriented toward Swift but organized not as inheritance chains but with protocols.

Will protocol oriented programming, along with such other Swift attractions as type and value safety and fast execution, lead to Swift becoming popular as a server-side language in the same way that Java, Python, and Ruby are now used? Will these other languages adopt some aspects of protocol oriented programming?

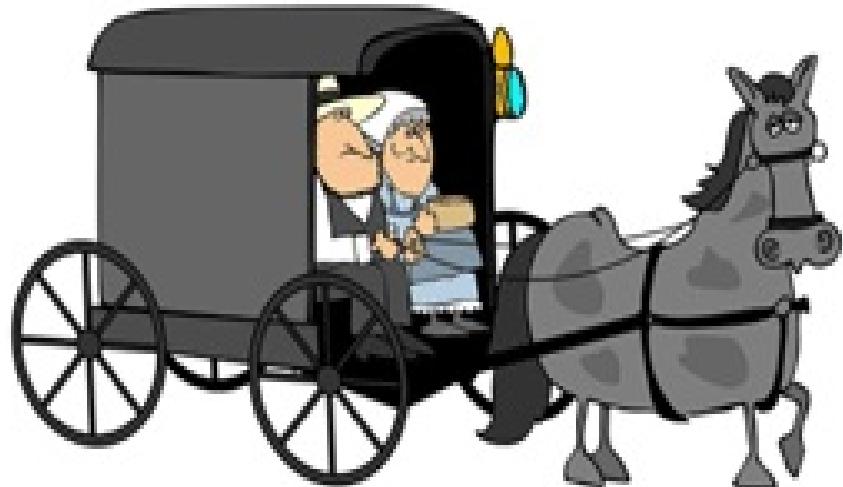
It is not clear whether how revolutionary protocol oriented programming will turn out to be, but its development not only provides seemingly valuable additional capabilities, but also a lot of encouragement for programmers to think more about the architecture of their apps and to avoid the overuse and abuse of inheritance.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 34 exercises, go to
understandingswiftprogramming.com/34

35 Building Mixed Swift and Objective-C Apps



When am I going to switch to Swift? It's up to my boss. It looks like he's on his way here, so we can ask him. But he really likes Objective-C.

It is quite easy to build an app using both Swift and Objective-C. The biggest problem is actually switching between languages: once you get used to not typing semicolons in Swift, it's hard to remember to put them in for Objective-C. And when you do, they look funny, as do many other things about the language.

You can create a project in either Swift or Objective-C, and then create files in the other language and call them from the language you started with.

Calling Objective-C Code From a Swift Project

We can see a simple example of calling Objective-C code from a project that is started up in Swift. This is the most common scenario. For example, you might have existing Objective-C code that you want to reuse in a new project.

The standard way to demonstrate code with Xcode is to use the “Single View Application” template. You first create a new project by going to File > New > Project and then select the “Single View Application” template. (“Application under IOS” should be selected in the left column.) Click on Next and you will get a window that asks for a Project Name (use “MixedCode1”), Organization Name (whatever you want to call yourself) and an Organization Identifier (anything). “Language” has a menu with “Swift” and “Objective-C” as choices and you should select “Swift”. Under Devices select iPhone and do not check the box for Core Data. Clicking Next again will give you a popup that allows you to select a place to put the project in your file system, “Desktop” is reasonable and clicking on “Create” will create the project files.

In the leftmost column you will see a file called “ViewController.swift” that contains the Swift code for the `ViewController` class, including a `viewDidLoad` method as part of a template. (Single click on it.) The template looks like this:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    // Do any additional setup after loading the view,  
    // typically from a nib.  
}
```

Just below the “Do any additional setup” line, add the following code:

```
1 print("Hi from the Swift code")  
2 let hi: HiFromObjectiveC = HiFromObjectiveC()  
3 hi.sayHi()
```

(If you are running this on an Xcode version earlier than Xcode 7, which has Swift 2, use “`println`” rather than “`print`”.)

This simple program starts up, writes “`Hi from the Swift code`” to the console

log, and then calls an Objective-C method that writes “Hi from the Objective-C code” to the console log. The first line here does the “Hi from the Swift Code” greeting.

The second line creates an instance of the class `HiFromObjectiveC` and assigns a reference to it to the constant `hi`. The third line calls a method in the `HiFromObjectiveC` class, `sayHi`.

The `sayHi` method, written in Objective-C, looks like this:

```
- (void) sayHi {  
    NSLog(@"Hi from the Objective-C code");  
}
```

To create the files to allow us to call this method, we mostly have to do the same thing we would do if we were adding a new class to an Objective-C project. We use Xcode to create a new class (a subclass of `NSObject`), selecting Objective-C as the language. With the Xcode menu at the very top of your screen, select File > New > File, then choose “Source” under iOS in the left column, and Cocoa Touch Class from the icons in the right column. Click on Next. In the next screen we provide a name for the class, e.g., `HiFromObjectiveC`, choose the class that this is a subclass of, `NSObject`, and choose Objective-C as the language. (Make sure it is not Swift.)

After clicking Next, you should be prompted with the directory that you created the project in. If you click on Create, it will create two new files: `HiFromObjectiveC.h` and `HiFromObjectiveC.m`.

When Xcode creates these files, it will do one extra thing: it will ask you if you want a bridging header. See the screenshot below. You should say “Yes”.



It will then create a third file, called something like MixedCode1-Bridging-Header.h.

In the .h (header) file for HiFromObjectiveC, it will initially look like this (I've removed some blank lines):

```
#import <Foundation/Foundation.h>
@interface HiFromObjectiveC : NSObject
@end
```

You just need to add the following line after the @interface line:

```
- (void) sayHi;
```

This tells the outside world that the class HiFromObjectiveC has a method called sayHi that has no input parameters and that returns no values, and that can be executed from instances of the class.

Then you need to add the actual method to the .m (implementation) file. When you look at that file, it will initially look like this:

```
#import "HelloFromObjectiveC2.h"
@implementation HelloFromObjectiveC2
@end
```

You need to add the following after the line that starts with "@implementation":

```
- (void) sayHi {
    NSLog(@"Hi from the Objective-C code");
}
```

This implements the method sayHi.

We then need to do one more thing: There should be a file called something like `mixedCode1-Bridging-Header.h`. Click on it. It will look like this:

```
//  
// Use this file to import your target's public headers  
// that you would like to expose to Swift.  
//
```

We add the following line to it:

```
#import "HiFromObjectiveC.h"
```

This tells the Swift code that there is an Objective-C class it can call and provides the name of the header file it can get information from about the interface for that class.



To build the project and run it, click on the “Run” button (shown above) in the upper left hand corner of the Xcode window:

If we run this project, we get the following:

```
Hi from the Swift code  
2015-06-29 11:04:47.244 MixedCode1[1451:53739] Hi from the  
Objective-C code
```

The time and date stamps, etc. are because Objective-C used an `NSLog` statement rather than the `print` statement that would be used from Swift.

Calling Swift Code from an Objective-C Project

We can also create an Objective-C project and call Swift code from an Objective-C method.

We again create a new project in Xcode:

Xcode > File > New > Project

Again choose a template. iOS, Application should be in the left column. The templates are in the right column. Choose Single View Application from among the icons in the right column, then click on Next.

A window will pop up and you should fill in the information. Use “MixedCode2” for the project name this time. For the Language, this time choose “Objective-C” from the menu. Click on Next.

You will again be presented with a location in your file system where it will create the project. Click on “Create” and the project, with its files, will be created.

Instead of the file ViewController.swift, you will get a pair of files called “ViewController.h” and “ViewController.m”. We’ll get to them later, but first we will create the Swift code. (I’m doing this in a particular order to avoid some error messages.)

Select Xcode > New > File, then select Source under iOS and the icon for a Cocoa Touch class. Click on “Next”.

In the following screen, we provide the class name `HiFromSwift`, and define it as a subclass of `NSObject` (which should be presented as the default). Make sure that the language is “Swift”—the default you will be presented with is “Objective-C”, and you will have to select Swift from the menu. Click on “Next”.

Xcode will then present the location in your file system where the new file will be saved, which should be the directory “MixedCode2”. Click on “Create”.

You will likely get a popup asking “Would you like to configure an Objective-C bridging

header.” The appropriate answer is “No”, since you don’t need this kind of bridging header. Instead, a bridging header with a different name (your project name followed by “-Swift.h”) will be automatically generated so that the Objective-C class will know about your Swift class.

A new file HiFromSwift.swift, containing the code for the new Swift class, will be created. It will initially look like this (I’ve removed some blank lines):

```
import UIKit  
class HiFromSwift: NSObject {  
}
```

This makes the new Swift class a subclass of NSObject.

(If you like, you can edit this to make the new class a base class, but if you do, you should put @objc before the class keyword, so that the compiler knows to make this available to Objective-C.)

We add the following code just after the line starting with “class”:

```
func sayHi () {  
    print ("Hi from the Swift code")  
}
```

This implements a method sayHi in the class HiFromSwift that prints to the console log. If you are running Xcode 6, use “println” instead of “print”.

We now have created the Swift class and method, and can go back to the Objective-C code.

You should click on the “ViewController.m” file to edit it.

At the top of the file, you will see:

```
#import "ViewController.h"
```

You should add the following line just below it:

```
#import "MixedCode2-Swift.h"
```

This will allow the Objective-C code to see the Swift class. “MixedCode2-Swift.h” is the file that was automatically generated by Xcode.

Next, scroll down and you will see the same `ViewDidLoad` method that you did in the earlier example. The template looks just a little different:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view,
    // typically from a nib.
}
```

Add the following code just after the “Do any additional setup” line:

```
NSLog(@"Hi from the Objective-C code");
HiFromSwift *hi = [[HiFromSwift alloc] init];
[hi sayHi];
```

This code will display “Hi from the Objective-C code” on the console log, and then create an instance of the (Swift) class `HiFromSwift`. It will assign a pointer to that new instance to the variable `hi`. It will then send a message to the new object `hi` specifying `sayHi` that will cause the method `sayHi` to be executed. Although it is calling a Swift method, it calls it in the way that it would call an Objective-C method.

You may see an error message or two, due to Xcode not immediately picking up the information in the header file, but they should disappear when you build the project.

Now click on the Run button to build and run the project. You should see something like the following in the console log:

2015-06-29 12:03:49.855 MixedCode2[1946:68413] Hi from the Objective-C code

Hi from the Swift code

You will not see the file “MixedCode2-Swift.h” in your project files. It is hidden, often very deep in the directory Library/Developer/ Xcode/DerivedData. If you are having difficulties, you might want to look at it to make sure that it has a reference to the Swift class you created.

The file “[name of project]-Bridging-Header.h” is only used for bridging to Objective-C files from Swift files, and it is not needed for bridging to Swift files from Objective-C. However, bridging from Objective-C to Swift can be buggy, and sometimes the compiler wants to see it. It can be easier to just include a file with the correct name just to keep the compiler happy, even though it is not used. (Although I think that in the more recent versions of Xcode this bug is gone.)

This chapter applies to using mixed Swift and Objective-C code in an app. (Apple calls this “mix and match”.) If you are building a framework, the procedure is slightly different. See the Apple documentation.

Hands-On Exercises

Go to the following web address with a Macintosh or Windows PC to do the Hands-On Exercises.

For Chapter 35 exercises, go to
understandingswiftprogramming.com/35

A What's Different About Swift?

Apple's announcement in June, 2014 of their new Swift language shocked the programming world and very probably makes Objective-C—the language normally used for building iOS apps—a dying language.

The need for a new language has been quite clear for some time. The surprise among developers was mainly that Apple actually did it and also that they had managed to keep it secret.

In looking at the language in detail there is another surprise. This language is very ambitious. Typically, mainstream programming languages, which Swift is, fall into one of two camps. They are either high-level scripting languages that are designed for ease of use (e.g., Python, Ruby), or they are lower level systems languages that are designed to be efficient (e.g., C, C++). Swift aims to be both as easy to use as a scripting language while at the same time run very fast. In addition, Swift intends to have a high degree of safety, meaning that it is designed to minimize and quickly detect errors by programmers.

Apple sets a very high bar for Swift by claiming that it is “the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language”.

In this paper I will describe the design of Swift and discuss the extent to which Swift meets its stated goals.

I will first briefly describe Objective-C, how it came to be, and why there has been a need for a new language.

Next, I will discuss some of the philosophical assumptions behind the Swift language.

I'll then go through the four aspects of Swift that Apple has indicated as the main goals for the language and describe what has been done in Swift for each of these characteristics. Swift is intended to have (1) Safety; (2) Clarity; (3) Modernity; and (4) Performance.

Finally, I'll draw some conclusions about Swift. In this discussion I will try to answer two questions:

(1) Will Apple achieve the goals it has set for Swift? and (2) What does the introduction of Swift mean for iOS app development?

What's Wrong With Objective-C?

Objective-C is peculiar as a programming language in that it was created by adding a new language on top of an existing language. The old language was C. The new language was based on Xerox's Smalltalk-80 object-oriented language and was initially a very thin layer indeed. The "Objective part of Objective-C" simply allowed the passing of "messages" to objects that either existed as part of a library or that were created by defining them in Objective-C. A "runtime" program, written in C and assembly language, interpreted the messages, which resulted in the call of methods that were eventually executed as C code. The real capability of the language was contained in the library of classes that implemented sophisticated user interface functions and systems functions.

The motivation for this hybrid language was driven largely by practical and business considerations. The original Objective-C was developed in the early 1980s to run in a computing environment that needed C because of existing code in that language. When the language was adopted by NextStep, a company that Apple later acquired so that the operating system it had developed could be used for the Macintosh, agreements were made that allowed NextStep to use the language itself and compiler as open source while the extensive library that NextStep developed that contained most of the functionality was proprietary and owned by Apple.

Objective-C was modified in 2006 to provide properties (a way of accessing instance variables of objects from outside an object) and a few other capabilities as part of "Objective-C 2.0". Later closures (called "blocks") were added, a way of passing a small chunk of code to an API along with variables and their values so that they could be executed at a later time. Capabilities for defining literal expressions (e.g. strings like @“Hello”) in arrays, dictionaries, and numbers were added that made the language a little better, but still clumsy. The ability to access arrays and dictionaries with subscripts (e.g., anArray[5] = @“Hello”) was also added.

A major addition was Automatic Reference Counting, an extension of the reference counting memory management scheme used to allow tracking and reuse of memory for objects in the heap part of the random access memory. This allowed reference counting to be used with relatively minimal effort by the programmer, (who now only needed to make sure there were no memory leaks caused by improper structures that the compiler could not detect, known as reference cycles.) This was a major advance in ease of use for programmers, who no longer had to track memory usage and specifically write code to deallocate memory. It also meant increased safety, since failure to deallocate and thus reuse memory (leading to memory leaks) and programmers attempting to deallocate memory that had already been deallocated (causing a crash) were major problems.

Despite these changes, Objective-C still has major limitations. It has safety issues. It is

messy, ugly, and often does things in convoluted ways. And, being long in the tooth, it lacks many of the things that (relatively) new scripting languages like Ruby and Python have, much less specialized languages like Haskell.

It's Not Very Safe

A safe programming language helps programmers avoid errors.

Objective-C is heavily dependent upon pointers to access objects in memory. A pointer holds a direct address of the memory location that an object (stored in the heap memory) resides in. The problem is that if a pointer gets corrupted, code that intends to write to an object's data ends up writing to the wrong object. Or, as one commentator put it, "With pointers you are one dereference (conversion to an address) away from scribbling all over memory."

Although Objective-C requires data types to be declared, it is nearly as flexible as many scripting languages in its willingness to convert data to another type when it seems desirable. Keeping types flexible was a major part of the design philosophy, which including making decisions about which method would get executed for a given message at runtime ("dynamic dispatching") rather than predetermined by the compiler.

Another common safety issue with C is programmers forgetting to put in code to deal with nil values.

In addition, Objective-C allows almost anything to be evaluated as a Boolean in an if statement, and has a confusing mess of data types that are defined in some way as Booleans.

It's Messy, Ugly, and Cluttered

Objective-C retains the division of header (.h) and implementation (.m) files used in C, requiring one for each class. Every class that refers to another class has to include a header for that class to get a program to compile. Things get messy when class A requires a header for class B, but class B requires the header for class A.

Properties must be defined in header files, along with their types, whether they are "atomic" or not (need to be written as a whole) and an ownership attribute ("strong", "weak", etc.). Header files must have signatures (names and types of input parameters and

return values) of all methods called by other classes. In general, it takes a lot of code to define a class. Often, you'll write the code in Objective-C to do something, and when you are done, you think, "I sure had to write a lot of code to do that!" Clumsy aspects of long-ago-decided C syntax, such as that used in switch statements, continue as a legacy.

A look at the syntax for blocks suggests how convoluted they can be. This declares a variable so that it can be assigned a block, then does the assignment, including the code in the block:

```
double (^blockVariable)(double a, double b);  
blockVariable = ^double(double a, double b){  
    return a * b;  
};
```

A look at modern scripting languages suggests many other aspects of its lack of clarity.

For example, the code below concatenates two strings in JavaScript:

```
var s1 = "Hello ";  
var s2 = "friend.";  
var s3 = s1 + s2; // s3 is now "Hello friend."
```

In Objective-C, the standard way of concatenating two strings is:

```
NSString *s1 = @"Hello ";  
NSString *s2 = @"friend."  
NSString *s3 = [s1 stringByAppendingString: s2];
```

It's Not Very Modern

Objective-C does not allow functions (and methods) to be "first class citizens" and thus passed as data values, assigned to variables, or returned from (other) functions.

Enumerations in Objective-C are crude, defined as integers. Objective-C allows type overloading (meaning that the same method names can be used in different classes), but not function overloading (different types or mixtures of types with the same name) or

operator overloading (using the same operator but different types for operands). Custom operators cannot be defined. Generic programming, or writing code that can operate with input parameters of multiple types without implicit type conversion, is not in the language (although it can be added), often resulting in essentially the same functions having to be written multiple times.

Other languages allow operations on strings, arrays, and dictionaries to be done in the language itself. For Objective-C, however, this is done more clumsily by calling the library, which handles arrays (`NSArray` and `NSMutableArray`), dictionaries (`NSDictionary` and `NSMutableDictionary`), and strings (`NSString` and `NSMutableString`).

One commentator writing in *Ars Technica*, John Siracusa, suggested in 2005 that Objective-C's failure to keep up with other languages and their higher level of abstraction was a serious potential problem. By 2010 he was calling the language (and the associated Cocoa API, designed to work with a low-level language) a "ticking technology time bomb". He suggested that it was only Apple's success with its mobile products—and the constraints involved to make software on those products work that favored lower-level languages—that has distracted developers from this reality, giving Objective-C "new life".

Swift's Basic Philosophy

The four stated goals of Swift (Safety, clarity, modernity, and performance) reflect both an accurate analysis of the deficiencies of Objective-C and an fair assessment of the various developments that have taken place over the last three decades in programming language design—mostly object oriented languages like C++, Java, Python, and Ruby, but also more specialized languages such as the functional language Haskell.

The Swift project was initiated by Chris Lattner in July, 2010 and Lattner has specifically pointed to some of the languages mentioned above as having influenced Swift.

At least four ideas seem major philosophical underpinnings of the new language. These include: (1) Static typing; (2) Make it easier to be safe but don't be heavyhanded; (3) Passing functions and closures as data; and (4) Not everything is an object.

Static Typing.

The use of static typing, and the particular methods used in Swift to enforce a high level of type safety, are some of the biggest differences between Swift and Objective-C. Objective-C adopted the extreme approach of Smalltalk-80 of deferring whatever decisions could possibly be deferred from compile time to runtime, including, particularly, the type of a variable. This has been completely reversed in Swift, in which types are known to the compiler and will not change during runtime.

Make it Easier to be Safe but Don't be Heavyhanded

Swift is designed to encourage safe practices and to discourage bad ones. At the same time, the programmer is still in control. Often, engaging in a less safe practice requires the programming to take an explicit step that he or she must be aware of, but the language rarely prohibits it or makes it especially difficult.

Passing Functions and Closure Expressions as Data

Swift allows functions, methods, and closure expressions to be handled very much like data. This is common in other languages, and was added to Objective-C relatively recently for closures/blocks only. It makes many interactions with an API (especially those that might otherwise have used delegation) simpler and cleaner and allows some forms of functional programming.

Not Everything is an Object

Swift avoids some of the extreme aspects of languages like JavaScript that have chosen to make everything an object. Instead, Swift has extended the idea of a structure (from C), made it a sort of lighter weight and safer version of a class, and implemented many of the primitives of the language in structures, such as numbers, strings, and characters.

Safety

Swift has a number of capabilities that promote safety. These include type safety, an avoidance of pointers, value existence safety (the use of optionals), and a few lesser steps, including realtime checking for arithmetic overflow and underflow, array bounds checking, some added rigidity in if clause testing, and encouragement to use constants rather than variables.

Type Safety

Type safety is the most obvious and the most important aspect of Swift's focus on safety. Knowing the data type of a stored value is necessary for interpreting what would otherwise be just an incomprehensible sequence of bits that has been retrieved from memory. If the data type is not correct, it's a big problem: that sequence of bits will be interpreted as something it is not. This is a major source of programming errors.

Swift has a high degree of type safety, obtained by having relatively rigid rules about how types are handled. The type of a variable must be defined, and, once set, it cannot be changed (with a few narrow exceptions). Values cannot have their types changed implicitly, such as changing an integer value of 2 to a floating point value of 2.0 when the compiler sees that the integer will be added to another floating point value. This contrasts with most scripting languages, which are much more flexible about types.

Types can be explicitly defined by the programmer as follows:

```
var x: Int
```

The “`: Int`” after the variable name `x` defines the type of that variable. Types can also be set by type inference, such as:

```
var x = 5
```

In this case the compiler will assign a type based on an analysis of the literal value `5`, which it infers to be an integer, or `Int`. Types can be inferred based on literal values or on types of values received from an API. In typical programs, the vast majority of type definitions are done by the compiler using type inference. So Swift's rigidity about types is not very burdensome on the programmer.

Types can be explicitly converted by the programmer, such as in the following:

```
var x = 5  
var y = Double(x) + 3.4
```

If an attempt were made to just add x to 3.4 there would be a compiler error because x was inferred to be an integer. But the `Double` constructor creates a floating point type from the integer value of x, which can then be added to the value 3.4 (which is also a `Double`).

Types that are class instances in an inheritance hierarchy can also be modified to change the type of a variable containing it to one that is higher or lower in the hierarchy. This is known as *type casting*.

Another aspect of type safety is restricting arrays to holding elements of only a single type. Swift's adherence to the object-oriented principle of polymorphism is interpreted here as allowing types that have a common ancestor in an inheritance hierarchy to be effectively of the same type, and thus allowed to be stored in the array. The array in such a case is then marked as having the type of that ancestor.

Arrays that result from reading an array from an Objective-C API may also have heterogeneous types, with the Swift array marked as an `Any` or `AnyObject` type, Swift's version of Objective-C's `id` generic type. Defining an array with a broad type is something of a loophole that avoids the restriction, but the practice should be to avoid doing this when creating arrays, and to quickly convert any arrays read from an API to lower-level types as soon as possible.

Dictionaries in Swift have a similar restriction, with all keys in a dictionary required to be of the same type, and all values required to be of the same type (though the keys do not have to be of the same type as the values).

No Pointers

Objective-C and other C-like languages use *pointers*. A pointer contains a number that specifies the location of the address in heap memory where a block of memory for a particular object is stored. The syntax looks like this:

```
NSString *pointerToString = @"Hello"
```

There is a big problem and a little problem with pointers. The big problem is that if a pointer happens to be written to with the wrong address (which can happen with bugs in arithmetic, and especially if a programmer likes to play tricks with pointers), it can result in a large block of memory being overwritten with the wrong information.

The little problem with pointers is the asterisk, which indicates that the variable is a pointer. It's easy to leave the asterisk off, a small annoyance for the programmer.

Value Safety and Optionals

Another important aspect of safety is dealing with situations in which variables do not have actual values. This is a common source of programmer error. Code is written with the assumption that a variable will have a value. At some later time, the code is executed but the variable does not have a value and bad things happen. This can occur because the programmer makes a wrong assumption, with prototype code written fast that turns into production code, or because the programmer plans to fix it later and then forgets. Ensuring that this does not happen might be called *value safety*. A common solution is the use of what are known as *optionals*.

In Swift, a variable either has a value or it has a `nil`. Also in Swift, an ordinary variable, known as a nonoptional, is required to have a value, which is to say that it is not allowed to have the value `nil`. An attempt to store `nil` to a nonoptional variable will cause a runtime exception. This might be considered a first line of defense in the optionals strategy for safety.

To be allowed to not contain a value (that is, contain `nil`), a variable must be declared as an optional, which is done by appending a question mark to its type, such as `Int?`.

Normally, a variable that is an optional must *unwrap* its value. This step is the second line of defense in the optionals strategy. Unwrapping can be done by appending an exclamation point to the variable name, like `x!`. The trap is that an attempt to unwrap an optional that does *not* have a value will trigger a program exception. So a common thing to do is to test the value of an optional variable and unwrap it only if it has a value.

Swift has a number of additional capabilities to deal with optionals—ways of testing for a value and unwrapping that are a little safer or easier to use, and ways of indicating that a variable is trusted to not be `nil`.

Optionals help to avoid errors resulting from values that should be there but aren't, but programmers must do the work. Especially when converting, say, Objective-C code to Swift it is easy to look at a large number of variables that produce compiler errors because the API they were read from said they were optionals. Programmers will then pepper the code with "?" characters to shut up the compiler on this issue, then pepper the same code with "!" characters to stop the runtime errors because the optionals have not been unwrapped. Doing this in a thoughtless way is easy but circumvents the value of optionals.

Checking the Boundaries of Arrays and Overflow/Underflow of Values

When a program is executing and an array is accessed, Swift will check the index used to access that array to see if it is within the bounds of the array and an exception will result if it is not. Similarly, when an arithmetic calculation is done a check will be made to see if the result is larger (overflow) than the type of the variable where it will be stored allows, or smaller (underflow) than allowed. An overflow or underflow will trigger an exception. This behavior can be turned off if desired with compiler flags to improve performance.

Booleans and Assignments in If Clauses

In Objective-C and many C-like languages, an if clause can test an arithmetic or logical expression that contains arithmetic values. A result of 0 is interpreted as false, while a result of anything nonzero is interpreted as true. Values of YES or true also count as true. Swift is much more rigid. Only logical expressions, with Swift Boolean types with values of either true or false, are allowed.

It is a common practice to put assignment statements within if clauses, such as:

```
if ( x = y - 4 ) print("do something");
```

If the resulting assignment is 0, it counts as false, while if the result is nonzero, it counts as true (and the statement is executed). This makes a language vulnerable to a programmer who intends to enter a conventional "==" operator but accidentally leaves off one of the equal signs.

Encouragement for Using Constants

In Swift, constants have a syntax similar to variables and are treated similarly. A variable is declared as follows:

```
var x = 5
```

A constant is declared like this:

```
let y = 8
```

The only real difference between a constant and a variable is that a constant can have a value stored in it only once.

Using a similar syntax makes constants easier to use (and not ugly, like X EQU 5). More importantly, it allows programmers to realize that they can often use a constant when they might normally expect to use a variable. This is safer, and can allow better optimization by the compiler.

Overriding Inherited Methods

Swift has conventional classes, objects, and inheritance. As in Objective-C, a method that has been inherited can be replaced for that class, or *overridden*, by providing code in a method in the class definition with the same name (and input parameter and return value types). Unlike in Objective-C, the keyword `override` must be used in Swift. This is to prevent a programmer from accidentally overriding an existing method by creating what he or she thinks is a new method. If the keyword `override` is not used there will be a compiler error. If a programmer attempts to override a method that has not been inherited, there will also be a compiler error.

Access Control

Swift controls access to code based on both where the code is and on tags in the code that specify it to be public, internal, or private. Every app runs in a particular module that can consist of multiple .swift source files. Code in a particular source file has access to all the code in that same source file. Code in a particular source file has access to code in other source files in the same module only if the particular entities in the code have been tagged to have an access level of public or internal. Code in a particular source file has access to code that has been imported only if that code has an access level of public. If code is not tagged it is assumed to be internal.

Clarity Helps Safety

The next section describes what has been done in Swift to make programs have better clarity. Although intended mainly to make programming easier and more fun, clarity also helps avoid errors.

Clarity

Code that is clear and clean helps make programming pleasant and fun.

Simple File Structure with No Crust

Gone are the two header (.h) and implementation (.m) files for every class that you would see in Objective-C. You don't need to import header files. Swift figures it out. You can have a file (.swift) for every class if you like, or put as much as you like in a single file. You'll likely just need a single import file for the framework packages that you'll be using, often just:

```
import UIKit
```

And no messing around trying to figure out whether the framework names should be in angle brackets or quotes.

You won't need a lot of crust to define properties and list signatures of methods that you use. And you won't see _variable names with underscores.

Semicolons Not Necessary at the End of a Line

In Swift, both of the following statements are valid:

```
var x = 5  
var x = 5;
```

Semicolons are still necessary to separate statements if there is more than one statement on a single line:

```
var x = 5; y = 7
```

Even in this case a semicolon is still not required at the end of a line.

Parentheses Not Required for Conditional Statements

Conditional expressions statements in C-like languages usually look like the following:

```
if( x == 5) {  
    printf("x is equal to five")  
}
```

In Swift, you can leave out the parentheses:

```
if x == 5 {  
    printf("x is equal to five")  
}
```

Parentheses are still *allowed*, and in the case of complex expressions might be desirable, either to group things to make them more readable or give some parts of the expression precedence over others:

```
if x == 7 && y ==9 || (a != "basic" || z == "hey") {  
    print("The parentheses make this expression easier to  
    read. ")  
}
```

Nested Comments / */*

Perhaps you have written code in a C-like language, putting comments in like this:

```
x = 5;  
/* This is a comment, using the slash asterisk style to  
allow  
more than one line for a given comment, in this case  
three lines in total */  
y = 7
```

And then, later, you wanted to comment out the *all* the lines, including the assignments to x and y, to see if it makes a difference.

What you want to do is this:

```
/*
x = 5;
/* This is a comment, using the slash asterisk style to
allow
more than one line for a given comment, in this case
three lines in total */
y = 7
*/
```

You want to nest your comments using the /* */ comment style. This is not allowed in C, but it *is* allowed in Swift. This is a small thing, but it tells you something about the willingness of the designers of Swift to worry about the smallest details.

Closure Expressions

Swift has *closure expressions*, which are pieces of code that can be manipulated in very flexible ways.

A closure expression is Swift's version of an Objective-C block, but has a cleaner and more flexible syntax.

Closure expressions can be very simple. The expression to the right of the equals symbol in the second line in what follows is a closure expression:

```
var m = 0
var a = { print(m) }
m = 5
a() // Prints: 5
```

In that second line, the closure expression { print(m) } is assigned to the variable a. The closure expression itself is not executed.

In the fourth line, the a() syntax causes the closure expression to be executed. The interesting thing here is that what gets printed is 5, not 0. This is because the code in the closure is not executed until the a(), and after the value of the variable has been changed to 5.

(In other contexts, the value printed might be 0, if the closure expression saves the value of the variable at the time the closure expression is defined, but that does not happen if the closure is defined, as it is above, at the global level.)

The closure shown above is about the simplest possible closure. An example of one of the more complex forms of a closure is shown below:

```
var a = { (a: Float, b:Float) -> Float in
  var c = a * b
  return c
}

print(a(2.0, 3.0)) // Prints: 6.0
```

This is a closure that is assigned to the variable a. It accepts two floating point input values, multiplies them together, and provides the result as a return value. (The “->”, known as a *return arrow*, is part of the syntax of a more complex closure. The type to its right indicates the type of the return value. The last statement does the same thing as the a() in the first closure example, but in this case the closure code accepts the two input parameters. This prints the value 6.0. This variation works like a function, and in fact the main difference (aside from some syntax differences) is that the closure does not have a name.

Closures can have many variations of syntax that range between the simple version that has only a pair of braces and the more complex version that has input parameter names and types, a return arrow and return type, the keyword in, the braces and a return keyword. In general, the most minimal form of the syntax that can be used given what needs to be in the closure expression is what is used.

Closures are often used to pass code as an argument in an API call that is intended to be executed when an API performs some action and initiates a callback. This is a common alternative to the use of a delegate method, because it allows the code to be placed in the

call to the API rather than buried in a delegate method. This provides cleaner, more readable code.

Strings

One of the more antiquated statements in Objective-C is the standard way to concatenate strings:

```
NSString *a = @“This is rather “;
NSString *b = [a stringByAppendingString: @“absurd”]
```

This is replaced in Swift by:

```
var a = “This is rather “
var b = a + “absurd”
```

Earlier, in discussing safety, I noted that pointers were discarded because of their lack of safety and because a programmer could forget the required asterisk.

Now, Swift allows us to use just a pair of double quotes rather than an at-sign and double quotes for a string, cleaning that up.

And, finally, we’ve gotten rid of the standard Objective-C way of concatenating two strings. (Most programmers found another way to do this.)

Matt Thompson ([NSHipster.com](#)) is skeptical that using an “addition” operator to mean concatenation makes much real sense. But it beats `stringByAppendingString` for brevity.

Appending to the End of a String or Array

A variation on the above is simply appending a string, or elements of an array, to an existing string or array. Swift not only allows the addition operator, but the *addition assignment operator*.

Thus:

```
var a = "This is rather "
var a += "absurd"
// a is now "This is rather absurd"
```

And for arrays:

```
var cities = ["San Francisco", "Oakland"]
cities += ["Monterey", "Los Angeles"]
// cities is now "[San Francisco", "Oakland", "Monterey",
"Los Angeles"]

airports = ["San Francisco", "Oakland", "San Jose",
"Stockton"]
```

Arrays can be very nicely addressed with subscripts, as is common in other languages:

```
airports[2] = "Monterey"
```

(Some of this has been added fairly recently to Objective-C.)

A simple syntax for ranges has been defined:

```
airports[1...2] = ["Monterey", "San Luis Obispo"]
```

The [1...2] notation indicates the elements in the array accessed by the subscripts 1 through 2. An alternative [1.. <3] notation means the same thing, subscripts 1 through 2. The names for these aren't very easy to remember. [1...2] is called a "full closed" range. [1.. <3] is called a "half open" range.

Tuples

Tuples are a simple way of handling multiple values that are of different types. They are

most obviously useful when you want to return multiple values from a function. Functions are designed to allow multiple input values, but only one output value. Putting such values in an array seems like overkill, and in Swift all of the values in an array are supposed to be of one type.

```
func getErrorCodeAndMessage() -> (Int, String) {  
    return (345, "Not Found")  
}  
  
var error = getErrorCodeAndMessage()  
print(error.0) // Prints: 345  
print(error.1) // Prints: Not Found
```

Defining and Using a Class

In Objective-C, defining a class is a bit of a mess. First, in a header (.h) file, you need:

```
fruit.h file:  
  
#import <Foundation/Foundation.h>  
  
@interface Fruit : NSObject {  
}  
  
@property (nonatomic, strong) NSString *color;  
- (void) printColor;  
  
@end
```

Then, in an implementation (.m) file, you need:

```
fruit.m file:  
  
@implementation  
- (void) printColor {  
    NSLog(@"The color of the fruit is %@", self.color);  
}  
  
@end
```

To create an object, set a property, and execute a method, you need:

```
Fruit aParticularFruit = [[Fruit alloc] init];
aParticularFruit.color = "green";
[aParticularFruit printColor];
```

Swift does it much simpler. To create the class, you just need one file:

fruit.swift file:

```
import Foundation
class Fruit {
    var color = "red"
    var lengthInInches = 3.5
    func printColor() {
        print("The color of the fruit is \(self.color)")
    }
}
```

To create an object, set a property, and execute a method, you need:

```
let aParticularFruit = Fruit()
aParticularFruit.color = "green"
aParticularFruit.printColor()
// Prints: The color of the fruit is green
```

Requiring Braces Around Conditional Expressions

Most C-like languages have conditional statements like the following:

```
if(x == 5) print("I am a single statement in a
conditional"); // Not valid in Swift

if(x == 5) {
```

```
    print ("I am the first statement of two in a  
conditional");  
  
    print ("I am the second statement of two in a  
conditional");  
}
```

In C and most similar languages the first statement is valid. It is not valid in Swift. Consistently requiring braces reduces errors, particularly when statements are moved in and out of the braces. (This isn't really a safety issue because the compiler can catch it.)

Changes to the Switch Statement

In C and most C-like language the Switch statement looks like this:

```
switch(a) {  
case 10: {  
print("value is 10")  
break;  
}  
case 11: {  
print("value is 11")  
break;  
}  
case 12: {  
print("value is 12")  
break;  
}  
}
```

In Swift the same switch statement would look like this:

```
switch(a) {  
case 10: {
```

```
print("value is 10")
}

case 11: {
print("value is 11")
}

case 12: {
print("value is 12")
}

default: {
print("value not matched")
}
}
```

The switch statement works a little differently in Swift. In C, if a switch value matches the value associated with a case, the code in that case is executed, after which control will fall through to the next case and potentially execute additional code. To prevent this, a break statement is normally added as the last statement in a case.

In Swift, if there is a match, the code for a case is executed and then the switch statement is done. There is no need for break statements. (If you want the fall through behavior you can get it with a fallthrough statement.)

Modernity

Much of Swift's claim to be a "modern" language is fulfilled by what it has done to improve safety. Modern languages typically do not have pointers. They often have something like optionals. Checking to see if a reference to an array is out of bounds, and not allowing assignments within if clauses are similarly modern, as are most of the other minor things that Swift does to promote safety.

Most modern languages, however, do not have typing as rigid as does Swift, and in this respect Swift is deliberately non-modern, if "modern" is about the convenience (and lack of safety) of implicit type conversion. Swift's use of type inference to make its strict typing easier for programmers to deal with, though, is very modern.

Modern languages also have the clean syntax like that shown in the previous section. And they have the closures that have been described.

Modern languages do have strings, arrays, and dictionaries implemented directly in the language and allow elements to be accessed with subscripts.

Swift has modern capabilities beyond these. They include (1) The ability to pass functions and closures as if they were data; (2) Adding many modern capabilities to classes; (3) Some additional string capabilities; (4) The ability to iterate through a string, array, or dictionary; (5) The addition of structures; (6) The addition of enumerations (7) The ability to overload operators and functions; (8) Defining custom operators; and (9) The use of generic programming.

Functions and Closure Expressions as First Class Citizens

An important capability is that of treating functions and closure expressions as so-called "first class citizens". The term comes from a comment made in the 1970s about the programming language Pascal. According to this comment, it would be helpful if things like functions could be used in the same way as data, meaning that they could be passed around as arguments in calls to other functions, be passed from a function as a return value, and be assigned to a variable. Why can't a function have the same rights as a mere piece of data? Why can't a function be a "first class citizen"?

Swift defines both functions (and thus methods, which are just functions associated with classes and other instance-creating types) and closure expressions as first class citizens. It thus allows them to be passed as input arguments to other functions and closures, to be

returned from functions and closures, and to be assigned to a variable.

The following shows an example of how a closure can be assigned to a variable:

```
var m = 7
var a = { print(m) }
a() // Prints: 7
```

This ability to pass functions as if they were data allows Swift to be used as a functional programming language. It makes for clearer code, as discussed earlier, and makes Swift more powerful.

Functional Programming

Functional programming is a very different programming paradigm from either the typical object-oriented languages of Objective-C, Java, C++, and Ruby, and procedural languages like Pascal.

All of these languages are based on the idea of statements that perform a computation by steps and write the result to memory after every step, known as *imperative computing*. This is about as basic an idea to today's programmers as the idea of water is to a fish—so much a part of the environment that he or she might not even be very specifically aware of it.

Functional programming is completely different. Functional programming attempts to specifically *not* write results to memory. To use the term commonly used, functional programming is *stateless*.

Swift is not as easy to use for functional programming as a specialized functional programming language like Haskell. However, it does have the basic capability. Although functional programming cannot be used for every problem, where it can it can result in highly reliable, easily debuggable code that is usually much shorter than comparable non-functional code.

Adding Modern Capabilities to Classes

In addition to cleaning up the way classes are defined, Swift also adds several new, modern capabilities to classes:

Computed Properties. These are accessed like properties of an object but their values are calculated at the time they are accessed. This is useful for values that rely on real-time information (e.g., time of day), or where it is desirable to maintain only one stored property for certain information and compute more than one value based on this information (e.g., use a stored property for Fahrenheit when storing temperature but a computed property for Celsius). A computed property can also be “written to” with the result the execution of setter code that often writes values to other properties.

Lazy Properties. Lazy properties wait to be initialized until they are accessed. They are used in cases where it takes some effort to initialize them (e.g., accessing a server) and the property may not always be used (and thus the effort of initializing it wasted) or when it is better for a delay to occur at time of first access rather than initialization.

Flexible Initializers. Swift has a flexible (and rather complex) set of ways to initializing an object. A special method-like syntax using the keyword init is used, rather than the initialization methods like those of Objective-C. Defining an initializer isn’t actually required if properties have an initial value defined. A more complex initializer can allow properties to be set with arguments when an object is created. So-called convenience initializers can also be defined that can be called by other initializers to provide additional flexibility.

Subscripts. Classes can be defined so that data associated with an instance can be accessed with a subscript when that way of accessing data makes sense. For example, a class PeoplesTopTenMovies might be defined, and an instance georgesMovies created, referring to the favorite movies for someone named George. The title of George’s 3rd most favorite movie might then be accessed with georgesMovies[3]. Storing to and retrieving from the instance is defined by getter and setter functions in the class as part of the use of the keyword subscript in the class definition.

String Capabilities

As indicated earlier, strings in Swift can be concatenated by the simple addition (“+”) operator and appended to by the addition assignment operator (“+=”).

Strings in Swift comply with the Unicode standard, and can represent characters like Asian logographs and emojis:

```
var s = "的"  
var hamburger = "🍔"
```

The Unicode characters can even be used in variable names:

```
var 的 = 37.6
```

Iterating Through An Array, String, or Dictionary

In Swift it is possible to directly iterate through an array, string, or dictionary, without having to worry about indexes. Thus:

```
var s = "This is a 🍔 hamburger"  
for ch in s {  
    print(ch)  
}
```

This is particularly useful in the case shown, with the hamburger emoji: an iteration is done for each visible Unicode character, not for each internal character. (The hamburger emoji takes 4 internal characters to represent.)

Structures

Structures in Swift are used when you have data that has multiple instances with each having the same properties. They are in a sense lighter weight, slightly safer versions of classes. They have instances and properties and can have methods associated with them, but cannot perform inheritance. They are value types, meaning that they are stored in the stack, not the heap, and memory allocation and deallocation is not performed.

Enumerations

An enumeration in Swift is a custom data type that can only have a value that is one of a set of values specified in the definition of the enumeration.

A classic example of an enumeration is that of the day of the week. The value must be one of the following: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday. A Swift enumeration defining a `DayOfTheWeek` type has the following syntax:

```
enum DayOfTheWeek {  
    case Monday  
    case Tuesday  
    case Wednesday  
    case Thursday  
    case Friday  
    case Saturday  
    case Sunday  
}
```

A variable declared with a type of `DayOfTheWeek` that had a value stored in it would be like this:

```
var today: DayOfTheWeek = DayOfTheWeek.Friday
```

The value is actually that of `Friday`, not the string “Friday” or an integer value representing Friday.

Enumerations in Swift are, like structures, a lightweight, slightly safer version of a class. They are safer than the Objective-C version because the latter just defines each possible enumeration as a particular integer value, and it is possible for the program to write an invalid nonsensical integer value into the variable representing the enumeration. Defining an enumeration this way also makes the code clearer. Finally, enumerations make Swift more modern and powerful because the enumerations can hold associated values (similar to properties) and can have methods and instances in the same way as structures.

“Instance Creating Types”—A Note About Terminology

The addition of structures and enumerations helps make the language more powerful and modern, but at some cost of confusion. Because they have much of the functionality of classes, lots of things in Apple’s documentation have to refer to language types that include structures and enumerations as well as classes. Apple often refers to the three

types—classes, structures, and enumerations—as a group using the word “types”, which is very confusing, because it seems to include *any* type. I suggest calling these *instance creating types*, since this uniquely identifies them.

Overloading Operators

Swift allows overloading operators, as part of its adherence to the polymorphism principle of object oriented programming.

In operator overloading, an operator has different behavior depending upon the data types of its arguments.

Thus, in Swift the addition (“+”) operator allows you to add two integers:

```
var a = 5 + 7      // result is 12
```

However, if the types involved are strings, the result is different:

```
var a = "5" + "7" // result is "57"
```

If this wasn’t already in the language, it would be very easy to add. An operator is in most respects really a function, and it can be redefined:

```
func +(left: String, right: String -> String) {
    var concat = left.append(right)
    return concat
}
```

This redefinition only applies to the context in which the operator “+” has a String value on the left and another String value on the right as operands.

Defining Custom Operators

Defining custom operators is done in a similar manner as overloading an operator.

Say we think that the name of the function normally used to calculate a square root is ugly:

```
var y = sqrt(5.7)
```

We'd prefer to use the Unicode square root symbol $\sqrt{}$. ($\sqrt{}$ can be entered on a Macintosh by pressing the V key with the option key held down.)

We first declare the operator:

```
prefix operator √ { }
```

We then define the function named $\sqrt{}$:

```
prefix func √(x:Double) -> Double {  
    return sqrt(x)  
}
```

And we can now use the following to calculate a square root:

```
var y = √(5.7)
```

Overloading Functions

Swift allows functions to have multiple definitions, with the same name but different types, or different combinations of types. At compile time, the correct version of the function will be identified based on the type or combination of types in the calling statement. This reflects the principle of polymorphism in object-oriented programming that Swift subscribes to.

Generic Programming

Swift also has the modern capability of generic programming, or generics.

It is often the case that functions have to be written multiple times, with a different version for each type (or set of types) that is used in its input parameters, code, and return values. Generic programming allows a single function to be written by a programmer that covers the different types. This function is then compiled automatically in the different forms for different types as required. Because the actual compiled code uses only particular types, it still has a high degree of safety, as much as if the programmer had written different pieces of code for the different required types.

An example of a function written generically is as follows:

```
func addTwoNumbersWithFirst<T> (first:T, second:T) -> T
{
    var c = first + second
    return c
}
```

The symbol T, shown between the angle brackets just after the function name, is the standard way of referring to a parameter type when only one parameter is involved. When that type is to be defined, the symbol T is used.

Performance

As part of Apple's goal of creating an "industrial quality systems programming language", Swift is claimed to have high performance. This is usually interpreted as just fast execution time, although considering the intimacy of a mobile app and Apple's reasons for choosing reference counting over automatic garbage collection, it is also fair to consider not just peak execution time but also the smoothness of processing.

Assessing the actual speed of execution of a programming language compared to another language is quite difficult. Obviously, it is easy to write the same essential program in both languages, run them, and time the results. But many issues come up. Is this particular task representative of the kind of processing that will be done by the code in a real app? Would this task even normally be done by code or would some module that is part of the API be called, better optimized and perhaps written in a different language?

Computers are fast enough, and the way interactive applications use processing power so sporadic (on average not very much and occasionally with a very high demand), that the programming world isn't necessarily that enamored of brute processing power. Scripting languages like Python are popular because whether processing is ten times slower than it might otherwise be is often less important than how easy the language is to use, how much time it takes to write the program, and how big the source program ends up being.

The normal practice is to simply write the code, and then, if there is a problem with an app running slow, to tweak the (some claim) one to five percent of the lines of code that are typically performing a time-sensitive function.

Still, Apple has set a high bar of claiming Swift to be "industrial quality", implying that it has the speed of low-level compiled languages like C and C++.

There is some benchmark data available comparing Swift, Objective-C, and C. In an August, 2014 test with the beta version of Swift then available, Jesse Squires found Swift to be surprisingly fast, ranging from six times faster to nearly eighteen times faster for different kinds of sorting. When compiler flags were set to not check for such things as overflows, Swift was even faster.

This test was later criticized because of the way that Objective-C handles arrays. While Swift stores actual integers in arrays, Objective-C doesn't store integers directly in arrays —it wraps them up in an NSNumber object. To sort such an array, you have to constantly unwrap and wrap the objects to get at the integers. The example shows in part why speed comparisons across languages are difficult. This test obviously exaggerates the difference

between Swift and Objective-C. Yet wrapping up primitive values as objects is common in Objective-C, and along with dynamic dispatching of methods (see below), is a serious reason for why Swift is, or at least should be, faster overall than Objective-C.

We don't, at this stage, have seriously credible evidence comparing the speed of Swift to other languages. Interestingly, the principal person who designed Swift, Chris Lattner, is an expert in compilers. But designing the language and getting the compiler and other tools to work at all has taken priority over optimizing the compiler. And so we will have to wait until the compiler is in a more mature state to get any reliable measurements.

However, we can get some idea by looking at how Swift does things. It is pretty clear that Swift will be faster than languages like Python and Ruby, because they are interpreted rather than compiled.

It is also clear that Swift will normally be faster than Objective-C, although, on average, probably not by a large amount. An Objective-C method is called by passing a message to an object that dynamically decides what method is to be executed. This is done by a program, part of the Objective-C runtime, known as `objc_msgSend`. Objective-C, based on Smalltalk's love of dynamic dispatch and "late binding of all things" (to quote Alan Kay, the principal designer of Smalltalk), uses dynamic dispatch. However, only a tiny proportion of typical Objective-C code actually needs it. And, although the dynamic dispatch code has been carefully designed to run fast—written partly in assembly code and using caching extensively when possible—it still significantly wastes processing cycles in the case of the vast majority of code that doesn't need the dynamic dispatch capability.

In contrast, Swift doesn't use dynamic dispatch at all, but simply looks up the methods in a fixed table that is produced by the compiler, which can create this because of Swift's static typing.

For various reasons, usually related to its static typing, the Swift compiler also has a greater capability than the Objective-C compiler to perform optimization, potentially eliminating code in circumstances in which it can determine that that code would not actually do anything productive. It can use memory more efficiently, in some cases allowing the use of stack memory instead of the less efficient heap memory. It can also use registers of the CPU more efficiently.

A particularly interesting case is a comparison between Swift and C. Surprisingly, Swift in some circumstances can actually be faster than C. C is required to allow multiple references to point to the same location in memory (known as *aliasing*). This makes it difficult for the C compiler to do certain kinds of optimization it might otherwise do. Swift doesn't have this requirement, and can thus in some circumstances produce code that runs

a little faster than C.

(This analysis of Swift's potential speed is based on Mike Ash's analysis in addition to my own knowledge.—See mikeash.com and his blog for July 4, 2014.)

Swift's performance also depends on how it is compiled. In some tests Swift has been shown to be many times slower than C when run normally, but nearly as fast as C, or even slightly faster, when the compiler has optimized it to run as fast as possible. However, such optimization may not be a good idea. This typically turns off checking such as overflow/underflow tests when doing arithmetic and array boundary checking. It also turns off compliance with certain ISO and IEEE standards for mathematical functions, running a risk that certain algorithms will not work properly,

With regard to Swift's performance in allowing a smooth presentation of the user interface, Swift's use of reference counting rather than automatic garbage collection indicates that Swift will be better than languages like Java running under Android. Reference counting, and automatic garbage collection, are two different schemes for managing memory and allowing its efficient reuse. Automatic garbage collection requires less work on the part of the programmer, but can cause user interfaces to freeze or "stutter" when the time comes for the system to reclaim unused memory. Reference counting, which is also used with Objective-C but no other major language environments, is far superior in allowing smooth processing. Automatic garbage collection can also significantly slow down processing speed under conditions in which memory is very constrained, a phenomenon that reference counting is far less subject to.

We can expect Swift to be far faster in terms of raw processing speed than languages like Python and Ruby, and slightly faster than Objective-C. Proof of this, however, will have to come later. We can also expect Swift to provide significantly smoother presentation of user interfaces.

Conclusions

The two questions I posed at the beginning of this paper were:

1. Will Apple achieve the goals it has stated for Swift? and
2. What does Swift mean for iOS app development?

Achieving Apple's Goals

Will Apple's goals be achieved? I believe that the answer is yes, with only a few caveats. I'll take up each goal first.

Safety. Swift is clearly far safer than Objective-C and scripted languages like JavaScript. This is primarily due to the use of static typing and of optionals. It is also due to a syntax that is simple, clear, and well-designed. The use of type inference allows a high degree of type safety while putting little burden on the programmer. Optionals do put a significant burden on the programmer, and it is quite possible that many programmers will find ways to avoid using them in the way they are most effective, so there is some uncertainty about their real impact.

Clarity. Swift is a very clear, well-designed language, with many developers saying that it is “a joy to use”, and a huge improvement on Objective-C’s ugliness and clutter.

Modernity. Swift’s closures and functions, including defining them as “first class citizens” that can be passed around as parameters and return values, are well defined. The ability to iterate through strings, arrays, and dictionaries, the addition of structures and enumerations, the ability to add new operators and overload operators and functions and to do generic programming are also substantial ways in which the new language is modern.

Performance. Swift’s level of raw processing performance has yet to be credibly demonstrated. But there is good reason to believe that this will be achieved, with Swift performing far faster than Python and Ruby, at least modestly faster than Objective-C, and perhaps even slightly faster than C in some circumstances.

An Enjoyable, Expressive Industrial-Quality Language. In my opinion, Apple has only partly achieved its goal of providing an “industrial-quality systems programming language that is as expressive and enjoyable as a scripting language.” It has (or will, with compiler improvements) produced an industrial quality language. It is capable of expressing the

programmer's intentions as well as any scripting language. Is it enjoyable? For much or even most of the code that will be written, yes. The problem comes when it is not. Swift is ultimately quite a complex language. If you think not, read the documentation on initializers and inheritance! It is probably the case that much of the complexity can be ignored by most programmers who are building simple apps. But a lot of it you can't. You can't write an app without dealing with the APIs, and you must understand optionals to deal with the API (and even to use a dictionary.). Optionals are a huge pain. They perhaps help avoid even more pain, but that pain is there. You must understand memory management enough to avoid retain cycles, which can be rather subtle. Every scripting language I am aware of runs in an environment that uses automatic garbage collection and does not require the programmer to intervene. If you write classes that inherit, you are going to have to call superclasses to make sure they get initialized. The defaults are probably set up so that you don't have to understand the complicated rules, but you may occasionally get in trouble. So it is very hard to say that Swift is as enjoyable to use as a scripting language overall. It just isn't, and can't be. Still, Apple deserves high marks for getting as close to their goal as could reasonably be expected.

Problems with the Compiler and Tools

Swift is not yet a production language for large-scale projects. Small and perhaps medium-scale apps have been successfully built and are in the App Store, built by developers with a certain persistence and willingness to work around problems. The compiler is slow, with delays apparently exponentially related to the amount of code and dependent upon which files and modules the code is packaged in. Error messages are not very informative, often at an extremely low level. Automatic bridging that is intended to make Swift work "seamlessly" with the iOS API doesn't always work. As discussed earlier, the compiler often produces poorly optimized code, including code with extraneous retain-release statements. The Xcode interactive development environment can be slow to respond with autocompletion and with flagging errors and turning error flags off when they have been fixed.

These issues may be largely solved when Swift 2 (and Xcode 7) is publicly released. Apple has a high level of skill with compilers and the task of making them work is a seemingly manageable problem. (Unlike Apple's ill-fated Copland operating system for the Mac that they eventually abandoned, being replaced with the NextStep OS that is now the foundation of the current OS.) Apple is extremely profitable and has the resources to finish it. I see no reason to believe that Swift will not eventually have the tools that support its already high-quality design. The worst-case scenario is probably just a further delay. The recent changes associated with Swift 2 may get Swift very close to a production language.

What Does Swift Mean for iOS App Development?

The reaction to Swift from the developer community has generally been very positive. Criticism has been minor. This is perhaps a pleasant surprise, given the programmer culture in which people have strong and often crazy opinions about everything, no matter how ill-informed.

Some of the negative reactions have been predictable: Enthusiasts of scripting languages like JavaScript think that not having implicit type conversion (automatically converting a data value's type when necessary) makes the language too inflexible. This is mostly a philosophical disagreement that has no solution. Some programmers feel that errors are inevitable and want their programs to run no matter what. Others want them to quickly fail in hopes of getting every last bug out.

Others think that Apple should have developed a general language that could serve not only iOS but be cross-platform and serve Android and other operating systems. (This is actually possible with Swift but unlikely.)

Most existing Objective-C developers will likely switch to Swift as soon as the compiler and tools are stable and run fast and as managers recognize that the language is ready. Some major projects that were started in Swift reverted to Objective-C when compiler and other issues developed. Most of these projects are looking to migrate to Swift as soon as they reasonably can.

Some programmers will not be so quick to switch. Automatic Reference Counting has solved the most significant annoyance with Objective-C (having to allocate and deallocate memory manually) and its source of the most critical errors (memory leaks when memory not deallocated properly and crashes when deallocation is done unnecessarily). Many programmers will have worked with Objective-C so long that they have adapted to its quirks, are blind to its confusing aspects, and can work productively with it (although they surely spend a lot of time debugging.) There are other programmers who like doing tricky (and arguably unsafe) things with low level pointers. Some will require more convincing data about the performance of Swift. Some have already announced their attitude: “You can take my Objective-C only when you pry it from my cold, dead hands”. In other words, when Apple decides to no longer allow apps to be written in Objective-C.

Apple is quick to deprecate APIs that it no longer considers those it wants developers to use, and is aggressive about pushing users and developers to the latest versions of iOS and to relatively recent hardware. But Apple clearly needs Objective-C to maintain the legacy APIs, and parts of the iOS and OS X operating systems, that have been written in it. And there is a large base of app code that has been written in Objective-C. Apple is unlikely to be very quickly aggressive about getting developers to switch to Swift. But Apple *has*

made it easy to mix Swift and Objective-C code. It is quite possible that Apple will slowly nudge developers in the direction of using Swift, without prohibiting Objective-C. This might include requiring apps being submitted to the App Store to have their root controller written in Swift but allow calling on Objective-C code. It could also involve developing new, attractive APIs for Swift that do not work in Objective-C.

The biggest question is what will happen with Cocoa Touch. There is a belief by some that Swift will motivate web developers to move to iOS native app development because the language's similarity to a scripting language. This isn't realistic for a number of reasons. The complexity of Swift will still be something of a barrier, although nothing like the steep learning curve Objective-C has long been for programmers, many of whom never got the hang of it and gave up. But the real problem is the complexity of the Cocoa Touch APIs used with iOS. These are far more complex than the Swift language, and many are arguably just as unpleasant to deal with as the worst aspects of Objective-C. As Matt Thompson ([NSHipster.com](#)) has indicated, many things could be done with Cocoa Touch that would better fit the Swift language than the current APIs.

The introduction of Protocol Oriented Programming, in which protocols can be extended with default methods and thus work something like (multiple) inheritance without the headaches of inheritance, is something that could make a big difference. Apple has made strong claims about the revolutionary nature of protocol oriented programming, and if the potential of the approach turns out to be realized, Apple might end up writing a Cocoa Touch-like library for Swift based on protocols rather than classes. This, plus a general simplification of the APIs, could help the language and platform move further toward being more accessible for programmers with scripting language backgrounds.

At the 2015 World Wide Developers Conference, Apple announced that it would make Swift open-source. This includes making the source code for the compiler public, and supporting the language not only on iOS and OS X, but also on Linux. Developers will be encouraged to contribute code that implements new features to the language. It is quite possible that Swift and the Swift compiler becoming open source will lead to the development of a server-side environment (analogous to Rails for Ruby or Django for Python.) Indeed, it might be in Apple's interest to fund such an effort. Safety, a clean design, and high performance might be sufficient incentives to encourage this use. If it got going, new programmers would have good reason to learn Swift because it could be used for both apps and web sites. The availability of programmers skilled in the language would then be a further reason for its use. It's probably unlikely for many iOS developers to also develop web server code, but it is quite plausible for Swift to become a popular language for learning programming, leading to a career path of using Swift on servers and Linux first and then to iOS.

Matt Thompson has suggested that Apple could put a Swift capability in the Safari browser as a first step toward potentially replacing JavaScript as a web browser language.

It would be in Apple's interest to make Swift work in a broad range of environments and thus increase the number of skilled Swift programmers. And the fact that Swift's claim to be easy to use, safe, and fast is fundamentally true does make it a language that could be applied far more generally than just for iOS apps.

Apple has produced an excellent language with Swift. There is every reason to believe that Apple will develop tools that will make Swift an effective large-scale production language. Swift's advantages over Objective-C will induce the overwhelming majority of developers to use Swift for building apps, probably within the next couple of years for new projects. Revising the Cocoa Touch APIs to make better use of Swift (perhaps based on protocols) and to generally be a cleaner, more productive API for programmers should be a next step for Apple. And Swift could become a language that is much more broadly used than just on Macs and iOS devices.

B Playgrounds

A *Playground* is a part of Xcode (the interactive development environment for Swift and Objective-C) that presents a window and allows you to enter a sequence of Swift statements in a column at the far left of the window. The statements are executed and the results displayed in a second column to the right, known as the “results sidebar”.

To run a Playground, you need to have Xcode installed on a Macintosh. The latest public version of Xcode can be obtained, for free, from the Apple Mac App Store. (It is fairly large, about 2.6 Gigabytes for Xcode 6.4.)

Once Xcode is installed, when it starts up it will display a window that will ask you if you want to do various things. One option is to run a Playground.

(This startup window is usually turned off by users, and if it has been turned off you can start a Playground by going to the File menu of Xcode at the top of your Macintosh screen and selecting New, then Playground.)

When you indicate that you want a Playground, you’ll get a popup window that asks for a name (the default it provides is MyPlayground, which is fine) and asks you to choose your desired platform, iOS or OS X. Then click on Next.

You’ll get another popup window displaying a part of your file system to indicate where the Playground will be saved. The likely default, Desktop, is fine. Click on Create.

A window like that shown below will then be displayed:



MyPlayground.playground

//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"

"Hello, playground"

It will already be populated with two lines of code: an import statement, importing UIKit, which provides access to some useful libraries, and an example statement declaring a variable and setting a string to it, “Hello, playground”.

This allows you to see the results that Playground displays in the right-hand column. Shown is the result of evaluating the part of the statement that is to the right of the assignment operator.

In the figure below I have entered some additional Swift statements, and you can see more of Playground’s evaluation of results. Simple assignments to variables are evaluated and displayed immediately. The definition of a class, however, produces no results. If an instance is created of that class, however, the values of any properties defined in the class will be evaluated (e.g., the value of the property `skinColor` is shown as “red”).



MyPlayground.pl

MyPlayground.playground

//: Playground - noun: a place
where people can play

```
import UIKit
```

```
var str = "Hello, playground"
```

```
"Hello, playground"
```

```
var a = 55
```

```
55
```

```
var b = 2.3
```

```
2.3
```

```
var c: Double = 5
```

```
5
```

```
var d: Double = 2.3
```

```
2.3
```

```
var e = 2.3 + 4
```

```
6.3
```

```
class Apple {
```

```
    var skinColor = "red"
```

```
}
```

```
let anApple = Apple()
```

```
(skinColor "red")
```

```
println(anApple.skinColor)
```

```
"red"
```

A third column, known as the “Assistant Editor” view, can be added if desired. This view shows what is normally seen in the console log.

Any print statements will have their output displayed in the third column, as console output, as shown in the figure below.



MyPlayground.playground

88 | < > | MyPlayground.playground

88 | < > | MyPlayground....rou

//: Playground - noun: a
place where people can
play

```
import UIKit
```

```
var str = "Hello,  
playground"           "Hello, playground"
```

```
var a = 55             55
```

```
var b = 2.3            2.3
```

```
var c: Double = 5       5
```

```
var d: Double = 2.3      2.3
```

```
var e = 2.3 + 4        6.3
```

```
class Apple {  
    var skinColor = "red"
```

```
}
```

```
let anApple = Apple()      (skinColor 'red')
```

```
println(anApple.skinColor)  'red'
```

x

Console Output

red

To display this third column, go to the Xcode menu at the top of your Mac screen and select the View menu, then Standard Editor, then Show Standard Editor. You may have to adjust the relative widths of the columns by dragging them to get a reasonable display.

Another Playground shows another interesting capability, known as the “value history” option:



playground



Playground - now a place
where people can play

import

var getHello = "Hello, playground!"

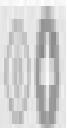
Hello, playground!

var e = "

"Hello, world!"

e

(8 times)



A while loop is defined, with the variable `e` starting at 0 and then being incremented, with the loop ending when the variable `e` is 8.

The figure will indicate that execution passed through the loop 8 times. If you move the mouse cursor over the line with the `e++`, it will become shaded in gray as shown, and a circle with a “+” character will appear at the far right.

If you click on the “+” character, you will see the following:

```
var e = 0  
while e < 8 {
```

e=e+1

0

(8 times)

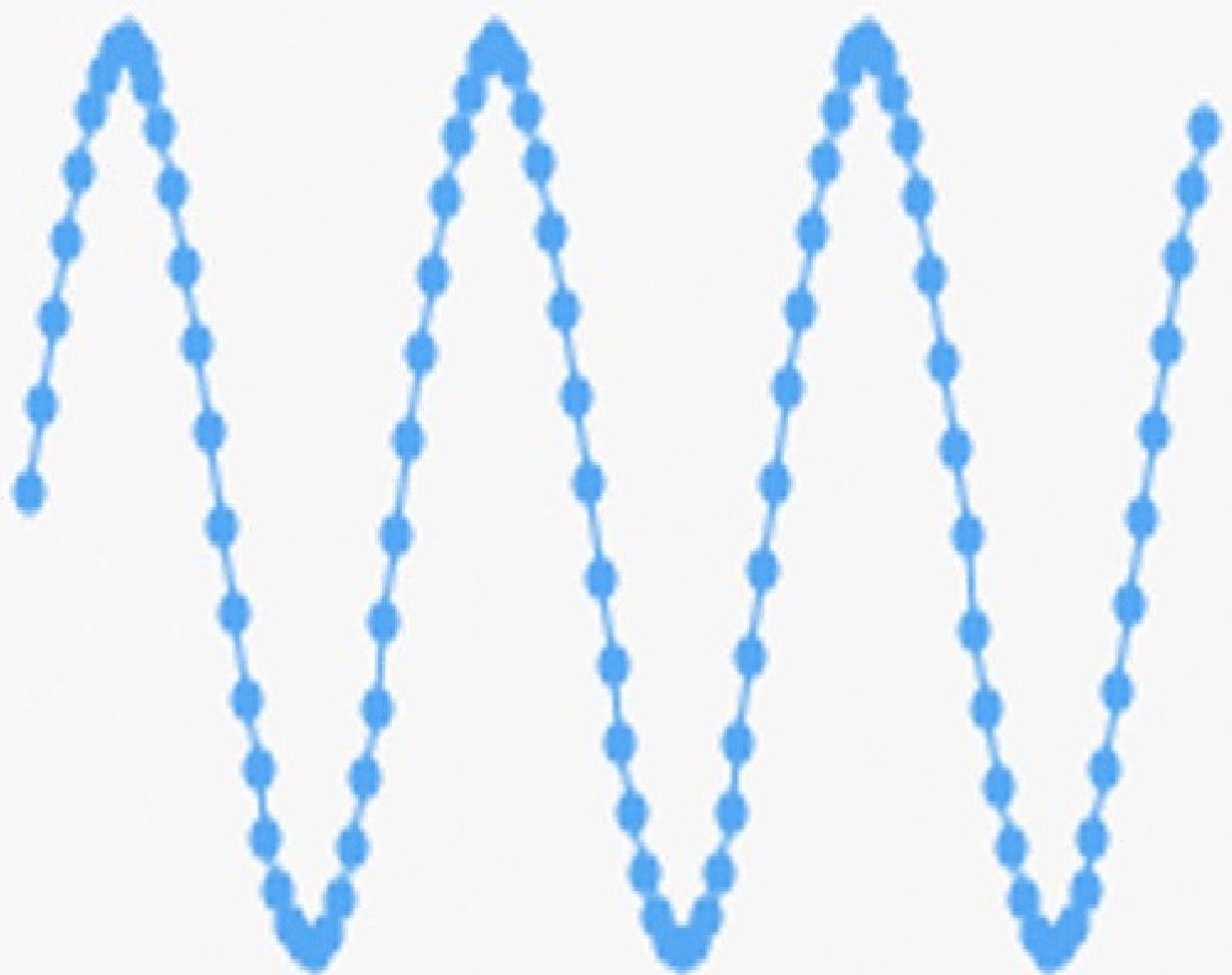


}

This is known as the “value history” option, and graphically displays the value of the variable over time. (The x coordinate is time, while the y coordinate shows the value.)

This works for any expression that is evaluated while being repeated. Thus, the expression shown in the code in the figure below generates the display of a sine wave:

```
for var i = 0; i < 100; i++ {  
    sin(0.2*Double(i))
```



```
}
```


Rich Text Markup in Comments

This capability is sometimes described as a feature of Swift 2 (and Xcode 7), but is available in Xcode version 6.3 and later.

Comments can be marked up with rich text in a Playground. This isn't shown in the normal display of a Playground. The Editor menu in Xcode has an item (at the bottom) that can be toggled between "Show Raw Markup" and "Show Rendered Markup". The raw markup is just the commands, such as:

```
//: Playground - noun: a place where people can play
```

The rendered markup version looks like this:

Playground - now it's a place

where people can play

To make use of markup, a colon character (“:”) is added to the characters indicating the beginning of a comment. This is done when the comment is a single line comment using two forward slashes:

```
//: This is a single-line comment
```

which is rendered as:



The same syntax (not /* */) is used for a multi-line comment:

```
//: This is a multi-line  
//: comment and is displayed  
//: as a block comment
```



The Playground markup can also display three levels of headers.

```
//: # This is a high level header
```

which is rendered like this:

This is a first level
header

This is a second-level header:

```
//: ## This is a second-level header
```

which is rendered like this:

This is a second level

header

This is a third-level header:

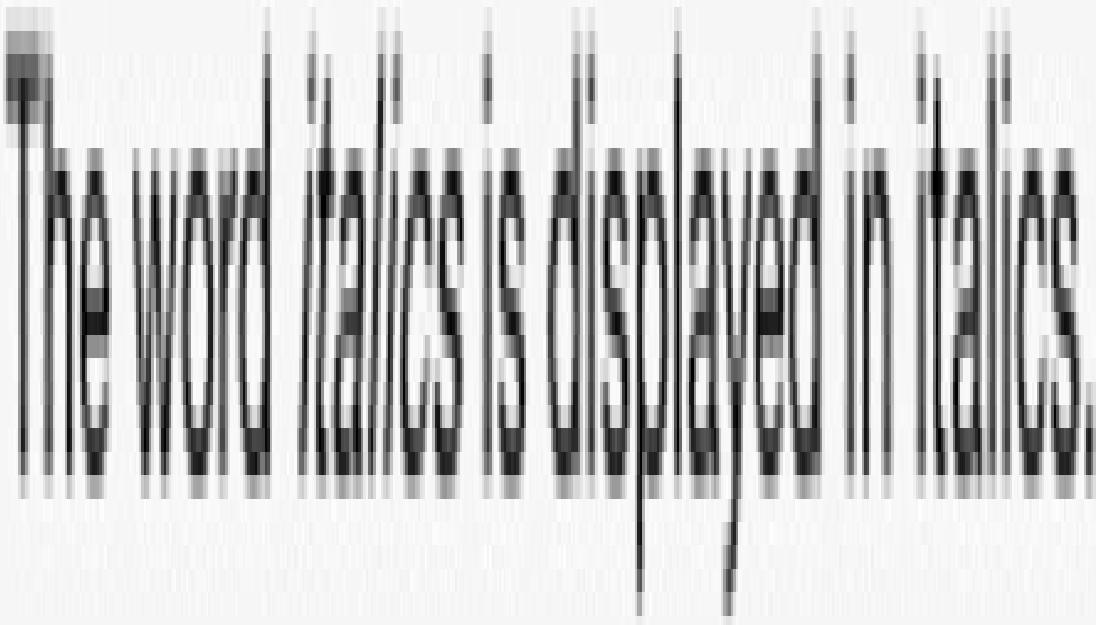
```
//: ### This is a third-level header
```

which is rendered like this:

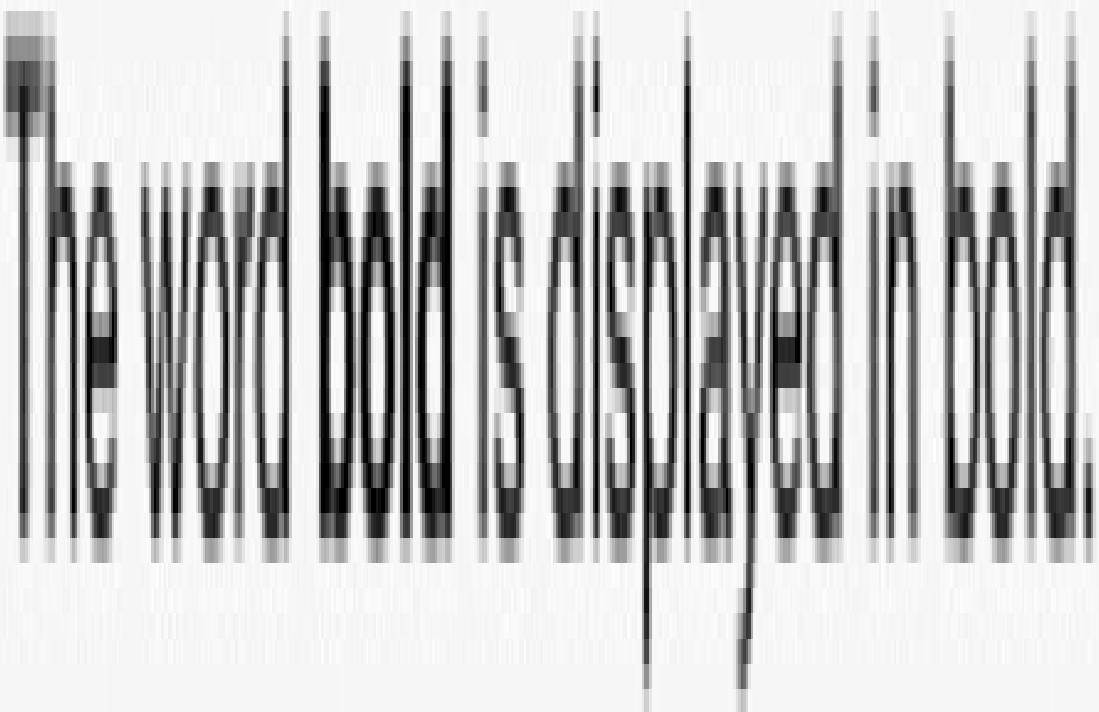


Strings can be displayed in italics and in bold:

//: The word **italics** is displayed in italics.



//: The word **bold** is displayed in bold.



It is also possible to put links to different pages of a Playground and navigate between them.

The markup syntax also allows block quotes, bullet lists, numbered lists, blocks of code, horizontal lines (“horizontal rule”), inline elements, images, and special characters.

For details, see the Apple documentation.

Playgrounds have been used for purposes far beyond that of debugging code, with some people writing books with it. See the 2015 WWDC video “Authoring Rich Playgrounds”, session 405. For a transcript go to <http://asciwwdc.com/2015/sessions/405>.

This particular syntax for displaying information, which is similar to but simpler than (more “lightweight”) than HTML, is called “markup” by Apple but is widely known as “markdown” elsewhere. The scheme was originated by John Gruber.

C The Interactive REPL Loop

An alternative to a Playground is a command-line tool, known as the “REPL Loop,” that is part of Xcode and allows you to enter short segments of code into it. It then evaluates what you have entered and responds with a printed result.

“REPL Loop” stands for “Read-Eval-Print-Loop”. The idea is that the code that implements this reads what you have entered, evaluates it, prints the result, and then continues looking for new input.

It’s easy and quick to run the REPL tool to check out something about Swift, since you don’t need to run Xcode, you just start the terminal tool.

Running the REPL Tool

To run the REPL loop, once Xcode is installed on your machine, first get a terminal window. Typically this is done by going to the Applications folder on your Mac, then the Utilities subfolder, and double clicking on Terminal.

This will display a window with a command-line user interface, including a prompt like:

```
Last login: Mon Jun 29 12:18:55 on console  
craig$
```

If you are on Mavericks (OS X version 10.9), you should enter “xcrun swift”

If you are on Yosemite or later (OS X version 10.10+), you should enter “swift”.

You will then get something like this:

```
craig$ xcrun swift  
Welcome to Swift version 1.2. Type :help for assistance.  
1>
```

Entering Swift Statements into the REPL Loop

Entering a Swift statement will yield an immediate response with the REPL indicating the type and value of a variable:

```
1> var a = 1.2  
a: Double = 1.2  
  
2> let b = "hello friend"  
b: String = "hello friend"
```

(The blue text is how the REPL displays it—it does not mean that this is a link.)

Statements on Multiple Lines

Some statements, particularly the definitions of classes and methods, usually require multiple lines. The REPL tool will allow you to enter the entire statement and not immediately evaluate it.

In such a case the prompt that you will get in such a case is a line number followed by a period rather than a line number followed by a right angle bracket. (If you get confused and get stuck in a mode where it is continuing to accept input this way and you want out, just enter “}” characters until you get back to a “>” prompt.)

```
3> class Fruit {  
4.     var skinColor = "red"  
5. }
```

If you then create an instance of the class, it will indicate the type and provide information about the properties and what they were initialized to:

```
6> var c = Fruit()  
c: Fruit = {  
skinColor = "red"  
}
```

The REPL tool has quite sophisticated editing tools and maintains a history from session to session. See the Apple documentation for details.

If You Have Multiple Versions of Xcode Installed

If you have multiple versions of Xcode installed (e.g., a production version and a beta version), you may need to use `xcode-select` to switch your REPL tool to the version of Xcode that you want to use.

Thus, if you have Xcode versions 6 and 7 both installed, and when you run the REPL tool it says that it is running Swift 1.2 (Xcode 6), but you want to run Swift 2 in the REPL, you should enter the following in the Terminal tool:

```
sudo xcode-select -switch /Applications/Xcode-beta.app/Contents/Developer
```

(The sudo command will ask you to enter the admin password for the Macintosh that you are on.)

Use the sudo command only when you are in the basic terminal window, before you start up the Swift REPL.

Note that Apple's practice is to call the production version of Xcode "Xcode.app" and the beta versions "Xcode-beta.app", so both can coexist in the Applications folder.

Thus the above command will switch to the version of Xcode named Xcode-beta.app:

```
Welcome to Apple Swift version 2.0 (700.0.38.1 700.0.53).  
Type :help for assistance.
```

1>

To get back to the production version of Swift, use:

```
sudo xcode-select -switch  
/Applications/Xcode.app/Contents/Developer
```

D Additional Resources

This book, and its online interactive exercises, is focused on the Swift language. To develop apps for iOS devices and the Mac, though, you'll need to learn quite a bit more. You'll need to (1) Continue learning Swift (which changes); (2) Learn the Xcode Development Environment (including Xcode and Interface Builder); and (3) Learn the iOS (or Mac) Application Programming Interfaces, including Foundation and either Cocoa Touch (iOS) or Cocoa (Mac).

You will need a Macintosh to run Xcode to get access to the Playground, REPL interactive tool, and the Xcode interactive development environment for developing apps. The overwhelming majority of developers use the standard Apple-supplied development environment, Xcode, even though it has issues (It has only a three plus out of five star rating on the Apple App Store, and many one star ratings. Read the reviews.)

The latest public version of Xcode is available free from the Apple Mac Store. At the time of my writing this (after the June 2015 WWDC but before the fall public release of Swift 2/Xcode 7), that version is Xcode 6.4, and it requires the Yosemite version (10.10) of OS X. Running Yosemite and Xcode requires a Macintosh with an Intel processor. Although it will run with 2 Gig of RAM, realistically you need 4 Gig of RAM or it will be very slow. Xcode version 6.4 runs Swift version 1.2.

If you are a registered Apple developer, you can get access to the beta versions of Xcode from the Apple developer site (developer.apple.com). This includes Swift 2 and Xcode 7. Apple's normal practice is to announce new versions of iOS (and Swift) at the June World Wide Developer's Conference, but to allow use of them only in beta versions available from the developer web site. In the fall (about September) they then release the new versions publicly. Thus, the next version of OS X, 10.11, known as El Capitan, is available as a beta and is expected to be available publicly in Fall, 2015. Apple's practice with iOS has been to keep information about new versions available to developers only under a non-disclosure agreement until the public launch. Their practice with Swift, however, has been to make most of the new information publicly available when announced in June.

The Xcode environment includes an interactive editor for Swift and Objective-C, a software simulator for devices including the iPhone, iPad, and Apple Watch, and Interface Builder. Interface Builder is an interactive editor for the UI components of iOS and Mac. Usually, you can create UI components either with Interface Builder or directly in code. (In some cases you can only do it with code.) The environment also allows you to load development versions of apps onto actual devices and to test them.

There is an alternative IDE for Swift known as AppCode that has been developed by JetBrains.

It is also possible to use a Macintosh server from a Windows machine via the Internet. Companies that offer such a service typically have the latest public version of Xcode installed, but not the beta version.

Apple Documentation Resources

Apple has a huge amount of documentation about Swift, Xcode, iOS, OS X, and related topics. Perhaps the biggest problem is finding what you want in the torrent of information. The quality is mixed. At its best it is beautifully written and illustrated and easy understood. At its worst it can be cryptic, incomprehensible, repetitious, and frustrating. There is more of the former than the latter.

To have access to the beta versions of Xcode and much of the documentation, you will need to become a registered Apple developer (\$99/year for individuals).

It is sometimes difficult to tell which documentation Apple makes public and which it keeps proprietary (but available to registered developers under a non disclosure agreement). For example, the videos from the 2015 Apple World Wide Developer's Conference are apparently available only to registered developers, but transcripts from them are publicly available at asciwwdc.com. One of the WWDC videos, on Protocol Oriented Programming, has been made available by Apple on YouTube.

APPLE RESOURCES ON SWIFT

Apple has generally followed a practice of making almost everything about Swift that is available to registered developers also publicly available. In addition, when Swift 2 is publicly released, source code will be publicly available as part of Apple's recent decision to make Swift available as open source.

Swift Blog. Apple maintains a blog on Swift that contains significant announcements about the language. It can be found at <https://developer.apple.com/swift/blog>.

Swift Books. Apple publishes two books, available via the iBooks store for free, that cover Swift: (1) *The Swift Programming Language*. This describes the language itself. The last part of the book is a formal definition of the language. (2) *Using Swift with Cocoa and Objective-C*. This describes how to use Swift with the Cocoa and Cocoa Touch APIs and how to write mixed Swift/Objective-C programs.

(At the time of writing, after the 2015 WWDC but before the public launch of Swift 2, Apple actually had four books available. Each of the above two books also has a Pre-release version that contains information about Swift 2.)

Swift Online Documentation. This is the same material as found in the books. Go to:

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ or Google “swift programming language”.

Swift Developer Forum. Available to registered developers only, Apple runs a forum where you can post questions and get answers.

APPLE RESOURCES FOR IOS AND OS X

Apple has very extensive documentation available on the various iOS and OS X APIs. Most of this is in one of two forms, Class Reference Manuals and Programming Guides.

Class Reference Manuals. Every class that is part of the iOS operating system (e.g., part of Cocoa Touch or Foundation) has a Class Reference Manual that is online. These give a general description of what the class does, and lists all of the methods associated with the class and describes their syntax and what they do.

Programming Guides. Many topics in iOS and OS X have programming guides, which are general descriptions of how a particular aspect of iOS or OS X works. They are less specific about particular methods (and less complete) than the Reference Manuals, but are more conceptual and give good descriptions (and often examples) of how to use a particular API.

Developer Forums. Available to registered developers only, Apple runs many different forums on various aspects of iOS and OS X where you can post questions and get answers.

Third Party Books

BOOKS ON SWIFT

Although there are many books about Swift available, most largely cover the same material as the present book. Still, I have listed those books on Swift itself that I have seen and would recommend below, as well as one on functional programming in Swift (which I do not cover in this book).

One exception with regard to overlap is the Neuburg book, listed as the first one. Matt Neuburg's book describes not only Swift but also the tools for developing it, including Xcode and Interface Builder, and some information about the Cocoa library.

iOS 8 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics. Matt Neuburg. O'Reilly Media.

Swift in 24 Hours. B. J. Miller. Sam's Publishing. This is a relatively good and complete book about Swift that is part of a series of technology books that purport to teach topics in "24 hours." The gimmick is that each chapter supposedly takes an hour to read, and there are 24 of them.

Swift Pocket Reference. Anthony Gray. O'Reilly Media. This has relatively good, if brief, descriptions of the different parts of the Swift language. It's not a "cheat sheet", as the title might suggest, but is 187 pages long in a small format book.

Swift for Programmers. Paul and Harvey Deitel. Caveat: This book claims that instances created by structures and enumerations are "objects", which is not consistent with Apple's terminology.

Functional Programming in Swift. Chris Eldhof and Florian Kugler. There is a debate among people who do functional programming about whether Swift is a decent language to do that style of programming in. (It has many of the things desired, such as the ability to pass functions as parameters.) The authors of this book believe that it is, and describe how to do it.

BOOKS ON IOS

Books on iOS are valuable because they bring the most important information about the iOS operating system in one place. However, iOS is huge, and gets more so every year. Most of the books are nearly 1000 pages long, yet still don't fully cover all of the aspects

of iOS that developers need to know about.

Most books on iOS are revised every year and come out soon after Apple makes public the information about the latest version. The books below are thus somewhat dated, but you can expect a new version coming out for each of them that covers iOS 9.

Programming iOS 8: Dive Deep into Views, View Controllers, and Frameworks. Matt Neuburg. O'Reilly Media. Neuburg's books get mixed reviews, as they are not necessarily well organized or easily understood. However, Neuburg has a great deal of knowledge about iOS, and you will often read things in his books that you will not read anywhere else.

iOS 8 App Development Essentials. Learn to Develop iOS 8 Apps Using Xcode and Swift 1.2. Neil Smyth. Second Edition. Good and fairly complete. Much of this is available online at techotopia.com.

Beginning iPhone Development with Swift: Exploring the iOS SDK. David Mark, Jack Nutting, Kim Topley, Fredrik Olsson, and Jeff LaMarche. aPress.

Newsletters on Swift

The following newsletters cover Swift development:

This Week in Swift. Weekly, curated by “Natasha the Robot”, Natasha Murashev.
<http://swiftnews.curated.co>

Swift Weekly. <http://swiftweekly.com> Twitter: @swift_weekly

Ray Wenderlich. Monthly newsletter. <http://www.raywenderlich.com/category/swift>

iOS Dev Weekly. Weekly, curated by Dave Verwer.
<https://iosdevweekly.com>

Newsletters will likely change, with some disappearing and others appearing. Do a Google for “swift programming newsletters”.

About the Author

The author, Craig Will, is currently an iOS app developer based in Northern California, both writing code (Objective-C and Swift) and doing usability design.

He is a graduate of the University of California, Irvine. He also has a Ph.D. from the University of California, San Diego.

He has worked in product development for software, particularly mobile devices, and in research, where his specialties have included artificial intelligence, neural networks, speech recognition and human-computer interaction.

He holds 19 patents, mostly in the areas of mobile devices, speech recognition, and telecommunications.

Copyright Details

Copyright © 2015 by Craig A. Will

The content of this book is protected by U.S. and international copyright law. Do not copy it except as allowed by copyright law without prior written permission.

The following is for testing fonts on different Ebook devices:

WiiW **Swift** **Tenaya** Times New Roman, Courier New, **Arial**
(bold), **Chalkboard (bold)**

ISBN-13: 978-0-9962281-0-7 Paperback edition

ISBN-10: 0996228101

ISBN-13: 978-0-9962281-1-4 Ebook edition

ISBN-10: 099622811X

The purpose of the book is to help the reader learn programming in Swift. I have tried to provide accurate information, but cannot guarantee that it is correct, complete, or current. If the reader writes a program based on information in this book and something bad happens, I am not responsible, nor is the publisher.