

O'REILLY®

2nd Edition
Covers Swift 2.1



Swift Pocket Reference

PROGRAMMING FOR iOS AND OS X

Anthony Gray

Swift Pocket Reference

Get quick answers for developing and debugging applications with Swift, Apple's multi-paradigm programming language. Updated to cover the latest features in Swift 2.1, this pocket reference is the perfect on-the-job tool for learning Swift's modern language features, including type safety, generics, type inference, closures, tuples, automatic memory management, and support for Unicode.

Designed to work with Cocoa and Cocoa Touch, Swift can be used in tandem with Objective-C, and either language can call APIs implemented in the other. Swift is still evolving, but Apple clearly sees it as the future language of choice for iOS and OS X software development.

Topics include:

- Supported data types, such as strings, arrays, array slices, sets, and dictionaries
- Program flow: loops, conditional execution, and error handling
- Classes, structures, enumerations, and functions
- Protocols, extensions, and generics
- Memory management
- Closures: similar to blocks in Objective-C and lambdas in C#
- Optionals: values that can explicitly have no value
- Operators, operator overloading, and custom operators
- Access control: restricting access to types, methods, and properties
- Ranges, intervals, and strides
- A full list of built-in global functions and their parameter requirements

IOS PROGRAMMING

oreilly.com, Twitter: @oreillymedia

US \$14.99

CAN \$17.99

ISBN: 978-1-491-94007-5



9



SECOND EDITION

Swift Pocket Reference

Anthony Gray

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Swift Pocket Reference

by Anthony Gray

Copyright © 2016 Anthony Gray. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Rachel Roumeliotis

Production Editor: Colleen Lobner

Proofreader: Christina Edwards

Indexer: WordCo Indexing Services, Inc.

Interior Designer: David Futato

Cover Designer: Randy Comer

Illustrator: Rebecca Demarest

December 2014: First Edition

November 2015: Second Edition

Revision History for the Second Edition

2015-11-11: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491940075> for release details.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-94007-5

[M]

Table of Contents

Introduction	1
Conventions Used in This Book	2
Using Code Examples	2
Safari® Books Online	3
How to Contact Us	4
Acknowledgments	5
Getting Started with Swift 2	5
A Taste of Swift	12
Basic Language Features	14
Types	18
Variables and Constants	23
Tuples	27
Operators	30
Strings and Characters	39
Arrays	48
Dictionaries	57
Sets	62
Functions	68
Closures	75
Optionals	81

Program Flow	88
Classes	106
Structures	139
Enumerations	144
Access Control	151
Extensions	155
Checking and Casting Types	158
Protocols	162
Memory Management	187
Generics	193
Operator Overloading	201
Ranges, Intervals, and Strides	206
Global Functions	210
Changes From Swift 1.0	215
Index	217

Swift Pocket Reference

Introduction

Swift is an exciting new language from Apple, first announced at the Apple Worldwide Developers Conference (WWDC) in June 2014. The language started life as the brainchild of Chris Lattner, director of Apple's Developer Tools department, and is the next step in the evolution of Apple's software development ecosystem.

Swift brings with it many modern language features, including type safety, generics, type inference, closures, tuples, protocols, automatic memory management, and support for Unicode (for character and string values as well as for identifiers). You can use a mixture of Swift and Objective-C in a single project, and either language can call APIs implemented in the other.

The challenge for anyone learning or writing about Swift is that the language is still evolving. When they introduced it, Apple stated that the language specification was not final, and that the syntax and feature set would change. Since the initial release, there have been two significant updates (versions 1.2 and 2.0), both of which introduced new features, and in some cases changed existing features. Fortunately, Xcode can detect most of the cases where your code is using an older syntax or feature, and offers hints at what to change to address this.

Despite the uncertainty of a changing language, Swift continues to show great promise. It follows on from the company's other major developer tools initiatives (all led by Lattner) including LLVM, Clang, LLDB, ARC, and a series of extensions to Objective-C, and it's clear that Apple sees it as the future language of choice for iOS and OS X software development.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

NOTE

This element signifies a general note.

Using Code Examples


You can download code samples that accompany this book at <https://github.com/adgray/SwiftPocketReference2ndEd>.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Swift Pocket Reference* by Anthony Gray (O'Reilly). Copyright 2016 Anthony Gray, 978-1-491-94007-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

 **Safari**® *Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/swift_pocket_ref_2e.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I would like to thank fellow O'Reilly author Paris Buttfield-Addison for urging me (repeatedly) to write this book. He and his partner-in-crime, Jon Manning, suffer from boundless optimism and seem to regard “no” as a challenge rather than as a defeat. I'd also like to thank Rachel Roumeliotis and the other fine folk at O'Reilly for having faith in me and for shepherding the project (and the update) through to completion. Special thanks also go to the readers of the early release editions, who took the time to provide feedback and suggestions for improvement, for which I'm deeply grateful.

Getting Started with Swift 2

To code in Swift 2, you should be using Xcode 7 or later (available for free on the Mac App Store), which runs on either OS X 10.10 (Yosemite) or OS X 10.11 (El Capitan). You might also consider signing up as a registered Apple developer (with free and paid tiers) to gain access to a wealth of documentation and other developer resources at <https://developer.apple.com>.

You can use the version of Swift built into Xcode 7 to compile programs that will run on OS X 10.9 or higher, and on iOS 7 or higher.

After you have downloaded and installed Xcode 7, go ahead and run it and allow it to install the various tools it comes bundled with. When installation is complete, there are a number of ways you can get started with Swift:

- Click File → New Project to create a new Xcode project. The project wizard opens and offers you the choice of using Swift or Objective-C as the language for the project.
- Click File → New Playground to create a new playground document. Playgrounds are single-window dynamic environments in which you can experiment with Swift

language features and see results instantly alongside the code you enter.

- Create a Swift script and run it from the command line in the OS X terminal.
- Use the Swift Read-Evaluate-Print-Loop (REPL) in the OS X terminal.

Let's look at the REPL, Swift scripting, and playgrounds in more detail.

NOTE

As of this writing, some features of Swift and Xcode 7 are still unstable, but the situation improves with each release. Occasionally, you might need to quit and restart Xcode to get it back to a sensible state.

The Swift REPL

The Swift REPL provides command-line access to Swift and behaves like an interpreter. You can declare variables and constants, define functions, evaluate expressions, and use most other language features; they will be compiled and executed immediately.

Multiple Xcode installations

If you have more than one installation of Xcode on your computer, you will need to use the **xcode-select** command to choose the Xcode 7 environment as the active developer directory. In the terminal, type the following command:

```
sudo xcode-select -s /Applications/Xcode.app
```

When prompted, provide your administrator username and password. If you have installed Xcode in a different location or

changed its name, replace the path in the command with the location and name of your installed release.

Starting the REPL

To start the REPL so you can test Swift language features, use the following command:

```
xcrun swift
```

If you've never used Xcode before, you might see an authentication prompt from a process called *Developer Tools Access* (see [Figure 1](#)), prompting you for a username and password. You will need to enter an administrator username and password to continue. After you enter these, you might see the following error message:

```
error: failed to launch REPL process: process
exited with status -1 (lost connection)
```

At this point, type the `xcrun swift` command again. This time, the REPL should start normally.

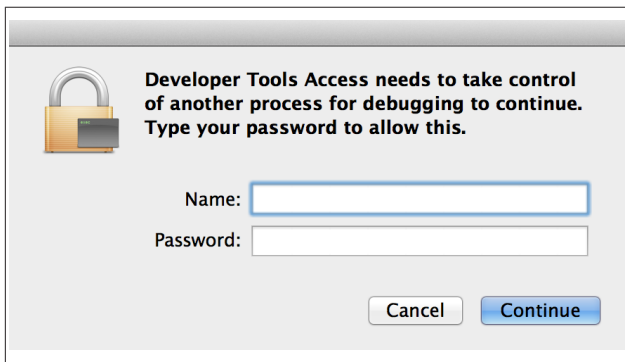


Figure 1. The Developer Tools Access prompt

When the REPL starts, you will see the following output:

```
Welcome to Swift! Type :help for assistance.
1>
```

You're now ready to try your first code in Swift. Try the `print` function:

```
1> print ("Hello, World")
Hello, World
2>
```

The REPL is a great way to test Swift features and experiment with the language.

Swift as a Scripting Language

You can use Swift as a scripting language, much like Perl, Python, or Ruby. To use Swift in this manner, ensure the first line of the script contains the path to the Swift “interpreter.” If you want to try using Swift this way, type the following into a text file named *hello.swift*:

```
#!/usr/bin/swift

print ("Hello, World")
```

Next, ensure the script is marked as executable with a `chmod` command:

```
chmod u+x hello.swift
```

Now, run the script as follows:

```
./hello.swift
```

Swift will compile your program, and assuming there are no syntax errors, will execute it.

Swift Playgrounds

To explore Swift in a playground, on the Xcode menu bar, click `File` → `New Playground`, or click the “Get started with a playground” option in the Welcome to Xcode window.

You are then prompted to enter a playground name (which becomes the saved document’s name) and a platform (iOS or OS X), as demonstrated in [Figure 2](#).

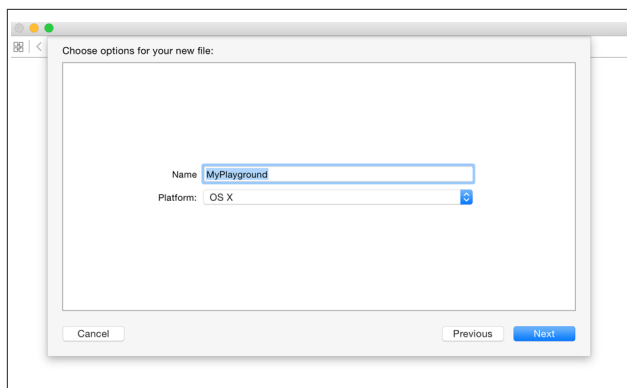


Figure 2. Creating a Swift playground

Once you've entered your playground name and selected your platform, click Next. You will then be prompted to select a location to which to save the file. When the file has been saved, you see the initial playground window, as shown in [Figure 3](#).

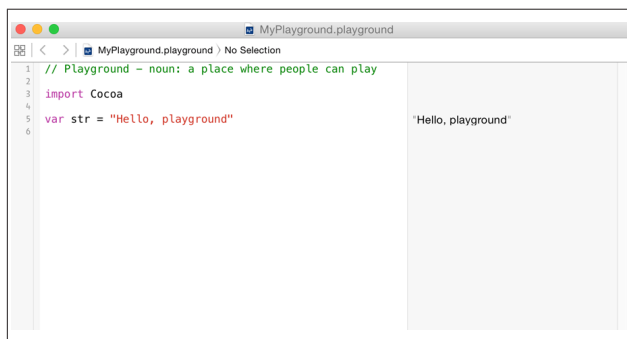


Figure 3. The initial playground window

The playground template imports either the Cocoa or UIKit Framework, depending on whether you selected OS X or iOS as your playground platform. This means you can experiment not just with basic Swift language features, but also with many

of the features provided by the framework, such as drawing views and images, and even implementing basic animations.

The playground also displays a line of code:

```
var str = "Hello, playground"
```

To the right of that code is the value “Hello, playground.” This demonstrates one of the most useful features of the playground: the result of every expression displays alongside it in the results sidebar.

Below the existing text, type the following:

```
for var i=0; i<10; i++ {  
    print (i)  
}
```

The results sidebar now displays the text “(10 times)” to confirm the number of executions of the loop.

If you hover the pointer over the entries in the results sidebar (Figure 4), you’ll see two symbols. The eye-like symbol provides a *Quick Look* view of the value (this includes viewers for complex data such as strings, arrays, dictionaries, images, views, URLs, and more). The button symbol opens a Result view in line with the Swift code that generated it (Figure 5).

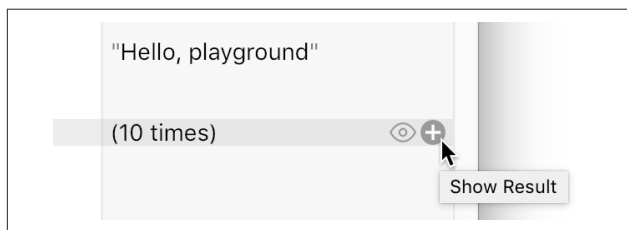


Figure 4. Accessing Quick Look and the Result view from the results sidebar

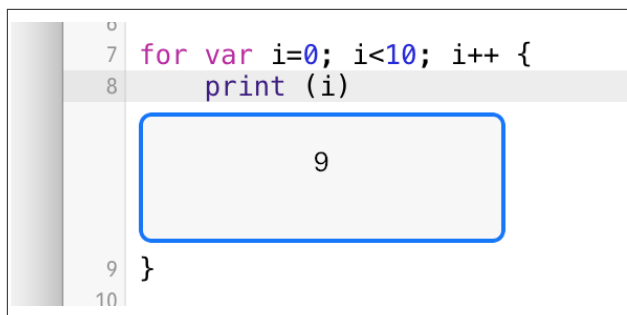


Figure 5. Showing a Results view in line with Swift code

Note that a Results view generally only shows the last result that was generated (9, in the case of the for loop). If you right-click in a Results view you'll expose a menu that allows you to choose between the latest value and the value history.

The Debug Area (Figure 6) shows you console output (e.g., text output by the print function). To show the Debug Area, click the small upward-pointing triangle inside the square box at the lower-left of the Playground window.

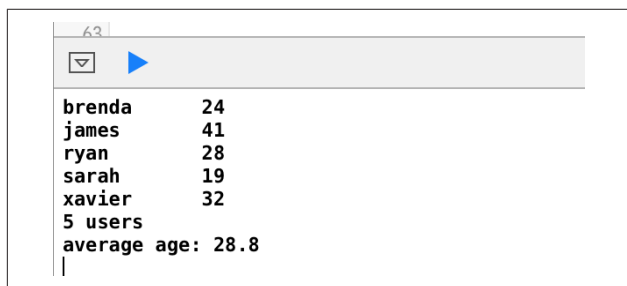


Figure 6. The Debug area showing console output

You can also open the Debug Area by going to the Xcode menu bar and clicking View → Debug Area → Show Debug Area, or you can use the keyboard shortcut Command-Shift-Y.

NOTE

For an excellent introduction to playgrounds, see the recording of session 408 (Swift Playgrounds) from the [2014 Worldwide Developers Conference](#), and session 405 (Authoring Rich Playgrounds) from the [2015 Worldwide Developers Conference](#).

A Taste of Swift

Let's dive right in. What follows is a simple program written in Swift. Work through carefully to get a sense of some of the features of the language.

The first thing the program does is define a pair of arrays: one named `users`, and another named `ages` for those users. This is meant to represent raw input of disassociated data that needs to be merged and then used as the basis of a report:

```
// some raw data to process
var users = ["xavier", "ryan", "brenda", "james", "sarah"]
var ages = [32, 28, 24, 41, 19]
```

The next section of code is a pair of extensions to the `String` type. Swift has built-in support for Unicode strings, but it also has a very flexible extension mechanism with which you can add new features to the language—even to built-in types. The extension adds two new member functions to the `String` type that can take a string and return a copy that is padded with leading or trailing spaces to a specified width:

```
// add some extensions to the String type
extension String
{
    func leadingSpaces(width: Int) -> String
    {
        var s = String(self)
        for i in s.characters.count..
```

```

func trailingSpaces (width: Int) -> String
{
    var s = String(self)
    for i in s.characters.count..

```

Next, a dictionary merged is declared. This is an associative array of key/value pairs to store each user's name and age. The variable `totalAge` is also declared, to store the sum of all of the ages so that later the average age of all users can be calculated:

```

// a dictionary to store merged input
var merged = [String: Int]()
var totalAge = 0.0

```

With the dictionary defined, the next step is to iterate over the two input arrays, merging them into the `merged` dictionary. The dictionary utilizes the user's name as the key and the user's age as the value:

```

// merge the two arrays into a dictionary
for var i=0; i < ages.count; i++ {
    merged[users[i]] = ages[i]
}

```

At this point the dictionary contains all of the raw input, and it's time to generate a report. We want to list the users in sorted order, and print each user's age along with their name using trailing and leading spaces so the names are left-aligned under one another and the ages are right-aligned:

```

// iterate over the dictionary in sorted order
// and generate a report
for user in merged.keys.sort() {
    let age = merged[user]!
    totalAge += Double(age)
    let paddedUser = user.trailingSpaces(10)
    let paddedAge = "\(<age)&quot;.leadingSpaces(3)
    print ("\"(<paddedUser) \"(<paddedAge)&quot;")
}

```

```
print (merged.count, "users")
print ("average age:", totalAge / Double(merged.count))
```

The output of the program looks like this:

```
brenda      24
james       41
ryan        28
sarah       19
xavier      32

5 users
average age: 28.8
```

If you've followed along, you should have a good sense of some of the language's capabilities. You've already been exposed to comments, arrays and dictionaries, various loop types, type conversion, function calls, extensions, string interpolation, and console output.

The remainder of this book will take you on a tour of these topics and all of the major aspects of the Swift language. Generally, the intention is to cover basic features before more advanced features, but at times it is necessary to dip into advanced topics early on. When this happens, we'll try to warn you in advance and point to where you can find more coverage.

Finally, note that this book is about the Swift language, not about iOS or OS X development. O'Reilly has some excellent titles that cover using Swift in those broader contexts, but the examples and discussion you'll find here are deliberately limited to pure Swift as much as possible.

Basic Language Features

Before delving into the specifics of the language such as data types or program flow, let's take a look at some of the more general aspects of Swift source code.

Comments

Swift supports C-style comments. Comments that extend to the end of the current line begin at a double forward-slash, as illustrated here:

```
// this is a comment  
print ("Hello, world") // really?
```

For multiline comments, you enclose them between a forward-slash followed by an asterisk at the beginning and end them with an asterisk followed by a forward-slash, as shown in the following:

```
/*you could be excused when looking at this comment  
for thinking that it was written in C! */
```

Unlike C, Swift supports multiline comment nesting. Swift treats the following as a nested comment block, whereas traditional C compilers will report an error at the last */:

```
/*  
/* Original comment here */  
*/
```

Multiline comment nesting makes it possible for you to define blocks of commented code using the /* and */ comment markers without having to first check whether an inner multiline comment block exists.

Semicolons

Semicolons in Swift are only required when you need to separate multiple statements on a single line. They are not required at the end of a statement if that is the last statement on the line (but it is not an error to use them):

```
var a = 45; // semicolons are optional  
var b = 55
```

Whitespace

Swift uses whitespace to separate tokens, but whitespace has no other intrinsic meaning. Whitespace include spaces, tabs, line

feeds, carriage returns, vertical and horizontal tabs, and the null character.

Because whitespace is ignored, you can use line breaks to split long lines at token boundaries. Thus, the following two statements are equivalent:

```
var a = 45
```

```
var a  
=  
45
```

There is no formal way to split a long string over multiple lines as there is in C or Objective-C, but strings can be concatenated by using the + operator. So, you could split a long string like this:

```
let longstr = "Hello there this is a very " +  
             "long string split over two lines"
```

Naming Conventions

There are some conventions in relation to naming entities you are strongly urged to follow. The principal conventions are:

- Names for classes, structures, enumerations, protocols, and types should begin with an uppercase letter, and use camel case for the remainder of the name.
- Names for functions, methods, variables, and constants should begin with a lowercase letter, and use camel case for the remainder of the name.

Importing Code from Other Modules

In C-like languages, the usual way to use code from other parts of a project or from libraries or frameworks is by using the #include directive. When placed in a source file, this directive instructs the compiler to read a header file into the compilation process.

The header file declares what classes, methods, functions, and external variables are provided by the external code, and are therefore valid to call or access in the source file that hosts the `#include` directive. The feature provided by the code, or links to it, are linked into the executable file at the last stage of the compilation process by a program called a *linker*.

Swift does away with header files and the need to include them. Instead, Swift uses the `import` command, which imports the definitions made available by another module.

The basic syntax is as follows:

```
import ModuleName
import Cocoa
```

Used in this way, everything that *ModuleName* makes public is imported.

If a module provides submodules, you can import a specific submodule, with this syntax:

```
import ModuleName.SubmoduleName
import Foundation.NSDate
```

If you only want to import a single feature from a module, use this syntax:

```
import Feature ModuleName.SymbolName
import func Darwin.sqrt
```

This last directive imports just the `sqrt` function from the Darwin module. *Feature* describes the type of the entity to be imported; the type can be one of the following: `class`, `enum`, `func`, `protocol`, `struct`, `typealias`, or `var` (all of which are described throughout the remainder of this book).

You can import most of the standard OS X and iOS frameworks into a Swift project, including, for example, AppKit, Cocoa, CoreData, Darwin, Foundation, UIKit, and WebKit. Refer to Apple's documentation on OS X and iOS for more information on these and other frameworks at <https://developer.apple.com>.

In most applications, you'll only need to import Cocoa (for OS X applications) or UIKit (for iOS applications), because they in turn import most other modules that are normally required for these application types.

If you are using the Xcode editor, you can see what additional submodules these modules import or make available by holding down Command-Option and simultaneously clicking the module name in the line that imports that module.

Types

Swift supports the standard data types you would expect in a modern programming language. These are listed in [Table 1](#).

Table 1. Supported data types in Swift

Data Type	Description
Bool	Boolean value (true, false)
Int	Signed integer value
UInt	Unsigned integer value
Double	Double-precision (64-bit) floating-point value
Float	Single-precision (32-bit) floating-point value
Character	A single Unicode character
String	An ordered collection of characters

Specific Integer Types

Int and UInt are 32- or 64-bit values, as determined by the underlying platform. Swift also supports integer types of specific size (and hence numeric range). [Table 2](#) shows the range of values for each type.

Table 2. Specific integer types and their value ranges

Name	Type	Range
Int8	Signed 8-bit integer	-128 to 127
UInt8	Unsigned 8-bit integer	0 to 255
Int16	Signed 16-bit integer	-32,768 to 32,767
UInt16	Unsigned 16-bit integer	0 to 65,535
Int32	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
UInt32	Unsigned 32-bit integer	0 to 4,294,967,295
Int64	Signed 64-bit integer	-2^{63} to $2^{63}-1$
UInt64	Unsigned 64-bit integer	0 to $2^{64}-1$

You can determine the maximum and minimum values that can be stored by each integer type by using the `max` and `min` properties, as demonstrated in the following example:

```
UInt8.max    // returns 255
Int16.min    // returns -32768
```

Numeric Literals

Numeric literals can be expressed in decimal, binary, octal, or hexadecimal, as presented in [Table 3](#).

Table 3. Expressing numeric literals

Prefix	Base	Example(s)
None	Decimal	17, 1024, 2.767, 2.5e2
0b	Binary	0b10001011
0o	Octal	0o213
0x	Hexadecimal	0x8C, 0x4.8p2

When using numeric literals, also note the following:

- Floating-point literals may be followed by an optional exponent, which is expressed with an `e` for decimal

floating-point literals, or a `p` for hexadecimal floating-point literals, which is in turn followed by the exponent itself, in decimal. For example, `e3` represents a decimal exponent that multiplies the mantissa by 10^3 , and `p4` represents a hexadecimal exponent that multiplies the mantissa by 2^4 . Examples include `2.7e4`, which equates to 2.7×10^4 and `0x10.4p2`, which equates to $0x10.4 \times 2^2$ (the decimal equivalent being 16.25×2^2).

- To be inferred as a floating-point value, a literal without an exponent must have a decimal point with digits on either side. The presence of an exponent removes the requirement for a decimal point if it is not needed (e.g., `5e2` evaluates to 5.0×10^2).
- Numeric literals can include underscores (but not commas) to aid readability. `1_000_000` and `1_00_00_00` are the same as `1000000`.
- Floating-point literals are treated as `Double` values, unless they are used in a `Float` context.

Character and String Literals

Character literals are single characters surrounded by double quotes (unlike C-based languages, in which single quotes are used), as shown here:

```
"A"  
"B"  
"!"
```

String literals are character sequences surrounded by double quotes:

```
"Hello, World"
```

The compiler cannot distinguish between a character literal and a single-character string literal. Literals enclosed in double quotes are treated by the compiler as strings, unless they appear in a character context, such as in assignment:

```
let someChar: Character = "C"  
// "C" is treated as a character literal  
var c: Character  
c = "A"  
// "A" is treated as a character literal
```

Type Aliases

The `typealias` keyword defines an alternative name for an existing type. The following example equates the identifier `Byte` with the type `UInt8`:

```
 typealias Byte = UInt8
```

After this declaration, `Byte` can then be used as a type anywhere that `UInt8` can be used, such as in the following example:

```
 var acc: Byte = 64
```

Aliases can be created for any type, including for example, function types:

```
 typealias DoubleInDoubleOut = (Double) -> Double
```

This creates a type alias for a function that takes one parameter of type `Double`, and returns a result of type `Double`. This alias can be used anywhere this function signature would normally be written, as in the following example:

```
 var f: DoubleInDoubleOut = {  
     return $0 * $0  
 }  
 f(12.0)  
 // returns 144.0
```

See also the sections [“Tuples” on page 27](#), [“Functions” on page 68](#), and [“Protocols” on page 162](#) for related examples of type aliases.

Nested Types

Swift supports the definition of types within types, as shown in this example:

```
 class A  
 {
```

```

class B
{
    var i = 0
}
var j = B()
var k = 0
}

```

Although you can use such nested definitions to provide utility classes, structures, or enumerations to support the implementation of the outer class, the nested type definitions are visible outside the class as well. For the preceding definition, you can create instances of A and B as follows:

```

var a = A()
var b = A.B()
a.j.i = 2
b.i = 5

```

If a class contains a nested enumeration, as follows:

```

class C
{
    enum TravelClass
    {
        case First, Business, Economy
    }

    // rest of class definition
}

```

then the enumeration can be accessed outside the class by specifying the “path” to the enumeration values, as follows:

```

var t = C.TravelClass.First

```

Other Types

In addition to the types already discussed, you can use many other Swift language elements in a type context, or they can behave as types. These include classes, structures, enumerations, functions, and protocols. These topics are covered in later sections of this book.

Variables and Constants

Variables and *constants* must be declared before you use them.

You declare variables by using the `var` keyword, followed by the variable's name, a colon, and then its type, as shown here:

```
var name: Type
var anInt: Int
var aStr: String
var aChar: Character
```

You can assign values to variables at the same time you declare them:

```
var anotherInt: Int = 45
var anotherStr: String = "Frodo"
```

Swift uses *type inferencing*, which means you don't need to specify a variable's type if you assign that variable a value when you declare it:

```
var someInt = 45
var someDouble = 23.0, someStr = "Strings too"
```

You declare constants by using the `let` keyword. They look like variables because of the way they are created and used, but they are *immutable*—meaning they cannot be changed. Because a constant cannot be changed, it must be assigned a value when it is declared (the exception is for *constant properties* in classes and structures, which can have their value assigned during initialization—see “Classes” on page 106 and “Structures” on page 139 for more information):

```
let name: Type = expr
let constFloat: Float = 23.1
let constStr: String = "Bilbo"
```

As with variables, the type of a constant will be inferred from the value you assign to it, so in most circumstances, you do not need to specify the type:

```
let someConstFloat = 23.1
let someConstStr = "Bilbo"
```

You can declare the type explicitly for circumstances in which the inferred type is not desired. This is useful when you want to declare a `Character` type where `String` might otherwise be inferred, or a `Float` type where a `Double` might be inferred, as illustrated here:

```
let c: Character = "A"  
// "A" is otherwise inferred to be a String  
let f: Float = 3.14159  
// 3.14149 is otherwise inferred to be a Double
```

The names of variables and constants can contain most Unicode and other characters. They cannot begin with a number.

Some keywords are reserved for specific language features, and you cannot use them as identifiers for variables and constants. Examples include `class`, `func`, `let`, `var`, and so on. However, if you enclose a keyword with backticks, you can use it as an identifier, like this:

```
var func = 4           // not allowed - func is reserved  
var `func` = 4        // allowed
```

Despite this, you should be wary of using backticks as a means of using keywords as identifiers. A best practice is to avoid using reserved keywords at all.

Computed Variables

A *computed variable* is not a variable in the usual sense—it is not a value that is stored in memory and read whenever it is referenced in an expression or during assignment. Instead, computed variables are functions that look like variables.

A computed variable contains two functions: a *getter* (identified with the keyword `get`, which returns the computed value) and a *setter* (identified with the keyword `set`, which might initialize the conditions that affect the value returned by the getter). The declaration looks as follows:

```
var variableName: someType {  
  get {  
    // code that computes and returns  
    // a value of someType
```

```

    }
    set(valueName) {
        // code that sets up conditions
        // using valueName
    }
}

```

The *valueName* is optional; you use it inside the code that implements the setter to refer to the value passed into the set method. If you omit it, the parameter can be referred to using the default name of *newValue*.

The setter is optional, and for most practical uses, you would not use it. If you don't use the setter, the get clause is not required, and all that is required is code to compute and return a value.

```

var variableName: someType {
    // code that computes and returns a value
}

```

When a computed variable is defined, it is used exactly like any other variable. If its name is used in an expression, the getter is called. If it is assigned a value, the setter is called:

```

var badPi: Float {
    return 22.0/7.0
}

let radius: Float = 1.5
let circumference = 2.0 * badPi * radius

```

As global or local variables, computed variables would appear to be of limited use, but the same syntax can also be used for properties in structures and classes. In this context, as *computed properties*, the feature becomes more useful. For more information about computed properties, see the section “[Properties](#)” on page 110.

Variable Observers

Variable observers are functions (or methods) you can attach to variables and that are called when the value of the variable is about to change (identified with the `willSet` keyword) or after

it has changed (identified with the `didSet` keyword). The declaration looks as follows:

```
var variableName: someType = expression {
    willSet(valueName) {
        // code called before the value is changed
    }
    didSet(valueName) {
        // code called after the value is changed
    }
}
```

When variable observers are used with global and local variables, the type annotation is required, as is the expression used to initialize the variable.

Both *valueName* identifiers (and their enclosing parentheses) are optional.

The `willSet` function is called immediately before the value of the variable is about to be changed. The new value is visible inside `willSet` as either *valueName* or `newValue` if *valueName* was not specified. The function is unable to prevent the assignment from happening and unable to change the value that will be stored in the variable.

The `didSet` function is called immediately after the value of the variable has been changed (except for after the initial assignment). The old value of the variable is visible inside `didSet` as either *valueName* or `oldValue` if *valueName* was not specified:

```
var watcher: Int = 0 {
    willSet {
        print("watcher will be changed to", newValue)
    }
    didSet {
        print("watcher was changed from", oldValue)
    }
}
```

The `didSet` function can modify the value of the observed variable without `willSet` or `didSet` being called recursively, so you can use `didSet` to act as a guard or validator of values stored in

the variable. Here is an example of using `didSet` to ensure an integer variable can only have an even value:

```
var onlyEven: Int = 0 {
    didSet {
        if ((onlyEven & 1) == 1) { onlyEven++ }
    }
}
```

It is not necessary to define both `didSet` and `willSet` functions if only one of them is required.

You can use the same syntax that is used for variable observers for properties in structures and classes, creating *property observers*. See “[Properties](#)” on page 110 for more details.

Tuples

A *tuple* is a group of values you can treat as a single entity. Tuples are enclosed in parentheses, with each element separated by a comma. [Table 4](#) provides a few examples.

Table 4. Tuple examples

Tuple	Description
(4, 5)	A tuple with two integer parts
(2.0, 4)	A tuple with a double-precision floating-point part and an integer part
("Hello", 2, 1)	A tuple with a string part and two integer parts

The collection of types of each component of the tuple, in order, is considered to be the *type* of the tuple.

The type of each tuple in [Table 4](#) is as follows:

```
(Int, Int)
(Double, Int)
(String, Int, Int)
```

You can store a tuple in a variable or constant of that tuple’s type, or pass it to or from functions for which that tuple’s type is acceptable.

NOTE

Although they are useful for storing temporary or related values in a single container, tuples are not an appropriate method for storing structured, complex, or persistent data. For such cases, consider using dictionaries, classes, or structures instead.

Tuple Variables and Constants

To create a variable or constant that stores a tuple, you list the tuple's component types inside parentheses where you would usually specify the type, as shown in the following:

```
var a: (String, Int) = ("Age", 6)
let fullName: (String, String) = ("Bill", "Jackson")
```

Because Swift uses type inferencing, the tuple type can be inferred if the variable or constant is initialized when it is declared. In the following example, there is no need to specify that the tuple's type is `(String, Int, String)`, because it is inferred by the compiler:

```
var loco = ("Flying Scotsman", 4472, "4-6-2")
```

Extracting Tuple Components

Much like arrays, you can access tuple components by position, with the first component having an index of 0:

```
var loco = ("Flying Scotsman", 4472, "4-6-2")
let name = loco.0 // assigns "Flying Scotsman"
let number = loco.1 // assigns 4472
```

Naming Tuple Components

You can name tuple components and then access them by those names. This example names the first component of the tuple *name* and the second component *age*:

```
var person: (name: String, age: Int)
person.name = "Fred"
```

```
person.age = 21
let c = person.age

let result = (errCode: 56, errorMessage:"file not found")
var s = result.errorMessage
// s is now the string "file not found"
```

Using Type Aliases with Tuples

You can use type aliases to associate a type identifier with a tuple type, and that alias can then be used to create new instances of that tuple type:

```
typealias locoDetail =
    (name: String, number: Int, configuration: String)
var thomas: locoDetail = ("Thomas", 1, "0-6-0")
```

Or a function could return a tuple of that type (see also “[Functions](#)” on page 68), as demonstrated here:

```
func getNextLoco() -> locoDetail
{
    // do something then return a value of type locoDetail
}
```

Type inferencing works with type aliases, so in

```
var anEngine = getNextLoco()
```

the variable `anEngine` will also be of type `locoDetail`.

Tuples as Return Types

Tuples are a convenient way to return more than one value from a function or method call.

Consider a function that, on each successive call, returns the next line of text from a file. At some point, the end of the file will be reached, and this needs to be communicated to the caller. The end-of-file state needs to be returned separately to the line of text itself, and this is a natural fit for a tuple:

```
func readLine () -> (Bool, String)
{
    ...
}
```

The function could even name the tuple parameters, as is done here:

```
func readLine () -> (eof: Bool, readLine: String)
{
    ...
}
```

Using tuples in this way produces a more natural expression and avoids more opaque techniques to test if the end-of-file was reached.

Operators

Operators are symbols that represent some operation to be applied to values (usually expressed as literals, variables, constants, or expressions). Examples of well-known operators include the plus sign (+), which normally represents addition (or, in the case of strings, concatenation), and the minus sign (-), which represents subtraction.

Operators are often characterized as *unary* (which operate on a single value), *binary* (which operate on two values), or *ternary* (which operate on three values).

The Swift language supports *operator overloading*, so it is important to remember that the actual operation performed by an operator will be determined by the type of data to which it is applied. The descriptions that follow relate to the default behavior. (See also “[Operator Overloading](#)” on page 201.)

No Implicit Type Conversion

Before considering the specific operators supported by Swift, you should note that Swift does not do implicit type conversion. This means the following will not compile, because the operands `f` and `i` are of different types (one is a `Double`, one is an `Int`):

```
var i = 2
var f = 45.0
let errResult = (f / i) // error
```

Unlike C-based languages, Swift will not do implicit type conversion in expressions—you must explicitly convert operands to the desired type. For numeric types, that means treating the type as a function, and the operand to be converted as its argument:

```
let result = (f / Double(i))
```

It is also important to note that Swift’s type inference rules will treat a floating-point literal as a `Double`, unless it is used to initialize a variable of type `Float`. In the preceding example, `f` is inferred to be a `Double`, not a `Float`, so `i` must be cast to a `Double`.

Arithmetic Operators

The standard binary arithmetic operators in Swift are the same as in other languages:

- + Addition (or string concatenation, if both operands are strings)
- Subtraction
- * Multiplication
- / Division
- % Remainder

NOTE

Unlike other languages, Swift does not allow an overflow or underflow using these operators. If such an overflow or underflow occurs, the program will terminate (or the issue will be flagged ahead of time by the compiler, if possible). For more information about this, see the section “[Overflow Operators](#)” on page 35.

- ++
Pre- or post-increment
- Pre- or post-decrement

As with C, these last two unary operators will increment or decrement a variable of `Int`, `Float`, or `Double` type. They also return a value. When you use them as a prefix (the operator appears to the left of the operand), they return the new (incremented or decremented) value. When you use them as a postfix (the operator appears to the right of the operand), they return the original (pre-increment or pre-decrement) value.

Bitwise Operators

The following operators are used with integer data types and permit bit-level manipulation:

- ~ (~A)
Bitwise NOT; inverts all bits in a number
- & (A & B)
Bitwise AND of A and B
- | (A | B)
Bitwise OR of A and B
- ^ (A ^ B)
Bitwise XOR of A and B
- << (A << B)
Bitwise left-shift of A by B bits
- >> (A >> B)
Bitwise right-shift of A by B bits

When the left operand is an unsigned type, the left-shift and right-shift operators always shift in new bit values of zero.

When the left operand is a signed type, the left-shift and right-shift operators preserve the sign bit at all times. The left-shift operator always shifts in new bit values of zero, whereas the right-shift operator always shifts in new bits with the same value as the sign bit.

Assignment Operators

Other than the regular assignment operator (=), all of the other operators described here are *compound* assignment operators (i.e., they combine another operation, such as addition or subtraction, with an assignment):

=	Assignment
+=	Add and assign (a += n is equivalent to a = a + n)
-=	Subtract and assign (a -= n is equivalent to a = a - n)
*=	Multiply and assign (a *= n is equivalent to a = a * n)
/=	Divide and assign (a /= n is equivalent to a = a / n)
%=	Remainder and assign (a %= n is equivalent to a = a % n)
<<=	Bitwise left-shift and assign (a <<= n is equivalent to a = a << n)
>>=	Bitwise right-shift and assign (a >>= n is equivalent to a = a >> n)
&=	Bitwise AND and assign (a &= n is equivalent to a = a & n)
=	Bitwise OR and assign (a = n is equivalent to a = a n)
^=	Bitwise XOR and assign (a ^= n is equivalent to a = a ^ n)

NOTE

Unlike C-based languages, assignment operators do not return a value. This prevents a potentially serious error whereby you accidentally type an = operator in an if statement when you meant to use == and end up with code that makes an assignment instead of testing a condition.

Comparison Operators

The *comparison operators* return a Boolean value that represents whether the comparison is true or false. *Equality* refers to whether the left and right operands have the same value. *Identity* refers to whether the operands reference the same object:

== (A == B)

Test equality (same values)

!= (A != B)

Test inequality

=== (A === B)

Test identity (same objects)

!== (A !== B)

Test unidentity

< (A < B)

Test less than

<= (A <= B)

Test less than or equal to

> (A > B)

Test greater than

>= (A >= B)

Test greater than or equal to

~= (A ~= B)

Pattern match—used indirectly in the case labels of switch statements.

Logical Operators

In Swift, non-Boolean values (such as `Int`) cannot be silently cast to Boolean values. The following logical operators can only be used on `Bool` values:

`!` (`!A`)

Logical NOT; returns the logical opposite of the operand

`&&` (`A && B`)

Logical AND; returns true if both operands are true

`||` (`A || B`)

Logical OR; returns true if either operand is true

Overflow Operators

The *overflow operators* only accept integer operands; they do not cause an error if an arithmetic overflow occurs:

`&+`

Overflow addition

`&-`

Overflow subtraction

`&*`

Overflow multiplication

If you need to know whether an overflow actually occurred, the integer types implement function equivalents of these operators that return a tuple containing the result and a Boolean value indicating overflow state. These functions are:

- `addWithOverflow()`
- `subtractWithOverflow()`
- `multiplyWithOverflow()`
- `divideWithOverflow()`
- `remainderWithOverflow()`

For example, the function to perform overflow addition for the `Int` type is defined as:

```
Int.addWithOverflow(Int, Int) -> (Int, Bool)
```

and could be called as follows:

```
let (result, overflow) = Int.addWithOverflow(someInt, someInt)
```

Following this assignment, `result` would contain the integer result of the addition and `overflow` would contain a Boolean value of `true` if an overflow occurred.

Type Casting Operators

`is`

Checks whether an instance is of a specific subclass type, or an instance conforms to a protocol.

`as`

Casts an instance reference to another type, when it is known that the cast will always succeed. Used for upcasting (treating an instance as its supertype) and for bridging (e.g., casting an `NSString` to a `String` type, or vice versa).

`as!`

Forcibly casts an instance reference to a specific subclass type, or an instance reference to a specific protocol type. Causes a runtime error if the cast fails.

`as?`

Optionally casts an instance reference to a specific subclass type, or an instance reference to a specific protocol type. Returns an optional value or `nil` if the cast fails.

See also the sections [“Checking and Casting Types” on page 158](#) and [“Protocols” on page 162](#).

Range Operators

The *closed range operator* ($x...y$) represents all integer values starting at x and ending at y . x must be less than or equal to y . This operator can be used in a loop, as in the following:

```
for i in 1...5 {  
    // i will successively take values from 1 through 5  
}
```

The *half-open range operator* ($x.. y) represents all integer values starting at x and ending at $y - 1$. The value for x must be less than or equal to $y - 1$:$

```
for i in 0.. $5$  {  
    // i will successively take values from 0 through 4  
}
```

See also the section “[Ranges, Intervals, and Strides](#)” on page 206 for more information.

Ternary Conditional Operator

Swift’s *ternary conditional operator* performs the same function as its syntactic counterpart in C. The basic format is as follows:

```
 $expr1$  ?  $expr2$  :  $expr3$ 
```

If $expr1$ evaluates to true, the operator returns $expr2$. Otherwise, it returns $expr3$.

This operator provides a shorthand equivalent of:

```
var a: Int  
if ( $someCondition$ ) {  
    a = 6  
} else {  
    a = 9  
}
```

reducing it to the following:

```
var a: Int =  $someCondition$  ? 6 : 9
```

Operator Precedence

When evaluating expressions that consist of more than a single operator, and where there are no parentheses to control evaluation order, Swift uses a simple set of rules to determine the order of evaluation. Let’s look at the following expression:

```
4 * 5 + 3
```

By convention, the multiplication is treated as a higher priority operation than the addition, and so the expression is evaluated to 23 and not 32.

Swift classifies the built-in operators as belonging to one of 11 groups and uses numeric *precedence* levels to determine overall evaluation order. Operators at higher levels are evaluated before operators at lower levels.

In addition, when two operators with the same precedence level are being evaluated, Swift uses predefined *associativity* values to determine which to evaluate first. Associativity values are declared as `none`, `left`, and `right`:

- A value of `left` means the lefthand subexpression will be evaluated first.
- A value of `right` means the righthand subexpression will be evaluated first.
- A value of `none` means that operators at this precedence level cannot be adjacent to each other.

Table 5 shows the precedence and associativity values for the built-in operators.

Table 5. Built-in operator precedence and associativity values

Precedence	Associativity	Operators
255	Left	~>
160	None	<<, >>
150	Left	*, /, %, &*, &
140	Left	+, -, &+, &-, , ^
135	None	..<, ...
131	Right	??
130	None	<, <=, >, >=, ==, !=, ===, !==, ~=
120	Left	&&
110	Left	
90	Right	*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Using [Table 5](#), you can see that in

```
4 << 5 * 4
```

the left-shift operator (<<) will be evaluated first because it has a higher precedence level than the multiply operator (*).

For an expression in which operands have the same precedence, the associativity values are applied. Consider the following expression:

```
4 + 3 &- 88
```

Both the addition operator (+) and the overflow subtraction operator (&-) are precedence level 140, but they are left associative, which means the lefthand subexpression is evaluated first, causing the expression to be interpreted as follows:

```
(4 + 3) &- 88
```

Strings and Characters

A `String` is an ordered collection of characters. The `Character` type is Unicode-compliant, so strings are also fully Unicode-compliant.

Empty string and character variables are declared as follows:

```
var astring: String
var achar: Character
```

Or they can be initialized by using a string literal value:

```
var astring: String = "Hello"
var achar: Character = "A"
```

Like `String` literals, `Character` literals are enclosed in double quotes. (Swift does not permit characters to be enclosed in single quotes, which might confuse C programmers.)

Because Swift can infer types, it is not necessary to include the `String` keyword when assigning a value, so you can also write the previous examples as follows:

```
var astring = "Hello"
var achar: Character = "A"
```

You can concatenate String types by using the + operator to create a new String:

```
let newString = "Hello" + " Bill"
```

Or you can append a String to an existing String by using the += operator:

```
var welcome = "Hello"  
welcome += " there"
```

String is a *value type* and instances are copied when assigned or passed to a function or method (unlike NSString, which is passed by *reference*).

String Properties

You can use the following features to check string length and get alternate views of the string in different character formats. See also “[Character-Level Access in Strings](#)” on page 43 for information on accessing the characters that make up the string.

someString.isEmpty

Boolean; true if the string contains no characters.

someString.characters

A view of the string’s Character storage that provides access to individual characters and iteration support via the String.Index type.

someString.characters.count

The number of characters in the string. Because Swift strings are Unicode-compliant, the number of characters might not be the same as the length of the string in bytes.

someString.lowercaseString

Returns the lowercase version of *someString*.

someString.uppercaseString

Returns the uppercase version of *someString*.

someString.utf8

A view of the string in UTF-8 format (of type `String.UTF8View`), for iterating over the string's characters in 8-bit format.

someString.utf16

A view of the string in UTF-16 format (of type `String.UTF16View`), for iterating over the string's characters in 16-bit format.

someString.unicodeScalars

A view of the string in `UnicodeScalar` format (of type `UnicodeScalarView`), for iterating over the string's characters in `UnicodeScalar` format.

Comparing Strings

You can compare strings and substrings by using the following comparison operators and methods:

`==`

Returns true if two strings contain the same sequence of characters.

`!=`

Returns true if two strings contain different sequences of characters.

`<`

Returns true if the string to the left of the operator sorts lexicographically before the string to the right of the operator.

`<=`

Returns true if the string to the left of the operator sorts lexicographically before or is equal to the string to the right of the operator.

`>`

Returns true if the string to the left of the operator sorts lexicographically after the string to the right of the operator.

`>=`

Returns true if the string to the left of the operator sorts lexicographically after or is equal to the string to the right of the operator.

`someString.hasPrefix(prefixString)`

Returns true if the sequence of characters in `prefixString` matches the start of `someString`.

`someString.hasSuffix(suffixString)`

Returns true if the sequence of characters in `suffixString` matches the end of `someString`.

NOTE

Note that Swift string and character comparisons are not locale-sensitive, but sort according to Unicode values (e.g., all uppercase letters sort before their lowercase variants). For locale-sensitive comparison and sorting, import Foundation and use locale-sensitive NSString methods.

Escaped Characters in Strings

To use certain special characters in string literals, use a backslash escape sequence:

`\0`

Null character

`\\`

Backslash

`\t`

Tab

`\n`

Line feed

`\r`

Carriage return

`\"`

Double quote

`\'`

Single quote

`\u{n}`

Arbitrary Unicode scalar; *n* is from 1 to 8 hex digits

String Interpolation

Expressions can be evaluated and the result substituted in a string literal using the escape sequence:

```
\(expr)
```

For example:

```
let costOfMeal = 56.80
let advice = "Consider tipping around \(costOfMeal * 0.20)"
```

String interpolation is not restricted to numeric values:

```
let a = "Hi"
let b = "there"
let c = "\(a) \(b)" // c is now "Hi there"
```

Numeric types can also be converted to strings by using the `String()` initializer, as in the following example:

```
let valueAsString = String(52.56)
// valueAsString now holds the string "52.56"
```

Converting Strings to Numeric Types

Strings that are meant to represent numeric values can be converted to numeric types using the same syntax that you would use when converting one numeric type to another—i.e., by treating the numeric type as a function, and using the string as an operand, as in the following examples:

```
let i = Int("45")
let d = Double("23.7")
```

Since the string may contain invalid characters that prevent its conversion to the desired type, the actual result is wrapped in an optional. See [“Optionals” on page 81](#) for more information.

Character-Level Access in Strings

In Swift, a `Character` is a single extended grapheme cluster—a sequence of one or more Unicode scalars that combine to represent a single character.

For example, the Unicode character U+308 (COMBINING DIAERESIS, or the umlaut) can be combined (in German) with the vowels “a,” “o,” or “u” to indicate a change in the way those vowels are pronounced, and also how they are displayed (as ä, ö, and ü).

The use of Unicode-combining characters can give rise to some apparent anomalies. Consider this sequence:

```
let aWithUmlaut: Character = "a\u{308}"
```

While it appears that two characters are being assigned to the character variable (which would not be possible) the combining feature of Unicode character U+308 means it and the character it combines with are treated as a single character, even though it is represented by multiple bytes.

Unicode characters aren't fixed-length entities, so character-level access to a string is not as simple as accessing bytes in an array, but access to each component of a string is still possible using a *view* of the string and its matching index type. Views are accessed as properties. To illustrate, consider this string assignment:

```
var uString = "a\u{308}"
```

- The `characters` view of this string provides access to each component of the string as an extended grapheme cluster (or `Character` type). In the preceding example, that is the single character ä.
- The `utf8` view provides access to each component of the string as a UTF-8 code unit. In this view, commonly used Roman characters are represented by a single byte, whereas multibyte sequences are used to represent combining characters and characters from non-Roman alphabets. In the preceding example, the character a would be represented by the 8-bit value 0x61, and the umlaut would be represented by the 8-bit value 0xCC followed by the 8-bit value 0x88.

- The `utf16` view provides access to each component of the string as a UTF-16 code unit. In this view, commonly used Roman characters are represented by a double-byte (16-bit) value. Combining characters and characters from non-Roman alphabets can be represented by a single 16-bit value, rather than multibyte values. In the preceding example, the character `a` would be represented by the 16-bit value `0x0061`, and the umlaut would be represented by the 16-bit value `0x0308`.
- The `unicodeScalars` view provides access to each component of the string as a unicode scalar value. In this view, combining characters are presented as separate components, and are not combined with the preceding character. In the preceding example, the scalars would consist of the character `a` represented by the 16-bit value `0x0061`, and the umlaut would be represented by the 16-bit value `0x0308`.

Each view has a `count` property that indicates how many components comprise that view. For the preceding example, the `count` property returns the following values for each of the views of the string:

```
uString.characters.count // 1 (ä)
uString.utf8.count       // 3 (0x61, 0xCC, 0x08)
uString.utf16.count      // 2 (0x0061, 0x0308)
uString.unicodeScalars.count // 2 (0x0061, 0x0308)
```

To iterate over all characters of a string, use a `for-in` loop. In the following example, the variable `i` will be of type `Character`, and will successively take on the value of each character from the string's `characters` view:

```
let str = "Swift"
for i in str.characters {
    print(i)
}
```

Individual components of any view can also be accessed using an appropriate index type as a subscript on that view:

- The `startIndex` property represents the position of the first element of the view. If the string is empty, this value is identical to the `endIndex` property.
- The `endIndex` property represents the “past the end” position of the view. This value cannot be used as a subscript on the view, but can be used in comparisons with other index values of the same type.

To iterate over the components of a specific view of a string, use an index and the `successor()`, `predecessor()`, or `advancedBy()` methods to modify that index. For example, this code iterates over the `utf8` view of the string, accessing each character as an 8-bit value:

```
let str = "Swift"
var idx = str.utf8.startIndex
while (idx != str.utf8.endIndex) {
    print (str.utf8[idx])
    idx = idx.successor()
}
// prints: 83, 119, 105, 102, 116,
```

To search the characters view for a specific character, use the `indexOf()` function, which returns an optional integer representing the position of the character, as follows:

```
let sought = Character("Y")
if let foundAt = "New York".characters.indexOf(sought) {
    print (sought, "found at position", foundAt)
}
// prints: Y found at position 4
```

See [“Optionals” on page 81](#) for more information.

String Inherited Functionality

Strings inherit from many standard Swift protocols, some of which include `Comparable`, `Equatable`, `Hashable`, `Streamable`, `OutputStream` Type, `StringInterpolationConvertible`, and

`StringLiteralConvertible` (see “Built-In Protocols” on page 175).

This provides them with many additional capabilities, including the following:

```
someString.append(character)
```

Appends *character* to *someString*.

```
someString.append(UnicodeScalar)
```

Appends *UnicodeScalar* to *someString*.

```
someString.appendContentsOf(anotherString)
```

Appends *anotherString* to *someString*.

```
someString.removeAll([keepCapacity: Bool])
```

Removes all characters from the string. The `keepCapacity` argument is optional and defaults to `false`. If set to `true`, the capacity of the string will remain unchanged.

```
someString.removeAtIndex(someIndex)
```

Removes and returns the item at the position indicated by *someIndex*. Will terminate with a runtime error if *someIndex* is not a valid index in the string’s characters view.

```
someString.removeRange(someRange)
```

Removes the items indicated by the range *someRange*. Will terminate with a runtime error if *someRange* is not a valid range within the string’s characters view.

```
someString.reserveCapacity(someInt)
```

If necessary, resizes the storage allocated to *someString* so it can store at least *someInt* characters.

Programs developed on iOS and OS X often implicitly import the Foundation framework, which provides additional extensions, such as bridging the `String` type to the `NSString` class. This means that much of the capability of `NSString` becomes available to `String` types. For example:

```
import Foundation
let message = "Hi there %name%, welcome to the party!"
```

```
let tag = "%name%"
let guest = "Lisa"
let greeting = message.stringByReplacingOccurrencesOfString(
    tag, withString: guest)
```

NOTE

The Foundation framework opens Swift up to a significant and substantial library of additional functionality. In addition to bridging `String` to `NSString`, Foundation bridges `Array` to `NSArray`, `Dictionary` to `NSDictionary`, and `Set` to `NSSet`. However, documenting Foundation is beyond the scope of this book.

Arrays

An array is a collection of items of the same type, be it a simple type (such as `Int`, `Double`, or `String`) or a more complex type (such as a class or structure).

The type of an array is formally specified as `Array<Type>`, although the more frequently used shorthand equivalent is `[Type]`. Thus, if you see the term `[String]`, you can conclude that it means an array of type `String`.

You declare arrays in a similar way as variables and constants. You create empty arrays as follows:

```
var arrayName = [Type]()
var daysOfWeek = [String]()
```

You can declare arrays with a specified number of pre-initialized entries:

```
var vertex = [Double](count: 10, repeatedValue: 0.0)
```

Or you can initialize them by using an *array literal*:

```
var locos: [String] = ["Puffing Billy", "Thomas"]
let daysPerMonth: [Int] =
    [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31, 31]
var primes = [1, 3, 5, 7, 11]
```

You can use the + operator to create an array that combines existing arrays of the same type, as shown here:

```
let vowels = ["A", "E", "I", "O", "U"]
let consonants = ["B", "C", "D", "F", "G", "H", "J" ...]
var allLetters = vowels + consonants
```

Here are some more characteristics of arrays in Swift:

- An array's type does not need to be specified if it is initialized from an array literal, because the type can be inferred.
- All entries in an array must be the same type (unlike Objective-C's NSArray class, which can store a collection of arbitrary objects).
- Array is a *value type* and instances are copied when assigned or passed to a function or method (unlike NSArray, which is passed by *reference*).

Accessing Array Elements

You access specific array elements by using C-style subscript syntax. The first element in an array has an index of 0:

```
let days = daysPerMonth[5]
```

You can access a subset of an array's elements by using a *range*. This operation returns a new *slice* (see “Slices” on page 56 for more information):

```
let daysPerNorthernSummerMonth = daysPerMonth[5..7]
```

If you attempt to access an element beyond the end of the array, a runtime error will occur.

`arrayName.first`

Returns the first element in the array.

`arrayName.last`

Returns the last element in the array.

`arrayName.maxElement()`

Returns the maximum element in the array.

`arrayName.minElement()`

Returns the minimum element in the array.

Array Properties

To examine the properties of an array, use the following features:

`arrayName.capacity`

Integer: the number of elements the array can store without it being relocated in memory.

`arrayName.count`

Integer: the number of elements in the array.

`arrayName.isEmpty`

Boolean: true, if the array has no elements.

Modifying a Mutable Array

You can modify mutable arrays in the following ways:

`arrayName.append(value)`

Adds a new element to the end of the array.

`arrayName += array`

Appends (copies) one array to the end of another.

`arrayName[n] = value`

Stores a value in element *n*, replacing the existing value there. A runtime error will occur if you attempt to write beyond the end of the array. To “grow” the array (i.e., add more entries), use the `append` method.

`arrayName[range] = array`

Replace a range of elements with an array of the same type. Ranges are specified as `[start...end]`. All elements from `[start]` to `[end]` are removed and replaced with copies of the elements in `array`. The size of the range does not have to be the same as the size of the array replacing it; the array will expand or contract to hold the replacement.

`arrayName.insert(value, atIndex: n)`

Insert a new value in front of element *n*.

`arrayName.removeAll([keepCapacity: Bool])`

Removes all elements from the array. The `keepCapacity` argument is optional and defaults to `false`. If set to `true`, the capacity of the array will remain unchanged.

`arrayName.removeAtIndex(n)`

Remove (and return) element *n* from the array.

`arrayName.removeLast()`

Remove (and return) the last element of the array.

`arrayName.reserveCapacity(n)`

Ensures the array has sufficient capacity to store *n* elements without further relocation, by relocating it if necessary.

`arrayName.sortInPlace()`

Sorts an array in place. Can be used with an optional trailing closure to define how two elements sort with respect to each other (e.g., `names.sortInPlace() { $1<$0 }`). (See also the section “[Closures](#)” on page 75.)

Iterating Over Arrays

To iterate over all elements in an array, use a `for-in` loop:

```
for item in arrayName {  
    ...  
}
```

Let’s take a closer look at how this works:

- The code in the braces is executed once for each item in the array.
- For each execution, *item* takes on the value of the next element, starting at the first element.
- *item* is a constant—although its value changes with each iteration, it cannot be modified in the loop.

To use both the position and value of items from the array, use the `enumerate()` method, as shown here:

```
for (index, item) in arrayName.enumerate() {  
    ...  
}
```

The `enumerate()` method returns a tuple consisting of the integer index and value of each item in the array.

Array Inherited Functionality

Arrays inherit from a number of standard Swift protocols, some of which include `ArrayLiteralConvertible`, `MutableSliceable`, `CustomStringConvertible`, `CollectionType`, `MutableCollectionType`, `Indexable`, `RangeReplaceableCollectionType`, and `SequenceType` (see [“Built-In Protocols” on page 175](#)). This provides them with many additional capabilities, some examples of which are provided here.

Much of the inherited functionality you can apply to arrays uses *closures*. These are anonymous functions that perform some operation on one or two elements of the array (such as a transform, or comparison). See the section [“Closures” on page 75](#) for more information.

The examples that follow are demonstrated using this array of strings:

```
var names = ["John", "Zoe", "Laura", "albert", "Allen"]
```

`arrayName.contains()`

Returns a `Bool` value that indicates if a specific value is contained in the array.

```
names.contains("John") // returns true  
names.contains("Lisa") // returns false
```

`arrayName.dropFirst([i])`

Returns an array slice that contains all but the first `i` elements of `arrayName`. If `i` is omitted, it defaults to 1. See [“Slices” on page 56](#).

```
names.dropFirst(2)  
// returns ["Laura", "albert", "Allen"] as a slice
```

`arrayName.dropLast([i])`

Returns an array slice that contains all but the last *i* elements of *arrayName*. If *i* is omitted, it defaults to 1. See “Slices” on page 56.

```
names.dropLast(2)
// returns ["John", "Zoe", "Laura"] as a slice
```

`arrayName.elementsEqual()`

Returns a Bool value that indicates if two arrays contain equal values in the same order.

```
let namesCopy = names
names.elementsEqual(namesCopy) // returns true
names.elementsEqual(namesCopy.sort()) // returns false
```

`arrayName.filter()`

Returns a new array that contains only the elements that match some condition, which is defined by using a closure. This example filters names longer than four characters:

```
names.filter { $0.characters.count > 4 }
// returns ["Laura", "albert", "Allen"]
```

`arrayName.flatMap()`

Returns a new array that is dimensionally flatter than *arrayName*, in which each element has been transformed by a mapping function that is defined by using a closure. This example returns an array in which all strings from the original array of arrays have been converted to uppercase characters:

```
let arrOfArrays = [["Bill", "Fred"], ["Mary"]]
let flatArray = arrOfArrays.flatMap()
    {$0.map{$0.uppercaseString}}
// flatArray holds ["BILL", "FRED", "MARY"]
```

`arrayName.forEach()`

Calls the body of the closure on each element of the array, producing similar functionality to `for i in arrayName { ... }`.

```
names.forEach { print($0) }
```

`arrayName.indexOf(someValue)`

Returns an optional integer representing the position of *someValue* in the array, or `nil` if the value is not present.

```
names.indexOf("John") // returns 4?
```

`arrayName.joinWithSeparator(someString)`

For an array of strings, returns a new string comprising the elements of *arrayName* interposed with *someString*.

```
names.joinWithSeparator("; ")
// returns "John; Zoe; Laura; albert; Allen"
```

`arrayName.map()`

Returns a new array in which each element has been transformed by a mapping function, which is defined by using a closure. This example returns an array in which any string from the original array that does not start with an uppercase “A” is prefixed with an asterisk (*):

```
names.map { $0.hasPrefix("A") ? $0 : "*" + $0 }
// returns:
// ["*John", "*Zoe", "*Laura", "*albert", "Allen"]
```

`arrayName.prefix(i)`

Returns an array slice that contains up to the first *i* elements of *arrayName*. See also “[Slices](#)” on page 56.

```
names.prefix(2)
// returns ["John", "Zoe"] as a slice
```

`arrayName.reduce()`

Returns a single value (of the type stored in the array) derived by recursively applying a reduction filter (defined by using a closure) to each element of the array and the output of the previous recursion. This example seeds the recursion with an empty string (`$0`) and concatenates each element of the array (`$1`) to the output of the previous recursion (`$0`):

```
names.reduce("") { $0 + $1 }
// returns "JohnZoeLauraalbertAllen"
```

`arrayName.reverse()`

Returns a new item of type `ReverseRandomAccessCollection<T>` that contains the elements of `arrayName` in reverse order. You can cast this back to an array with `as Array`:

```
names.reverse() as Array
// returns ["Allen", "albert", "Laura", "Zoe", "John"]
```

`arrayName.sort()`

Returns a new array that contains the elements of `arrayName` in sorted order. Use with an optional closure to define how two elements sort with respect to each other. For example:

```
names.sort() { $0<$1 }
// returns ["Allen", "John", "Laura", "Zoe", "albert"]
```

`arrayName.split(separator: [, maxSplit:][, allowEmptySlices:])`

Returns an array of array slices formed from the elements of `arrayName`, split at each occurrence of `separator`. `maxSplit` is optional, and defaults to `Int.max`. It specifies the maximum number of splits that will take place. Any remaining unsplit content will be included in the last slice. `allowEmptySlices` is optional and defaults to `false`. If set to `true`, an empty slice will be included in the returned array for each pair of consecutive elements that match `separator`.

```
let intArray = [5, 2, 0, 4, 5, 6, 0, 9, 0]
let aa7 = intArray.split(0, maxSplit: 1)
// returns [[5, 2], [4, 5, 6, 0, 9, 0]]
```

`arrayName.startsWith(anotherArray)`

Returns a `Bool` value that indicates if `arrayName` begins with the same elements as `anotherArray`, in the same order. For example:

```
names.startsWith(["John", "Zoe"]) // returns true
names.startsWith(["Allen"])      // returns false
```

```
arrayName.suffix(i)
```

Returns an array slice that contains up to the last *i* elements of *arrayName*. See also the next section, “Slices” on page 56.

```
names.suffix(2)
// returns ["albert", "Allen"] as a slice
```

Slices

A *slice* is a view into a subset of the elements of another collection (such as an array). Slices are space and time efficient—they do not initially copy the original collection’s elements, but refer to them *in-situ*. Copies will be made, however, of elements that are mutated.

You can create a slice from an array using a range operator, as in this example:

```
var names = ["John", "Zoe", "Laura", "albert", "Allen"]
var someNames = names[1...3]
// someNames is ["Zoe", "Laura", "albert"]
```

The new slice `someNames` is of type `ArraySlice<String>` and has the following characteristics:

- The `startIndex` property of the slice is 1, and the `endIndex` property is 4, since the slice is a view into the elements of the original array, in their positions in that array.
- If the source array is mutable, its elements can be modified and this will not affect the contents of the slice (this includes inserting and removing elements, not just mutating those from which the slice was initially constructed).
- If the slice is mutable, its elements can be updated and this will not affect the contents of the source array (this includes inserting and removing elements).

- So long as both the source array and slice are not mutated, indices of the slice and the source array can be used interchangeably.
- A slice is a view onto the elements of another collection, and will keep references to those elements even after the original collection goes out of scope, prolonging their life.

Dictionaries

Much like arrays, dictionaries store a collection of values, but whereas array elements are referenced via position, dictionary elements are referenced via unique *keys*.

A dictionary's type is formally specified as `Dictionary<KeyType, ValueType>`, although `[KeyType:ValueType]` is the preferred shorthand equivalent. Thus, if you see the term `[String:Int]`, you can assume it means a dictionary whose key type is `String` and whose value type is `Int`.

Dictionaries are declared in a similar way to variables and constants. You can create empty dictionaries like so:

```
var dictionaryName = [Type: Type]()  
var cpus = [String: String]()
```

Or you can initialize them upon declaration by using a *dictionary literal*, as demonstrated here:

```
var cpus: [String:String] =  
    ["BBC Model B": "6502", "Lisa": "68000", "TRS-80": "Z80"]
```

The type of both the key and the value can be inferred when initialized with a dictionary literal, so the previous example can be reduced to the following:

```
var cpus =  
    ["BBC Model B": "6502", "Lisa": "68000", "TRS-80": "Z80"]
```

If you're not initializing a dictionary with a literal value, you can specify a minimum capacity by using the following:

```
var dictionaryName = [Type: Type](minimumCapacity: Int)
var cpus = [String: String](minimumCapacity: 5)
```

Specifying a minimum capacity could be useful for improving the performance of dictionaries that are frequently mutated. Unlike arrays, you cannot determine the capacity of a dictionary, and you cannot reserve additional capacity after creation to improve performance.

Here are some more characteristics of dictionaries in Swift:

- All keys in a dictionary must be the same type.
- All values in a dictionary must be the same type.
- The contents of a dictionary are stored in arbitrary order.
- Dictionary is a *value type* and instances are copied when assigned or passed to a function or method (unlike `NSDictionary`, which is passed by *reference*).
- The key type used in a dictionary must conform to the `Hashable` protocol, which includes all of Swift's basic types such as `Int`, `String`, and `Double`. See [“Protocols” on page 162](#) for more information.

Accessing Dictionary Elements

To access dictionary values, you use the key as a subscript, as illustrated here:

```
let cpu = cpus["BBC Model B"]
```

Dictionary Properties

You can use the following features to access various properties of a dictionary:

`dictionaryName.count`

Integer: the number of key-value pairs in the dictionary.

`dictionaryName.isEmpty`

Boolean: true, if the dictionary has no elements.

`dictionaryName.keys`

Returns an array of all keys in the dictionary, which you can use for iterating over those keys (see [“Iterating Over Dictionaries” on page 60](#)). To use this in an array context, copy it to a new array. For example:

```
let newArrayName = [Type](dictionaryName.keys)
```

`dictionaryName.values`

Returns an array of all values in the dictionary, which can be used for iterating over those values (see [“Iterating Over Dictionaries” on page 60](#)). To use this in an array context, copy it to a new array. For example:

```
let newArrayName = [Type](dictionaryName.values)
```

Modifying a Mutable Dictionary

You can modify mutable dictionaries in the following ways:

`dictionaryName[key] = value`

Sets (or updates) the value of the element identified by *key*. To remove this key-value pair from the dictionary, set *value* to `nil`.

`dictionaryName.updateValue(newValue, forKey: key)`

Sets (or updates) the value of the element identified by *key*. Returns the old value as an *optional* (see [“Optionals” on page 81](#)) if there was one.

`dictionaryName.popFirst()`

Removes and returns the first key-value pair of *dictionaryName*, or returns `nil` if the dictionary was empty.

`dictionaryName.removeAll([keepCapacity: Bool])`

Removes all elements from the dictionary. The *keepCapacity* argument is optional, and defaults to `false`.

If set to true, the capacity of the dictionary will remain unchanged.

`dictionaryName.removeValueForKey(key)`

Removes a key-value pair from the dictionary identified by *key*. Returns the value that was removed or `nil` if there was no value for *key*.

Iterating Over Dictionaries

To iterate over all elements in a dictionary, you use a `for-in` loop, as follows:

```
for (key, value) in dictionaryName {  
    ...  
}
```

Let's take a closer look at how this works:

- The code inside the braces is executed once for each item in the array.
- For each execution, *key* and *value* take on successive key-value pairs from the dictionary.
- *key* and *value* are constant—although their values change with each iteration, they cannot be modified in the loop.

To iterate over just the keys or values of the dictionary, use the `keys` or `values` property, which returns an array:

```
for value in dictionaryName.values {  
    ...  
}  
  
for key in dictionaryName.keys {  
    ...  
}
```

Dictionary Inherited Functionality

Dictionaries inherit from a number of standard Swift protocols, some of which include `CollectionType`, `SequenceType`,

Indexable, DictionaryLiteralConvertible, and CustomStringConvertible (see “Built-In Protocols” on page 175).

Since dictionaries inherit from the Indexable and SequenceType protocols, their elements can be accessed by index, but since dictionary contents are stored in arbitrary order, indexed access is less useful than it is for arrays. Dictionary indices are specified using DictionaryIndex values, not integer positions.

dictionaryName.dropFirst([i])

Returns a dictionary slice that contains all but the first *i* elements of *dictionaryName*. If *i* is omitted, it defaults to 1. See “Slices” on page 56.

dictionaryName.dropLast([i])

Returns a dictionary slice that contains all but the last *i* elements of *dictionaryName*. If *i* is omitted, it defaults to 1. See “Slices” on page 56.

dictionaryName.endIndex

Returns the index value that represents the “past the end” position of *dictionaryName*. The returned value cannot be used as a subscript on the dictionary, but can be used in comparisons with other DictionaryIndex values.

dictionaryName.forEach()

Calls the body of a closure on each key-value pair of the dictionary, producing similar functionality to `for i in dictionaryName { ... }`. The key-value pair is available in `$0` as a tuple.

dictionaryName.indexForKey(keyValue)

Returns an optional DictionaryIndex value that represents the position of the element with the specified key, or `nil` if the key is not present.

dictionaryName.removeAtIndex(someIndex)

Removes and returns the key-value pair at the position indicated by *someIndex*. Will cause a runtime error if the dictionary is empty or the index is not valid.

`dictionaryName.startIndex`

Returns the index value that represents the position of the first element of `dictionaryName`, assuming it is not empty. For an empty dictionary, this value is identical to that returned by `endIndex`.

Sets

A set is an unordered collection of unique values of the same type, which can be a simple type (such as `Int`, `Double`, or `String`) or a more complex type (such as a class or structure). Sets were first introduced in Swift 1.2.

The type of a set is formally specified as `Set<Type>`. There is no shorthand equivalent.

You declare sets in a similar way to variables and constants. You create empty sets as follows:

```
var setName = Set<Type>()
var daysOfWeek = Set<String>()
```

You can declare an empty set with space for a specified minimum number of members:

```
var vowels = Set<Character>(minimumCapacity: 5)
```

You can initialize a set using an array literal:

```
var chessPieces: Set =
    ["King", "Queen", "Rook", "Bishop", "Knight", "Pawn"]
```

Here are some more characteristics of sets in Swift:

- A set's type does not need to be specified if it is initialized from an array literal, because the type can be inferred.
- All items in a set must be the same type.
- You can compare two sets of the same item type with the `==` and `!=` operators. Sets are considered equal if they contain the same items.

- All items in a set are unique. If you try to add an item to a set that already exists in that set, the set will still only hold one instance of that item.
- Set is a *value type* and instances are copied when assigned or passed to a function or method (unlike `NSSet`, which is passed by *reference*).
- The type used for a set must conform to the Hashable protocol, which includes all of Swift’s basic types such as `Int`, `String`, and `Double`. See “Protocols” on page 162 for more information.

Accessing Set Items and Properties

You can use the following features to access the set and its properties:

`someSet.contains(someValue)`

Returns `true`, if `someValue` is a member of the set.

`someSet.count`

Returns the number of items in the set as an integer.

`someSet.isEmpty`

Returns `true` if the set contains no items.

Modifying a Mutable Set

You can modify mutable sets in the following ways:

`someSet.insert(newValue)`

Inserts a new item `newValue` into `someSet`. If the item is already present in the set, this has no effect.

`setA.exclusiveOrInPlace(setB)`

Replaces the contents of `setA` with the items that are unique to `setA` and unique to `setB` (the “opposite” of the intersection of the sets).

`setA.intersectionInPlace(setB)`

Removes from `setA` all items that are not also present in `setB` (`setA` becomes the intersection of `setA` and `setB`).

`someSet.remove(someValue)`

Removes and returns the item `someValue` in `someSet`, or returns `nil` if the item is not present.

`someSet.removeAll([keepCapacity: Bool])`

Removes all items from `someSet`. The `keepCapacity` argument is optional, and defaults to `false`. If set to `true`, the capacity of the set will remain unchanged.

`someSet.removeFirst()`

Removes and returns the first item in `someSet`. If the set is empty, this will cause a runtime error.

`setA.subtractInPlace(setB)`

Removes any items that exist in both `setA` and `setB` from `setA`.

`setA.unionInPlace(setB)`

Inserts into `setA` all of the items in `setB` that are not already present in `setA`.

Iterating Over Sets

To iterate over all items in a set, use a `for-in` loop:

```
for item in someSet {  
    ...  
}
```

Let's take a closer look at how this works:

- The code in the braces is executed once for each item in the array.
- For each execution, `item` takes on the value of the next item in the set, starting at the first element.

- *item* is a constant—although its value changes with each iteration, it cannot be modified in the loop.

Since the order of items stored in a set is arbitrary, use the `sort()` method to return an ordered sequence of items:

```
for item in someSet.sort() {  
    ...  
}
```

Set Operations

The following operations can be performed on sets:

`setA.exclusiveOr(setB)`

Returns a new set that contains only the items that are unique to *setA* and unique to *setB* (the “opposite” of the intersection of the sets).

`setA.intersect(setB)`

Returns a new set that contains the items that are common to both *setA* and *setB*.

`setA.isDisjointWith(setB)`

Returns a `Bool` value that indicates if the intersection of *setA* and *setB* is an empty set (i.e., that they do not have any elements in common).

`setA.isStrictSubsetOf(setB)`

Returns a `Bool` value that indicates if all of the items in *setA* are also present in *setB*, and that *setB* contains elements that are not present in *setA*.

`setA.isStrictSupersetOf(setB)`

Returns a `Bool` value that indicates if all of the items in *setB* are also present in *setA*, and that *setA* contains elements that are not present in *setB*.

`setA.isSubsetOf(setB)`

Returns a `Bool` value that indicates if all of the items in *setA* are also present in *setB*.

`setA.isSupersetOf(setB)`

Returns a Bool value that indicates if all of the items in `setB` are also present in `setA`.

`setA.subtract(setB)`

Returns a new set that contains only the items in `setA` that are not present in `setB`.

`setA.union(setB)`

Returns a new set that contains all of the items in `setA` and all of the items in `setB`.

Set Inherited Functionality

Sets inherit from a number of standard Swift protocols, including `ArrayLiteralConvertible`, `CollectionType`, `CustomStringConvertible`, `Equatable`, `Hashable`, `Indexable`, and `SequenceType` (see “[Built-In Protocols](#)” on page 175).

Since sets inherit from the `Indexable` and `SequenceType` protocols, their elements can be accessed by index. Set indices are specified using `SetIndex` values, not integer positions.

`someSet.endIndex`

Returns the index value that represents the “past the end” position of `someSet`. The returned value cannot be used as a subscript on the set, but can be used in comparisons with other `SetIndex` values.

`someSet.forEach()`

Calls the body of a closure on each member of the set, producing similar functionality to `for i in setName { ... }`.

`someSet.indexOf(someItem)`

Returns the index value of `someItem`, or `nil` if the item is not present in `someSet`.

`someSet.removeAtIndex(someIndex)`

Removes and returns the item at the position indicated by `someIndex`, or returns `nil` if the set is empty.

`someSet.startIndex`

Returns the index value that represents the position of the first element of `someSet`, assuming it is not empty. For an empty set, this value is identical to `.endIndex`.

Option Sets

A common way to represent a set of related features or characteristics in programming is to establish a single (possibly unsigned) integer value for storage, and use individual bits in that value to represent each feature or characteristic.

For example, a single integer could be used to represent multiple stylistic variants affecting the way a piece of text is to be rendered. Setting the least significant bit to 1 might mean the text should be displayed in bold. Setting the next most significant bit to 1 might mean the text should be displayed in italics. If both of these bits were set to 1, it would mean the text should be displayed in bold italics.

This approach is used in a number of C and Cocoa APIs. Determining the status of individual bits typically requires a bitwise AND operation. Setting a single bit requires a bitwise OR operation, and clearing a single bit requires a bitwise AND with an appropriate mask value. All such operations require a good understanding of Boolean logic and binary values.

Option sets were introduced in Swift 2.0 and bring standard set operations to bit-level values that are used in this way.

An option set to implement the simple text-style example just outlined would be defined as follows:

```
struct TextStyle: OptionSetType {
    let rawValue: UInt8

    static let Bold      = TextStyle(rawValue: 1)
    static let Italic    = TextStyle(rawValue: 2)
    static let Underline = TextStyle(rawValue: 4)
    static let Outline   = TextStyle(rawValue: 8)
}
```

So defined, simple set operations can now be used to set and reset bits in an instance of `TextStyle`, as follows:

```
var style: TextStyle = [] // clear all bits
style = [.Bold, .Italic]
style.insert(.Underline)
style.remove(.Bold)
if style.contains(.Outline) {
    ...
}
```

Functions

You declare functions in Swift by using the `func` keyword, as shown in the following:

```
func functionName(parameters) -> returnType
{
    // function body
}
```

Here are some characteristics of and usage tips for Swift functions:

- Functions can have zero or more parameters.
- If there are no parameters, empty parentheses must still be provided.
- Functions do not have to return a value, in which case you omit the arrow and *returnType*.

Parameter Types

By default, function parameters are constant (they cannot be modified in the function body). *Variable* parameters are created by preceding them in the function declaration with the `var` keyword, as shown here:

```
func someFunc(var s: someType) -> ...
```

You can use the variable *s* as a local, modifiable variable in the function body. Variable parameters are lost after the function returns—you cannot use them to pass values outside the function body.

In-out parameters are created by preceding them in the function declaration with the `inout` keyword, like so:

```
func someFunc(inout i: someType) -> ...
```

i becomes an alias for an external variable passed by reference. Modifying *i* inside the function modifies the external variable. To call such a function, you must place an ampersand (&) before a referenced variable's name in a function call:

```
var i: Int = 45
someFunc(&i)
```

Returning Optional Values

A function can return an *optional* value, which is a way to indicate that no valid return value can be provided. Suppose that you were implementing a (pointless!) function to do division. The function definition might start out like this:

```
func divide(dividend: Double, _ divisor: Double) -> Double
{
    return dividend / divisor
}
```

It is possible that the divisor might be zero, which would cause a runtime error. With an optional return value, you can indicate when the result is valid and when it is not. To specify that a return value is optional, follow it with a question mark like this:

```
func divide(dividend: Double, _ divisor: Double) -> Double?
{
    if (divisor == 0) { return nil }
    return dividend / divisor
}
```

If the divisor is zero, a `nil` value is returned; otherwise, the result of the division is returned.

Because the return value is now an optional, you need to test it before using it. To do that, use this syntax:

```
var d = division (9.0, 0.0)
if d != nil {
    // value is valid
    print (d!)
} else {
    // value is invalid
}
```

Alternatively, you can use *let binding* to test and use values from functions that return optionals. Let binding combines a local assignment with a non-nil test, as in this example:

```
if let d2 = division(9.0, 3.0) {
    print (d2)
}
```

See the sections “[Optional Binding](#)” on page 84 and “[Optional Chaining](#)” on page 85 for more information.

NOTE

Generally, a function that needs to indicate an error has occurred should throw the error using the new Swift 2.0 error handling features, as described in “[Error Handling](#)” on page 103.

Returning Multiple Values by using Tuples

You can have a function return more than one value by using a tuple, as in the following example:

```
func getRange() -> (lower: Int, upper: Int)
{
    ...
    return (someLowValue, someHighValue)
}
```

Because the tuple members are named in the function declaration in the preceding example, those names can be used to refer

to the components after the function call. Thus, you could access the two values like this:

```
let limits = getRange()
for i in limits.lower...limits.upper { ... }
```

The optional tuple return type

For cases in which a tuple is the return type of a function, you might want to indicate the tuple has no value. Extending the previous `getRange()` example, it might be the case that no valid range exists, and thus there is nothing to return. This can be managed by using an optional return type for the tuple, which you indicate by following the parentheses around the return type with a question mark, and returning `nil` (instead of a tuple) if the range is not valid:

```
func getOptionalRange() -> (lower: Int, upper: Int)?
{
    if (rangeIsValid) { return nil }
    ...
    return (someLowValue, someHighValue)
}
```

For more information, see the section “Optionals” on page 81.

Local and External Parameter Names

Parameter names such as `p1` in the following example act as a *local* parameter name that is used inside the function body:

```
func f(p1: Int) { ... }
```

A caller of that function provides a single integer value as the parameter value:

```
let a = f(45)
```

Where a function is defined with more than one parameter, the second and subsequent parameter names must be used in calls to that function. For example:

```
func addString(s: String, toString: String) -> String
{ ... }
```

must be called as follows:

```
let s = addString(someString, toString: someOtherString)
```

Note that the name of the first parameter is not provided in the function call, but the name of the second parameter is.

Formally, each parameter in a function definition has two names—a *local* name, used inside the function definition to reference the value, and an *external* name, which must be specified in calls to the function. Thus, every parameter in a function definition can be expressed as *externalName local Name: type*.

- For the first parameter, if an external name is not provided, then it defaults to `_` (underscore).
- For any parameter except the first, if an external name is not provided, then it defaults to the local name.
- For any parameter, if the external name is declared, then it must be used in calls to the function.
- If an external name is defined as `_` (underscore), then that parameter name *cannot* be used in calls to the function.
- Swift’s named parameter feature does not allow for parameters to be listed in “arbitrary order” when calling a function—they must be listed in the order that they are defined.

Default Parameter Values

To specify a default value for a parameter to a function, follow the parameter’s type specification with an assignment, as in the following example:

```
func addAnotherString (s: String,  
    toString: String = "pine") -> String  
{ ... }
```

If the `toString` parameter is omitted in a call to this function, such as in the example that follows, the default value of “pine” will be used for that parameter:

```
let s3 = addAnotherString("cone")
```

Parameters with default values should be defined after all other parameters in the function definition.

Variadic Parameters

A *variadic* parameter supports a variable number of input values. You specify a variadic parameter by following the parameter type with an ellipsis (...), as shown here:

```
func someFunc(param: Type...) -> returnType  
{ ... }
```

In the body of the function, the multiple values passed as parameters to the function are available in an array, as shown in the following example:

```
func sumOfInts(numbers: Int...) -> Int  
{  
    var tot = 0  
    for i in numbers { tot += i }  
    return tot  
}
```

The function would then be called as follows:

```
sumOfInts(2, 3) // returns 5  
sumOfInts(5, 9, 11, 13, 22) // returns 60
```

When using variadic parameters, note the following:

- A function can only have one variadic parameter, but it can appear anywhere in the parameter list (this is a change from Swift 1.0, where a variadic parameter had to be the last parameter in the list).
- If the variadic parameter is not the last in the list of parameters, then all subsequent parameters must have external parameter names.

Function Types

A function's type is an expression of the types of its input parameters and its result. For example, for:

```
func sumOfInts(numbers: Int...) -> Int {...}
func search(string s: String, forString s2: String)
    -> Int {...}
func doesNothing() {...}
```

the types are, respectively:

```
(Int...) -> Int
(String, String) -> Int
() -> ()
```

You can use function types in many places where you can use simpler types (such as `Int`). For example, you can declare a variable or constant to be a function type, as shown here:

```
var generalFunc: (Int) -> Int
```

You can then assign to the variable `generalFunc` a function of the same type:

```
func addOne (i: Int) -> Int { return i+1 }
func addTwo (i: Int) -> Int { return i+2 }
func times8 (i: Int) -> Int { return i*8 }
generalFunc = addOne
```

That variable can then be used where the function could be used:

```
addOne(4) // returns 5
generalFunc(5) // returns 6
generalFunc = addTwo
generalFunc(5) // returns 7
```

You can pass functions as parameters to other functions, and functions can be returned by other functions. You specify the function type as either the type of the parameter or the type of the returned value.

The example that follows defines a function that takes two parameters and then returns an integer value. The first parameter is a function type that takes an integer parameter

and returns an integer value. The second parameter is an integer value:

```
func adaptable(inputFunc: (Int)->Int, _ p: Int) -> Int
{
    return inputFunc(p)
}

adaptable(addOne, 4) // returns 5
adaptable(times8, 5) // returns 40
```

The next example defines a function that takes a single integer parameter and then returns a function. The returned function is defined as taking a single integer parameter and returning a single integer parameter:

```
func selectOperation(i: Int) -> (Int) -> Int { ... }
```

Closures

Closures are functionally similar to *blocks* in Objective-C and *lambdas* in languages such as Scheme, C#, and Python.

Closures are anonymous functions that can be passed as arguments to other functions (known as *higher-order functions*) or returned by other functions. They can refer to variables and parameters in the scope in which they are defined (sometimes called *outer variables*). In doing so, they are said to capture or *close over* those values.

You typically define a closure through a *closure expression*, which takes the following format:

```
{
    (parameters) -> returnType in
    statements
}
```

To gain a better understanding of where closures can be useful, consider the operation of sorting arrays. The C standard library provides a number of sorting functions, one of which is `qsort()`, which takes as a parameter a pointer to a comparison function. This comparison function, defined by the caller, takes as parameters two pointers to two entities that are to be com-

pared. The comparison function returns an integer that is less than, equal to, or greater than zero depending on whether the first entity is less than, equal to, or greater than the second.

A closure is a concise way of providing similar functionality without having to define a named function to do the comparison. Instead, the closure is passed as an inline parameter to the sort function.

The Array type implements a `sort()` method that creates a sorted copy of an array. You use it as follows:

```
let names = ["John", "Zoe", "Laura", "albert", "Allen"]
let s = names.sort()
// s is now ["Allen", "John", "Laura", "Zoe", "albert"]
```

Swift knows how to compare built-in types such as `String`, `Int`, and `Float`; therefore, it can sort arrays of these types into ascending or lexical order. In the preceding example, `sort()` applies its default behavior for the `String` type and sorts lexically (“albert” sorts after “Zoe” because lowercase characters sort lexically after uppercase characters).

There exists another version of `sort()` that takes a closure as a parameter. The closure takes two values (of the same type as the array’s elements) and must return `true` if the first value should sort before the second (much like the comparison function required by `qsort()` described earlier). By way of example, if the array being sorted contains strings, the closure would be defined as follows:

```
(String, String) -> Bool
```

In other words, the closure must take two `String` parameters and return a `Bool` value that indicates whether the first string sorts before the second.

This is how you would call `sort()` and provide a closure that replicates the behavior of the simpler version already described:

```
let t1 = names.sort(
    { (s1: String, s2: String) -> Bool in
        return s1<s2 }
)
```

Because Swift can infer types from context, you can usually omit them. The array is an array of strings, so it follows that the two closure parameters must also be of type `String`. The closure must return a `Bool`. Because all of the types can be inferred, they can be omitted. And as there are now no types to specify, you can also omit the parentheses and the arrow. Thus, you can reduce the closure to this:

```
let t2 = names.sort( { s1, s2 in return s1<s2 } )
```

For simple closures with a single expression, such as that just demonstrated, you can also omit the `return` keyword, which reduces the closure further to the following:

```
let t3 = names.sort( { s1, s2 in s1<s2 } )
```

To produce a reversed variant of the sort, switch the order of the strings being compared. This effectively returns `false` if the first value should sort before the second:

```
let t4 = names.sort( { s1, s2 in s2<s1 } )  
// t4 is ["albert", "Zoe", "Laura", "John", "Allen"]
```

Alternatively, reverse the comparison operator, as shown here:

```
let t5 = names.sort( { s1, s2 in s1>=s2 } )  
// t5 is now ["albert", "Zoe", "Laura", "John", "Allen"]
```

To sort by string length instead of lexically, modify the comparison operands in the closure to compare the lengths of the two strings being compared:

```
let t6 = names.sort( { s1, s2 in  
    s1.characters.count < s2.characters.count } )  
// t6 is now ["Zoe", "John", "Laura", "Allen", "albert"]
```

Automatic Argument Names

In the discussion of closures, parameter names were defined to refer to each argument required by the closure. For example, when sorting an array of strings, the arguments were named `s1` and `s2` so the parameters could be referenced in the comparison expression:

```
let t2 = names.sort( { s1, s2 in return s1<s2 } )
```

For simple inline closures, having to first name the arguments just so you can subsequently refer to them makes the closure longer than it needs to be. For inline closures, Swift assigns *automatic argument names* to each parameter by using a dollar sign followed by a position number (\$0, \$1, \$2, etc.).

Recall from earlier that (for sorting a string array) the closure required by the `sort()` method is defined as follows:

```
(String, String) -> Bool
```

There are two string parameters and Swift aliases their arguments as \$0 and \$1. Using these aliases means you don't have to define them yourself, and the `sort()` closure examples reduce further still to:

```
let u1 = names.sort( { $0 < $1 } )
let u2 = names.sort( { $1 < $0 } )
let u3 = names.sort(
    { $0.characters.count < $1.characters.count } )
```

Trailing Closures

When the last (or only) argument provided to a function is a closure, you can write it as a *trailing closure*. Trailing closures are written after the parentheses that wrap the function's arguments:

```
let v1 = names.sort() { $0 < $1 }
let v2 = names.sort() { $1 < $0 }
let v3 = names.sort()
    { $0.characters.count < $1.characters.count }
```

If the function has no other arguments than the closure itself, and you're using trailing closure syntax, you can omit the empty parentheses, reducing the closure to the simplest variant:

```
let w1 = names.sort { $0 < $1 }
let w2 = names.sort { $1 > $0 }
let w3 = names.sort
    { $0.characters.count < $1.characters.count }
```

Capturing Values

As with any regular function, closures are able to refer to the state of the scope in which they are defined (e.g., local variables or constants defined in the same scope). But, like functions, closures can be returned by their containing function, which means a closure could be executed after the values it refers to have gone out of scope.

This situation does not result in a runtime error. The closure is said to *capture* those values, and extend their lifetime beyond the scope in which they are defined.

In the example that follows, a function (`makeTranslator`) creates new functions (closures) and returns them as its result. It takes a single parameter (a string) with the local name `greeting`. The function it returns takes a single parameter (a string) and returns a single parameter (a string):

```
func makeTranslator(greeting: String) -> (String) -> String
{
    return {
        (name: String) -> String in
            return (greeting + " " + name)
    }
}
```

The closures that are built by this function capture the `greeting` string value and use it later whenever they are executed, even though that value has since gone out of scope.

Here is how you might use this function:

```
var englishWelcome = makeTranslator("Hello")
var germanWelcome = makeTranslator("Guten Tag")
```

After this has been executed, `englishWelcome` will refer to a closure that takes a single string argument and will return it with the word “Hello” prepended, whereas `germanWelcome` will refer to a closure that takes a single string argument and will return it with the words “Guten Tag” prepended.

Because `englishWelcome` and `germanWelcome` refer to closures, and closures are functions, you call them in the same manner you call any function:

```
englishWelcome ("Bill")
// returns "Hello Bill"
germanWelcome ("Johan")
// returns "Guten Tag Johan"
```

The closures and the values they have captured will remain available until the variables that refer to them go out of scope or are set to new values. For example, if you change the definition of `englishWelcome` like this:

```
englishWelcome = makeTranslator("G'day")
englishWelcome ("Bruce")
// returns "G'day Bruce"
```

then the storage allocated to the “Hello” version of the closure and its captured values will be released.

Capturing Values by Reference

In the preceding discussion, the value captured (the greeting string value) is actually copied when the closure is constructed, because that value is never modified by the closure.

Values a closure modifies are not copied but are instead captured by *reference*. Here’s a revised example that keeps count of the number of times it has been called:

```
func makeCountingTranslator(greeting: String,
    _ personNo: String) -> (String) -> String
{
    var count = 0

    return {
        (name: String) -> String in
            count++
            return (greeting + " " + name + ", " +
                personNo + " \{(count)\}")
    }
}
```

Next, construct two new closures to make greetings:

```
var germanReception =  
    makeCountingTranslator("Guten Tag", "Sie sind Nummer")  
var aussieReception =  
    makeCountingTranslator("G'day", "you're number")
```

And then call them:

```
germanReception ("Johan")  
// returns "Guten Tag Johan, Sie sind Nummer 1"  
aussieReception ("Bruce")  
// returns "G'day Bruce, you're number 1"  
aussieReception ("Kylie")  
// returns "G'day Kylie, you're number 2"
```

Each closure stores a reference to `count`, which is a local variable in the `makeCountingTranslator()` function. In doing so, they extend the lifetime of that local variable to the lifetime of the closure.

Note that each closure still gets its own instance of `count` because they existed as two different instances, separated by the two executions of `makeCountingTranslator()`.

Optionals

Swift's *optionals* provide a way to indicate a value exists without usurping some part of the value's set of possible states to do so.

For example, an application might want to record if a piece of string is present in the physical world, and if so, record what the length of that piece of string is. In this example, a negative value (such as `-1`) could be used to indicate the string is not present, because such a value could never represent an actual length. This example uses a single store to represent whether the string is present and (only if it is) what its length is.

A similar technique is often used in Objective-C, where objects (or, more precisely, *pointers* to objects) may be `nil`, indicating there is no object. Many Objective-C method calls return either (a pointer to) an object or `nil` if the method call has failed or some other error has occurred.

In Swift, object references are not pointers and may not normally be set to `nil`, unless they are explicitly declared to be optional values. An example of the syntax for such a declaration is as follows:

```
var str: String?
```

The question mark, which immediately follows the type, declares that the variable `str` is an optional. Its value might exist or it might not. Not having a value is not the same as `str` storing an empty string. (When a new optional is created in this way, its initial value is set to `nil`).

When a variable has been declared to be optional, it must either be used in places where an optional context for that type is allowed, or it must be unwrapped (see the section “[Unwrapping Optionals](#)” on page 82) to reveal the underlying value.

For example, you can assign an optional to another optional without issue, as shown here:

```
var n: String?
n = str
```

However, you cannot assign it to a nonoptional:

```
var r: String
r = str // will produce a compile-time error
```

Because `r` is not an optional, the compiler won’t allow an optional to be assigned to it.

If an optional’s value exists, assign `nil` to it to remove that value:

```
str = nil
```

Only optionals can be assigned `nil` in this way. Attempting to assign `nil` to a nonoptional variable or constant will result in a compile-time error.

Unwrapping Optionals

To access the value stored by an optional, first check if a value exists with an `if` statement. If it does exist, use the exclamation

mark to *force unwrap* the optional and access the raw value it stores, as demonstrated here:

```
if str != nil { // check if the optional has a value
    r = str!    // it does - unwrap it and copy it
} else {
    // the optional had no value
}
```

- Force unwrapping an optional for which no value exists will result in a runtime error.
- The more usual way to unwrap optionals is to use *optional binding*, as described later in this section.

Implicitly Unwrapped Optionals

In some situations, it might be appropriate to use an optional, even if it will always have a value. For example, an optional created by using `let` cannot be mutated (so it cannot be reset to `nil`), and it must be initialized when it is declared, so it can never not have a value:

```
let constantString: String? = "Hello"
```

Even though its value cannot change, the value must be still unwrapped to use it in a nonoptional context:

```
var mutableString: String
mutableString = constantString // compile-time error
mutableString = constantString! // allowed
```

For this and other uses, Swift provides *implicitly unwrapped optionals*, which are defined by using an exclamation mark after the type instead of a question mark. After it is defined, a reference to the optional's value does not need to be unwrapped; it is implicitly unwrapped whenever it is referenced:

```
let constantString: String! = "Hello"
mutableString = constantString
```

This example is contrived, but implicitly unwrapped optionals play a role during class initialization. See the section [“Classes” on page 106](#) for more information.

Optional Binding

Optional binding is a way to test whether an optional has a value, and, if it does, make a scoped non-optional copy, all in one operation.

You can use optional binding with the `if` statement (where the scoped non-optional copy is only valid inside the first set of braces) or the guard statement (where the scoped non-optional copy is valid for the remainder of the scope in which it was created).

Following is the syntax for optional binding with `if`:

```
if let aConst = someOptional {  
    // aConst is now an unwrapped version of someOptional  
    print (aConst)  
}  
// aConst is out of scope at this point
```

Assuming that *someOptional* has a value, *aConst* holds a non-optional copy of that value inside the first set of braces of the `if` statement. Because *aConst* is not an optional, its value can be used directly—the unwrapping has been handled in the `let` statement.

Similarly, here is the syntax for optional binding with `guard`:

```
guard let aConst = someOptional else {  
    // aConst is not valid here - exit this scope  
    return  
}  
// aConst is valid in this scope
```

With `guard`, if the optional can be unwrapped, then the binding remains in place until the scope that contains the guard statement terminates.

Some other points about optional binding include:

- You can use `var` instead of `let` to create a mutable unwrapped copy of the optional.

- You can add a `where` clause immediately before the `else` clause to further constrain whether the optional is unwrapped.
- You can use *compound bindings*—multiple comma-separated `let` or `var` assignments—to unwrap multiple optionals at the one time. If any of the bindings fail, the compound binding fails.

Optional Chaining

When you access an optional, it either has a value or is `nil`, and you need to test that the value exists before unwrapping it, as in the following example:

```
var s: String?  
  
if s { // check if the optional has a value  
    var r = s! // it does - do something with it  
} else {  
    // the optional had no value  
}
```

You can use optionals anywhere a value might or might not exist:

- A property of a class, structure, or enumeration might hold an optional value.
- A method of a class, structure, or enumeration might return an optional value.
- A subscript of a class, structure, or enumeration might return an optional value.

Optional chaining is a facility by which you can query an optional, or something that depends on an optional having a value, without specifically having to test the optional first. You can use optional chaining when accessing class, structure, or enumeration properties, methods, or subscripts using dot-syntax.

Consider this simple example of two classes, in which class A contains an optional reference to an instance of class B:

```
class A
{
    var otherClass: B?
}

class B
{
    var someProperty: Int = 7
    func someMethod()
    {
        print ("someMethod called!")
    }
    subscript (index: Int) -> String
    {
        get {
            print ("getter for [\\(index)] called")
            return "sample string"
        }
        set { print ("setter for [\\(index)] called") }
    }
}
```

Now, assume you have an optional reference to an instance of class A, as in the following example:

```
var a: A?
```

Let's further assume you want to follow the path from the optional reference `a` through to `someProperty` or `someMethod()` of class B. Without optional chaining, you would need to check that each optional has a value; if it does (and only if it does), could you descend down to the next level, like this:

```
if a != nil {
    if a!.otherClass != nil {
        print (a!.otherClass!.someProperty)
    } else {
        print ("no property available")
    }
} else {
    print ("no property available")
}
```

This leads to potentially deep conditional tests, which is what optional chaining simplifies. With optional chaining (and let binding), you can reduce the code to the following:

```
if let p = a?.otherClass?.someProperty {
    print (p)
} else {
    print ("no property available")
}
```

If any optional in the chain returns a `nil` value, the entire expression returns `nil`.

The use of optional chaining isn't restricted to reading property values; you can also write them like this:

```
a?.otherClass?.someProperty = 6
```

The statement will return `nil` if the assignment failed because some part of the optional chain returned `nil`, which can be tested like this:

```
if (a?.otherClass?.someProperty = 6) == nil {
    // unable to write the property
}
```

You can call methods using optional chaining, as follows:

```
a?.otherClass?.someMethod()
```

Again, the method call will return `nil` if the call failed because some part of the chain returned `nil`. Even if the method normally returns a nonoptional value, it will always be returned as an optional when used in an optional chain context.

You can also use optional chaining with subscripts:

```
a?.otherClass?[1]
// returns nil, or prints "getter for [1] called"

a?.otherClass?[3] = "Optional chaining is neat"
// returns nil if the assignment fails
```

Program Flow

Swift includes the usual collection of loops and conditional execution features. Most of these are superficially the same as their counterparts in C-like languages, but in some cases (e.g., the `switch` statement) they offer considerably expanded and safer functionality.

Loops

Swift provides the standard loop constructs you would expect, including `for`, `while`, and `repeat-while` loop variants.

for-condition-increment loops

The `for-condition-increment` loop is functionally the same as the `for` loop in C. The loop consists of an initialization phase, a test, an increment, and a set of statements that are executed for each iteration of the loop. Here's an example:

```
for initialization; condition; increment {  
    statements  
}
```

The three phases work as follows:

- The initialization phase sets up the conditions for the start of the loop (typically, initializing a loop counter).
- The condition tests whether the loop's termination condition has been met—whenever this evaluates to `true`, the statements in the body of the loop are executed once.
- The increment phase adjusts some variable or value that forms part of the condition test to ensure a stopping condition can be reached (typically, incrementing the loop counter).

It is possible that the statements in the body of the loop will never execute. For this to happen, the condition would have to evaluate to `false` the first time it was executed.

The body of the loop defines a new scope, inside which local variables and constants can be defined. These go out of scope as soon as the loop terminates, and their values are lost.

The most familiar version of this loop would be as follows:

```
for var i=10; i<15; i++ {  
    print (i)  
}
```

Note the following:

- Semicolons must separate the three expressions that define the setup, test, and increment phases.
- Unlike C, parentheses are optional around the setup, test, and increment code.

for-in loops

You use the `for-in` loop to iterate over sequences or collections of things, such as the elements of an array or dictionary, the characters in a string, or a range of numbers.

Here's the general format:

```
for index in sequence {  
    statements  
}
```

In the following example, which iterates over a range of numbers, the loop index variable (`i`) takes on the value of the next number in the range each time through the loop:

```
for i in 3..8 {  
    print (i)  
}
```

The example that follows iterates over the contents of an array. The loop index variable (`i`) takes on the value of the next entry in the array each time through the loop (see also the section [“Iterating Over Arrays” on page 51](#)):

```
var microprocessors = ["Z80", "6502", "i386"]  
for i in microprocessors {
```

```

    print (i)
}
// prints:
// Z80
// 6502
// i386

```

This next example iterates over the contents of a dictionary. A tuple is used as the loop index, so that for each iteration, you get the next key and its associated value (see also the section “Iterating Over Dictionaries” on page 60). This example also demonstrates that dictionaries are stored in arbitrary order:

```

var vehicles = ["bike":2, "trike":3, "car":4, "lorry":18]
for (vehicle, wheels) in vehicles {
    print (vehicle)
}
// prints:
// car
// lorry
// trike
// bike

```

If you want to implement a loop that runs for a specific number of iterations, and the code inside the loop does not need to know the current iteration number, you can replace the loop index variable with `_` (underscore). For example:

```

for _ in 3..8 {
    print ("**")
}

```

for-in variations. The index value of a `for-in` loop can be *filtered* with a `where` clause to exclude specific iterations. The following example skips iterations where the index is evenly divisible by 3:

```

var out = ""
for i in 0..15 where (i % 3) != 0 {
    if (out != "") { out += ", " }
    out += String(i)
}
print (out)
// prints "1, 2, 4, 5, 7, 8, 10, 11, 13, 14"

```


Instead of testing a condition, a for-in loop can take a case pattern match, which can include enumeration values and let variable binding (and an optional where clause). This example iterates through an array of tuples where the bus-width member has the value 8:

```
let processors: [(name: String, buswidth: Int)] = [
  ("Z80", 8),
  ("16032", 16),
  ("80286", 16),
  ("6502", 8)
]
for case let (name, 8) in processors {
  print ("the", name, "has a bus width of 8 bits")
}
// outputs
// the Z80 has a bus width of 8 bits
// the 6502 has a bus width of 8 bits
```

This example iterates through an enumeration for .IPv4 cases, and extracts the associated values for printing:

```
enum NetworkAddress {
  case MAC(String)
  case IPv4(UInt8, UInt8, UInt8, UInt8)
}
let addresses = [
  NetworkAddress.IPv4(192, 168, 0, 1),
  NetworkAddress.IPv4(8, 8, 8, 8),
  NetworkAddress.MAC("00:DE:AD:BE:EF:00")
]
for case let .IPv4(a, b, c, d) in addresses {
  print (a, b, c, d, separator:".")
}
// outputs:
// 192.168.0.1
// 8.8.8.8
```

See also “switch” on page 96 and “Enumerations” on page 144 for additional information.

while loops

while loops test a condition ahead of the loop body; only if the condition evaluates to true is the loop body executed. The general format is as follows:

```
while condition {
    statements
}
```

You can use the `while` loop to replicate the functionality of the `for-condition-increment` loop, as follows:

```
var count = 0
while (count < 10) {
    print (count)
    count ++
}
```

The condition is tested before the body of the loop is executed. If it evaluates to `false` the first time it is executed, the statements in the body of the loop will never execute.

Instead of testing a condition, `while` can take a case pattern match, which can include enumeration values, `let` variable binding, and a `where` clause. See “[for-in variations](#)” on page 90 for related information and examples.

repeat-while loops

`repeat-while` loops test the termination condition at the end of the loop, rather than at the start. This means the statements in the body of the loop are guaranteed to be executed at least once. Loop execution continues until the condition evaluates to `false`.

The general format for a `repeat-while` loop looks like this:

```
repeat {
    statements
} while condition
```

Here is an example:

```
var t = 0
repeat {
    print (t)
    t++
} while (t < 10)
```

NOTE

The `repeat-while` loop in Swift 2.0 replaces the `do-while` loop from Swift 1.0.

Early termination of loops

You can use a `continue` statement anywhere in the body of the loop to stop the current iteration and begin the next iteration.

To terminate the loop, you use a `break` statement anywhere in the body of the loop, which continues execution at the next statement after the loop.

Conditional Execution

There are three statements in Swift that support conditional execution of blocks of code: the `if-else` statement, the `guard-else` statement, and the `switch` statement.

if-else

The `if` statement tests a condition, and executes a block of code only if that condition evaluates to `true`.

Here's the simple form:

```
if condition {  
    // statements to execute  
}
```

Note that unlike many other languages, in Swift the parentheses are optional around the condition and *the braces are required*, even if only a single statement is to be executed when the condition evaluates to `true`.

The `if` statement has an optional `else` clause. If the condition evaluates to `false`, the statements in the `else` clause are executed:

```
if condition {  
    // statements to execute when condition met
```

```
    } else {  
        // statements to execute when condition not met  
    }  
}
```

In all but one situation, braces are required around the statements in the `else` clause. That situation is when the `else` clause is immediately followed by another `if`, as demonstrated here:

```
if condition {  
    print ("shouldn't see this")  
} else if condition {  
    print ("should see this")  
}
```

You can chain multiple `if` statements in this way, optionally ending with a final `else` clause.

if-case. Swift 2.0 adds the option of a case pattern match (as used in a `switch` statement) as an alternative to a condition test in an `if` statement, as in the following example:

```
var age = 23  
if case 16...35 = age {  
    print ("You're our target demographic!")  
}
```

The case pattern is not limited to simple ranges like that just shown, but can include enumeration values, let variable binding, and a `where` clause. See “[for-in variations](#)” on page 90 for related information and examples.

guard-else

The `guard` statement provides similar functionality to the `if` statement, but is intended for use in situations where you want to do an early bailout of the current scope if one or more conditions are not met.

The basic structure of the `guard` statement in a function is as follows:

```
func someFunc()  
{  
    guard condition else {  
        return // exit function  
    }  
}
```

```

    }
    // continue execution
    // ...
}

```

In this example, `return` was used to exit the function. Other statements that end the current scope can be used in other contexts, such as `break` or `continue`.

Optional binding (see “Optionals” on page 81) can be used as part of the condition, in which case the bound values are available for the remainder of the guard statement’s scope.

The following example shows a loop that processes an array of strings to see which can be interpreted as integers:

```

var input = ["45", "27", "Apple", "3"]

for str in input {
  guard let ageAsInt = Int(str) else {
    // not an int, so ignore
    continue
  }
  print ("age:", ageAsInt)
}

```

- Generally in functions, use `guard-let` rather than `if-let` to ensure an optional holds a value.
- The guard statement can include an optional where *condition* clause. Both conditions must be true in order for the guard statement to not execute the `else` clause.
- If using optional binding (as in `guard-let`), you can use compound conditions—multiple comma-separated binding `let` assignments that must all resolve to non-`nil` values in order for the guard statement to not execute the `else` clause.

As with `if-case`, `guard` can take a case pattern match instead of a condition, and can include enumeration values, `let` variable binding, and a `where` clause. See “for-in variations” on page 90 for related information and examples.

switch

The `switch` statement provides an alternative (and more concise) way to express a series of condition (or *pattern*) tests, which you might otherwise implement by using a chain of `if-else` statements.

The basic structure of the statement is as follows:

```
switch expression {
  case pattern1:
    // statements to execute
  case pattern2:
    // statements to execute
  case patternN:
    // statements to execute
  default:
    // statements to execute
}
```

The *expression* is evaluated, and the result is compared to each of the patterns associated with each case clause. If a match is found, the statements that form part of the matching case are executed. If no match is found, the statements that follow the optional `default` clause are executed.

A pattern may contain a single value or a series of values separated by commas, as shown here:

```
case 2, 4, 6:
```

The `switch` statement in Swift is considerably enhanced compared to its counterpart in C-like languages. Here are the notable differences:

- The case clauses must be exhaustive (all possible values of expression must match a cast pattern, or there must be a default case to catch those that aren't); otherwise, the compiler will report an error.
- Execution of statements attached to a case clause will not fall through into another case unless this behavior is explicitly enabled with the `fallthrough` keyword (this

prevents a common error in C, where a break statement may have been accidentally omitted).

- Every case must contain at least one executable statement.
- If more than one case pattern matches, the first matching case is the one that is used.
- A single case can test for a match against a range of values.
- You can use tuples to test multiple values in a single case pattern.
- The case clause can use an optional where clause to further refine the case match (see the section “[The where qualifier](#)” on page 100).
- The break statement is not required to prevent fall-through into the next case, but you can use it as a “no-operation” statement to terminate a case and continue execution at the next statement after the switch statement. This is useful when you need to match a specific case and exclude it ahead of another more general case that would otherwise include it.

Here is a simple example of a switch statement with multiple cases:

```
var a = "c"
switch a {
  case "a", "e", "i", "o", "u":
    print("this letter is a vowel")
  case "b", "d", "g", "k", "p", "t":
    print("this letter may be a plosive sound in "
          + "English")
  fallthrough
  case "c", "f", "h", "j", "l", "m", "n", "q", "r", "s",
       "v", "w", "x", "y", "z":
    print("this letter is a consonant")
  default:
    print("this character doesn't interest me")
}
```

Let's analyze this example a little closer:

- If a pattern match is made in the first case clause, a message is printed indicating the letter is a vowel, and execution continues at the next statement after the switch statement.
- If a pattern match is found in the second case clause, the `print()` function is called, but the `fallthrough` keyword causes execution to continue into the statement(s) defined as part of the next case clause (in this example, a second `print()` function is called).

Matching ranges in a case clause. A case pattern can be a range. If the switch expression is contained within the range, that case is executed, as in this example:

```
var marbles = 600
switch marbles {
  case 0:
    print("You've lost your marbles!")
  case 1:
    print("I see you have a marble")
  case 2...5:
    print("I see you have some marbles")
  case 6...10:
    print("That's quite a handful of marbles!")
  case 10...99:
    print("That's lots and lots of marbles")
  default:
    print("Were marbles on sale?")
}
```

Using tuples in a case clause. A case pattern can be a tuple. The following example demonstrates a crude class scheduling case statement, in which students in different grades (7–10) and from different houses (“Columbus,” “Cook”) are scheduled for specific activities on different days of the week:

```
let year = 9 // 7-10
let house: String = "Columbus" // "Columbus" or "Cook"
let weekday = "Fri" // "Mon" through "Fri"
```



```

let record = (house, year, weekday)

switch record {
  case ("Columbus", 7...8, "Mon"):
    print ("Sports: Cricket")
  case ("Cook", 7...8, "Mon"):
    print ("Sports: Netball")
  case ("Columbus", 9...10, "Tue"):
    print ("Sports: Football")
  case ("Cook", 9...10, "Tue"):
    print ("Sports: Tennis")
  case (_, 7...8, "Wed"):
    print ("Music")
  case (_, 9...10, "Wed"):
    print ("Theater")
  case (_, 7...10, "Thu"):
    print ("Sciences")
  case (_, 7...10, "Fri"):
    print ("Humanities")
  default:
    print("nothing scheduled or invalid input")
}
// outputs "Humanities"

```

In this example, the underscore (`_`) is used in some case patterns to match all possible values. This example also demonstrates matching a range of values (e.g., all students in grades 7–10, regardless of house, study Humanities on Fridays).

Value binding with tuples and ranges. Because a tuple in a switch-case pattern matches a range of inputs, you can use `let` or `var` *value binding* in a case clause to assign a temporary name to, or make a temporary copy of, part of a matched value, as shown here:

```

switch record {
  // ... preceding cases
  case (_, let yr, "Thu"):
    print ("Sciences - customized for year \{(yr}")
  // subsequent cases...
}

```

In this example, the second component of the tuple still matches any input value. You use the `let` keyword to make a

temporary local constant named `yr`; thus, you can subsequently use whatever value that might be in the scope of the case clause. Alternatively, use the `var` keyword to create a mutable copy of the value and then modify that copy in the scope of the case clause.

The where qualifier. You can use a where qualifier to further refine a case clause in a switch statement. The following example uses the where clause with a value bound to the day of the week to match cases in which students are of either house, in year 7, and the day is any day that begins with the letter “T”:

```
switch record {
    // ... preceding cases
    case (_, 7, let day) where day.hasPrefix("T"):
        print ("Home Economics")
    // subsequent cases...
}
```

Using switch with enumerations. You can use an *enumeration* as the value of a switch statement that is to be matched against each case clause, as illustrated here:

```
enum TravelClass {
    case First, Business, Economy
}

var thisTicket = TravelClass.First

switch thisTicket {
    case .First:
        print ("Cost is $800")
    case .Business:
        print ("Cost is $550")
    case .Economy:
        print ("Cost is $200")
}
// outputs "Cost is $800"
```

Because switch statements must be exhaustive, all possible enumeration values must be checked, or there must be a default clause. See the section “Enumerations” on page 144 for more information.

Statement labels

You can precede switch statements and loop constructs by an optional label, which you can then use as an argument to a break or continue statement to indicate to which switch or loop that break or continue should apply.

Statement labels precede the switch or loop construct as follows:

```
label: repeat {
    // some loop content
} while (someCondition)
```

Or alternatively:

```
label: switch expression {
    // cases
}
```

Here's a simple example of nested loops with a continue statement that causes early termination under some conditions:

```
outerloop: for var i=1; i<10; i++ {
    for var j=1; j<10; j++ {
        if ((i == 6) && ((i * j) >= 30))
            { continue outerloop }
        print (i * j)
    }
    print ("-")
}
```

Without the use of the statement label, the continue statement would skip the inner print function in some circumstances, but it would always execute nine inner loops for each outer loop. The presence of the label changes this, because it causes early termination of the inner loop in some circumstances and affects the overall number of iterations of both loops.

Do scopes

The do statement creates a new scope that can contain its own local variables, functions, and other definitions that will not be visible outside the scope. The syntax is as follows:

```
do {  
    // code  
}
```

On its own, this is not something that has much practical value. The `do` construct is wrapped around calls to functions that can throw an error, and is followed by one or more `catch` clauses to deal with those errors. See the section “[Error Handling](#)” on [page 103](#) for more information.

Deferred execution

Swift 2.0 introduces the `defer` statement, which allows you to specify a block of code that will be executed as the current scope exits.

The following example defers execution of a `print` statement until the end of the `do` scope:

```
do {  
    defer {  
        print ("Goodbye")  
    }  
    print ("Hello")  
}  
// prints:  
// Hello  
// Goodbye
```

`defer` is useful for placing clean-up code near set-up code, clarifying intent. For example, a function that opens a file can ensure the method for closing that file is always executed, regardless of how the function returns, by wrapping it in a `defer` statement at the same place in the code that the file is opened:

```
func parseFile()  
{  
    let handle = openSomeFile()  
    defer { closeSomeFile() }  
  
    // code to parse file  
  
    return  
}
```

Some other points to note about deferred execution are:

- Deferred code cannot include `break`, `continue`, `return`, or other statements that would transfer control elsewhere, and cannot throw errors.
- You can use multiple `defer` statements in a scope—they will be executed in the reverse of the order in which they are declared.

Error Handling

Swift 2.0 introduces error-handling support in the form of a `do-try-catch` sequence, used in conjunction with a `throw` statement and the `throws` annotation on function and method declarations.

NOTE

In earlier versions of Swift, it was common practice to return an optional with the value `nil` from a function or method to indicate an error condition had occurred. With Swift 2.0, you are encouraged to use the error-handling model to indicate errors, and use optionals only to indicate the presence or absence of a value (see “Optionals” on page 81 for more information).

A function (or method) that can detect errors it needs to pass out to its caller is declared with the `throws` annotation. If the error is detected, the function is terminated with the `throw` keyword, as follows:

```
func someFunc(parameters) throws -> returnType
{
    // ...
    if errorDetected { throw error }
    // ...
}
```

The `throw` statement must “throw” an instance that indicates what the error is. Specifically, it must be something that conforms to the `ErrorType` protocol, which can include enums, structs, and classes.

A function (or method) that can throw is normally called with a `do-try-catch` construct:

```
do {
    try someFunc(params)
}
catch [error] {
    // deal with error
}
```

You can provide multiple `catch` clauses, each matching specific errors, much like `case` clauses in a `switch` statement. The `catch` clauses must be exhaustive—i.e., they must match all possible errors that could be thrown by the called function or method.

The example that follows defines an enum, `InputError`, and conforms to the `ErrorType` protocol, with a single case for reporting input that is “not an integer.” The function takes an array of strings that represent integer values, sums them, and returns the result as an integer. If any of the array entries are not convertible to integers, the function throws the error:

```
enum InputError: ErrorType {
    case notAnInt
}

func sumArrayOfStrInts(strs: [String]) throws -> Int
{
    var tot = 0
    for str in strs {
        guard let strAsInt = Int(str) else
            { throw InputError.notAnInt }
        tot += strAsInt
    }
    return tot
}
```

This function might be called as follows:

```
nums = ["12", "24", "8", "17"]
```

```
do {
    let result = try sumArrayOfStrInts(nums)
    print ("Sum of ints is", result)
}
catch InputError.notAnInt {
    print ("Input error: string not an integer literal")
}
catch {
    print ("Some other error occurred")
}
```

Some points to note in relation to error handling include:

- An error will continue to propagate out of the current scope until it is handled by a catch clause that matches it.
- A catch keyword with no pattern acts as a default case, and will match any otherwise unmatched error and bind it to a local constant named `error`.
- If the containing function's catch clauses do not match the error that is thrown, that function must also be annotated with `throws` so the error can be propagated outward.
- The `throws` annotation is part of the function's (or method's) type.

Rethrowing functions and methods

A function or method can be annotated with `rethrows` (instead of `throws`) if it throws an error only as a result of one of its function parameters throwing an error.

- In subclasses, a throwing method cannot override a rethrowing method, but a rethrowing method can override a throwing method.
- In protocols, a throwing method cannot satisfy a rethrowing method requirement, but a rethrowing method can satisfy a throwing method requirement.

Forced Try

If you're sure an error won't actually be thrown when calling a function or method that throws, or you specifically want to abort with a runtime error if an error is thrown, use `try!` instead of `try`.

This *forced try* does not propagate errors, and does not need to be used in a `do-catch` sequence. In the following example, the function will return normally, or the program will crash:

```
try! sumArrayOfStrInts(["12", "24", "8", "17"])
// returns 61
try! sumArrayOfStrInts(["3", "0.4"])
// will produce a runtime error
```

Optional Try

You can disable error propagation from a throwing function or method by calling it using the `try?` keyword.

If the function returns normally (without throwing an error), the result it returns is wrapped in an optional. If the function throws an error, that error is discarded, and the returned value is `nil`.

Note that if the function (or method) originally returned an optional, the result returned by `try?` is an optional optional (it needs to be unwrapped twice to obtain a non-`nil` value).

```
try? sumArrayOfStrInts(["12", "24", "8", "17"])
// returns an Int? of value 61
try? sumArrayOfStrInts(["3", "0.4"])
// returns an Int? of value nil
```

Classes

A class is a flexible entity that encapsulates properties (or data) and the methods that operate on them. You can derive classes from other classes (a feature called *inheritance*), in which case the derived class is referred to as the *subclass*, and the class from which it is derived is referred to as the *superclass*.

NOTE

Much of what is described here also applies, to a greater or lesser degree, to structures and enumerations, which are closely related to classes in Swift. See the sections “Structures” on page 139 and “Enumerations” on page 144 for more details on how they differ.

Unlike Objective-C, in which all classes are subclasses of `NSObject`, Swift allows for classes that are not derived from other classes. Such a class is referred to as *base class*.

Another significant implementation difference between Objective-C and Swift is that Swift does not use a separate header file. An Objective-C class is typically implemented using two separate files—a *.h* file containing the class declaration, and a *.m* file containing the class implementation. With Swift, there is no separation of declaration and definition—the definition serves as the declaration in a single *.swift* file.

NOTE

Classes are *reference types*—when an instance of a class is passed to a function or stored in a variable of the same type, the instance is not copied. Instead, a new reference to the original instance is created.

This is in contrast to Swift’s primitive types, such as `Int`, `Double`, and `String`, as well as some of Swift’s more complex types, such as `Array`, `Dictionary`, structures, and enumerations, all of which are *value types*. When a value type is passed to a function or stored in another variable of the same type, a complete copy is made. Swift handles this efficiently, only copying array elements that need to be mutated and only if they are actually accessed.

Defining a Base Class

You declare a base class by using the following syntax:

```
class ClassName
{
    // property, member and related definitions
}
```

Here is an example of a simple base class that could be used to store a description of a microprocessor:

```
class Processor
{
    var dataWidth = 0
    var addressWidth = 0
    var registers = 0
    var name = ""
}
```

When it's defined, a class is like a new type: you can create variables or constants whose type is the class name, or use instances of the class in dictionaries and arrays as you would for built-in types.

The four variables defined in the class just shown are called *properties*. In other languages, they are variously called *instance variables*, *ivars*, or *data members*.

You must initialize properties that can store values. In the preceding example, they are initialized with an assignment (`dataWidth = 0`). Alternatively, you can initialize them by using a separate initialization method, which is described in the section [“Initialization” on page 130](#).

Instances

You can think of a class as a recipe for constructing something, but it isn't the something that is actually constructed. The entity made from the recipe is called an *instance* or an *object*. In the same way, `Int` is a type of data, and a variable of type `Int` is an instance of that type of data.

You create instances of simple classes, such as the `Processor` class shown in the example in the previous section, by using the class name followed by empty parentheses, as shown here:

```
let proc = Processor()
```

This process, referred to as *instantiation*, constructs a new instance of the `Processor` class and creates a constant called `proc` that refers to it.

After you've created an instance, you can access its properties and modify them by using dot syntax:

```
proc.name = "Z80"  
proc.dataWidth = 8  
print (proc.name)
```

NOTE

You might wonder why `proc` can be declared as an immutable constant (with `let`), and yet you can modify its properties. Although `proc` itself is immutable (and cannot later be used to refer to a different instance of the class), it is a *reference* to an instance of the class that *is* mutable. There is currently no way in Swift to create an immutable instance of a class that has mutable properties.

At this level, classes are not substantially different to structs in C. The usefulness of classes comes through the *methods* you can add to them, which can use and manipulate the properties of the class.

Because classes are reference types, new copies are not made during assignment. Consider this code:

```
var newProc = proc
```

Upon execution, `newProc` is a reference to the same object as `proc`. You can verify this by modifying a property of `proc` and checking that same property of `newProc`:

```
proc.name = "6502"  
print (newProc.name)  
// will output "6502"
```

You can test that `proc` and `newProc` refer to the same instance by using Swift's *identity* operators (`===` and `!==`). These operators test whether two references point to the same object:

```
if (proc === newProc) {  
    print ("references are to same instance")  
}  
// will output "references are to same instance"
```

Properties

Properties are values associated with a class or an instance of a class and the methods for accessing them. When they are associated with each instance of a class, they are known as *instance properties*, and when they are associated with the class itself, they are known as *type properties*. Properties can be stored values or they can be computed from other properties or values at runtime.

Stored properties

Stored properties are those for which space in memory is allocated to store the property's value. This is in contrast to *computed properties* (described a little later) for which no such space is allocated.

You declare stored properties with `var` (if they are to be mutable) or `let` (if they are to be immutable) inside the class definition. In the `Processor()` example from earlier, `dataWidth`, `addressWidth`, `registers`, and `name` are all examples of stored properties.

Stored properties must be initialized either by direct assignment in the class definition, or by using an initialization method, as described in the section “**Initialization**” on page 130.

NOTE

You can define stored properties for both classes and structures.

Lazy initialization of stored properties. *Lazy evaluation* is the practice of not computing a value until it is actually needed, which is sometimes useful when setting the initial value of a stored property.

Consider the following cases:

- A property may not actually be required during the life cycle of an instance. Not initializing it unless it’s needed saves time, especially if it’s computationally expensive to generate the initial value.
- A property’s initial value might depend on the result of a method call or the value of another property in the same class. The problem here is that none of the other methods or properties are available until after the class is fully initialized, and it can’t be fully initialized until all properties are given initial values. (Notably, `self` isn’t defined until the instance is fully initialized.)

One solution is to use a computed property (described next), which is in reality a method call, and consequently isn’t evaluated until it is called.

Another approach is to mark the property as `lazy`, which means its initial value won’t be computed until the first time its value is used, as in the following contrived example:

```

class LazyClass
{
    var aString = "elephant"
    lazy var bString: String = self.aString + " trunk"
}

```

Without the lazy prefix, the compiler will declare that “self is an unresolved identifier” and the code will not compile. With the lazy prefix, the class can be instantiated. The following code shows that bString really is only initialized when its value is referenced, as it uses the modified version of aString:

```

var lazyInst = LazyClass()
lazyInst.aString // returns "elephant"
lazyInst.aString = "tree"
lazyInst.bString // returns "tree trunk"

```

Here are some other points to note in relation to lazy stored properties:

- Lazy stored properties must be declared with var, not let.
- If any of the code that initializes a lazy property references other members of the class (or structure), those references must explicitly include the self. prefix.
- The lazy property’s type must be explicitly declared.

Computed properties

Like computed variables, computed properties do not store a value but are methods that look like properties. They are defined in terms of a *getter* (identified with the keyword get, which returns the computed property) and a *setter* (identified with the keyword set, which might change the conditions that affect the value returned by the getter). You can also use the getter and setter to read and write other properties, or call other methods of the class. You define a computed property as follows:

```

class someClass
{

```

```

var propertyName: someType {
    get {
        // code that computes and returns
        // a value of someType
    }
    set(valueName) {
        // code that sets up conditions using valueName
    }
}

```

valueName is optional. It is used to refer to the value passed into the set method. If you omit it, you can refer to the parameter by using the default name of *newValue*.

The setter is optional. If the setter is not used, the get clause is not required, and all that is needed is code to compute and return a value:

```

class someClass
{
    var propertyName: someType {
        // compute and return a value of someType
    }
}

```

After a computed property is defined, it is used exactly like any other property. If its name is used in an expression, the getter is called. If it is assigned a value, the setter is called.

Here is an example of a simple class named *Rect* that represents a rectangle in terms of a corner point, a width and a height, and defines a computed property called *area* to return the area of the *Rect*:

```

class Rect
{
    var x = 0.0, y = 0.0
    var width = 0.0, height = 0.0
    var area: Double { return (width * height) }
}

```

You could use this as follows:

```

var q = Rect()
q.width = 2.7
q.height = 1.4

```

```
q.area
// returns 3.78
```

NOTE

You can define computed properties for classes, structures, and enumerations.

Property observers

Property observers are functions you can attach to stored properties and that are called when the value of the property is about to change (identified with the `willSet` keyword) or after it has changed (identified with the `didSet` keyword). The declaration looks as follows:

```
class Observer
{
    var name: String = "" {
        willSet(valueName) {
            // code called before the value is changed
        }
        didSet(valueName) {
            // code called after the value is changed
        }
    }
}
```

Both *valueName* identifiers (and their enclosing parentheses) are optional.

The `willSet` function is called immediately before the property is about to be changed (except for assignment during initialization). The new value is visible inside `willSet` as either *value Name*, or `newValue` if *valueName* was not specified. The function can't prevent the assignment from happening and can't change the value that will be stored in the property.

The `didSet` function is called immediately after the property has been changed (except for assignment during initialization).

The old value is visible inside `didSet` as either *valueName* or `oldValue` if *valueName* was not specified.

Here are some other points about property observers:

- When creating a subclass, you can override properties of the superclass and then add property observers to them, allowing you to create new behaviors the designer of the superclass did not plan for or consider.
- Property observers in a superclass are called when a property is set in the initializer of a subclass.
- You can't define a property observer for a lazy stored property.

NOTE

Observers can be defined for regular variables in the same way they are defined for properties. See the section “[Variable Observers](#)” on page 25 for more details.

Instance versus type properties

For most applications of classes, properties are associated with each instance of the class. Using the `Processor()` class example from earlier, each microprocessor has a different name, different numbers of registers, and potentially different widths for their data and address paths. Each instance of the class requires its own set of these property values. Properties used in this way are called *instance* properties—there are unique copies associated with each instance of the class.

Some applications only require a single instance of a given property for the entire class. Consider a class that records employee data and must keep a record of the next available ID for a new employee. This ID should not be stored in each

instance, but it does need to be associated with the class in some way.

For such purposes, Swift provides *type properties*, which are properties associated with the class, not with a specific instance of the class. The same feature is referred to generically as a *class variable*, or in C++, Java, and C#, as a *static member variable*.

NOTE

Swift supports *computed* type properties in classes, but it does not currently support *stored* type properties in classes. Both are available in structures and enumerations. If you try to create a stored type property in a class, the compiler (as of the version of Xcode 7 available when this edition was updated) will give the error “class stored properties not yet supported in classes; did you mean static?” This would seem to imply that support *may* be coming in a future release.

To further complicate things, *static* stored properties *are* supported in classes and provide similar functionality. They are discussed next.

Static properties. In most cases where you would want to use the (unimplemented) stored type property in a class, you can use a static property instead. To create a static property, precede the property’s definition with the keyword `static`, as in the following example:

```
class Employee
{
    static var nextID = 1

    var familyName = ""
    var givenName = ""
    var employeeID = 0
}
```

In this class, only a single instance of the property `nextID` exists, regardless of how many employee instances are created. To access a static property use dot-syntax with the class name, as in this example:

```
var emp = Employee()
emp.employeeID = Employee.nextID++
```

Note the following points when using static properties:

- Static properties must be assigned a default value as part of the class definition (the initialization routines, described later, only get called when *instances* are created).
- Static properties are inherited and shared by their subclasses. For example, if a new class is defined that is a subclass of the `Employee` class, it will inherit and share the same single instance of the `nextID` property.

Computed type properties. To declare a computed type property for a class, precede the type property's definition with the keyword `class`. The syntax for creating a read/write computed type property is as follows:

```
class SomeClass
{
    class var someComputedProperty: SomeType
    {
        get { return SomeType }
        set(valueName) {
            // do something with valueName
            // that sets the property
        }
    }
}
```

The `get` method performs some operation to produce a value of the declared type, and then returns it.

The `set` method performs some operation to initialize things associated with the class—for example, it could modify a static property, and/or call system functions that have some interac-

tion with the class. *valueName* is optional. It is used to refer to the value passed into the function. If you omit it, you can refer to the value with the default name *newValue*.

If you want a read-only computed type property, omit the set definition. In this case, you can also omit the get keyword and reduce the var body to just the code that calculates and returns the desired value.

The next example reimplements the `Employee` class from earlier to show a computed type property in action. The `_nextID` static variable is still present, but it is now protected from external access (see “[Access Control](#)” on page 151) and a computed type property is implemented to provide gated access to that static value.

```
class BetterEmployee
{
    private static var _nextID = 1
    class var nextID: Int
    {
        get { return _nextID }
        set { _nextID = newValue }
    }
    var familyName = ""
    var givenName = ""
    var employeeID = 0
}
```

To access the computed type property, use dot-syntax with the class name, as before:

```
var be = BetterEmployee()
be.employeeID = BetterEmployee.nextID++
```

Even though this code appears to be the same as the earlier version, access to the static `_nextID` property is now only available indirectly through the `nextID` computed type property’s get and set methods, which are called automatically when the property is read and written.

Note the following when using computed type properties:

- A computed type property cannot access instance properties or instance methods because it is not called in relation to an instance, but it can call type methods (see “[Type methods](#)” on page 122) and other computed type properties, and it can access static properties in the class.
- Computed type properties can be overridden in subclasses. See “[Overriding Superclass Entities](#)” on page 126 for more information.

Constant properties

You can declare properties as constants by using the keyword `let`. Unlike regular constants, constant properties do not need to be assigned a value when they are defined. Instead, the setting of their value can be deferred until the instance is initialized.

The following example demonstrates a simple class that includes an uninitialized constant property, `cp`, the value of which is set by the initializer function (see “[Initialization](#)” on page 130):

```
class ConstPropClass
{
    let cp: Int

    init(v: Int)
    {
        self.cp = v
    }
}

var cpDemo = ConstPropClass(v: 8)
cpDemo.cp
// returns 8
```

When using constant properties, you should note the following:

- Constant properties can only be initialized by the class in which they are defined; they cannot be initialized in a subclass.
- A constant property's value can be modified during initialization, but it must have a value before the initialization process has completed.

Methods

Methods are functions that are either associated with a class (in which case, they are known as *type methods*) or with every instance of a class (in which case, they are known as *instance methods*).

You define methods like functions inside the class definition, as in the following example, which revises the earlier example of the `Rect` class to use a method to return the area of the rectangle:

```
class Rect2
{
    var x = 0.0, y = 0.0
    var width = 0.0, height = 0.0
    func area() -> Double
    {
        return width * height
    }
}
```

The method is called using dot syntax, as in this example:

```
var r2 = Rect2()
r2.width = 5.0
r2.height = 2.0
r2.area()
// returns 10.0
```

Local and external parameter names

In Swift 2.0, the rules for local and external parameter names in methods match those for functions. Each parameter in a method definition has two names—a local name, used inside the method definition to reference the value, and an external name, which must be specified in calls to the method. Thus, every parameter in a method definition can be expressed as *externalName localName: type*.

- For the first parameter, if an external name is not provided, then it defaults to `_` (underscore).
- For any parameter except the first, if an external name is not provided, then it defaults to the local name.
- For any parameter, if the external name is declared, then it must be used in calls to the method.
- If an external name is defined as `_` (underscore), then that parameter name can *not* be used in calls to the method.
- Swift’s named parameter feature does not allow for parameters to be listed in “arbitrary order” when calling a method—they must be listed in the order they are defined.

Self

Every instance of a class (and other types) has an automatically generated property named `self` that refers to that instance. Consider this extended version of the `Rect` class:

```
class Rect3
{
    var x = 0.0, y = 0.0
    var width = 0.0, height = 0.0
    func area() -> Double
    {
        return width * height
    }
    func sameSizeAs(width: Double, _ height:Double) -> Bool
```

```
    {  
        return width == self.width && height == self.height  
    }  
}
```

In the `area()` method, both `width` and `height` are properties of the instance, and the `return` statement could have explicitly referred to them as:

```
return self.width * self.height
```

But this is not necessary, because in method calls, `self` is normally implied whenever properties or methods of that class are used.

The exception is when a parameter name for a method is the same as a property name for the class, such as occurs with the `sameSizeAs()` method. Parameter names take precedence over property names in methods; therefore, `self` must be explicitly used to differentiate the `width` and `height` properties from the `width` and `height` parameters.

Type methods

To define a type method for a class, precede the method's definition with the keyword `class`, as in the following example:

```
class someClass  
{  
    class func someTypeMethod()  
    {  
        // implementation  
    }  
}
```

Despite appearances, this is not a nested class definition.

To call a type method for a class, precede it with the class name using dot syntax:

```
someClass.someTypeMethod()
```

Note the following points when using type methods:

- A type method cannot access instance properties or instance methods because it is not called in relation to an instance, but it can call other type methods and access computed type properties and static properties in the class.
- To access static or computed type properties, or to call one type method from another within the same class, you can omit the class name because it is equivalent to `self`.
- Type methods can be overridden in subclasses. See [“Overriding Superclass Entities” on page 126](#) for more information.

Subscripts

In Swift, you can define subscript methods for your own classes, which makes it possible to use *subscript syntax* to read and write values appropriate for an instance of your class.

Subscript syntax is how you access members of arrays and dictionaries, as shown here:

```
var m = [Int](count:10, repeatedValue:0)
m[1] = 45
m[2] = m[1] * 2

var cpus = ["BBC Model B":"6502",
           "Lisa":"68000",
           "TRS-80":"Z80"]
let cpu = cpus["BBC Model B"]
```

Subscript syntax affords both reading and writing of values, and adheres to the following general pattern:

```
class SomeClass
{
    subscript(index: someIndexType) -> someReturnType
    {
        get {
            // return someReturnType based on index
        }
        set(valueName) {
            // write valueName based on index
        }
    }
}
```

```

    }
}

```

You can omit the *valueName* parameter name, in which case the parameter to be written can be accessed as `newValue`.

Here's an example class that can represent byte values. It also defines a subscript method by which you can access individual bits as though the byte is an array of bits:

```

class Byte
{
    var d: UInt8 = 0

    subscript(whichBit: UInt8) -> UInt8
    {
        get { return (d >> whichBit) & 1 }
        set {
            let mask = 0xFF ^ (1 << whichBit)
            let bit = newValue << whichBit
            d = d & mask | bit
        }
    }
}

```

After it is defined, you can use the class like this:

```

var b = Byte()
b[0] = 1
// b is now 0000 0001, or 1
b[2] = 1
// b is now 0000 0101, or 5
b[0] = 0
// b is now 0000 0100, or 4

```

Here are some additional things you can do in relation to subscripts:

- For a read-only subscript, omit the set definition—in this case, you can also omit the `get` keyword and reduce the subscript body to just the code that calculates and returns the desired value.
- Subscript parameters aren't limited to single integer values; you can declare a subscript method that takes any

number of floats, strings, or other types that suit your requirements.

- You can define multiple overloaded subscript methods as long as they take different numbers and/or types of parameters, or return different types of value. Swift will determine the appropriate method to call using type inferencing.

Member Protection

Swift provides a mechanism for controlling access to properties, methods, and subscripts of classes as part of a broader access control system. Read the section “[Access Control](#)” on [page 151](#) for more information.

Inheritance: Deriving One Class from Another

You can define new classes in terms of existing classes. In doing so, the new class is said to *inherit* all of the properties and methods of the existing class; the new class is *derived* from the existing class.

A common example used to illustrate inheritance is 2D geometric shapes. The generic *base class* contains methods and properties that should be common to all shapes, such as color, fill, line thickness, and perhaps origin or enclosing rectangle. Derived classes include actual geometric shapes, such as lines, circles, ellipses, quads, polygons, and so on. Each of these introduce new methods and properties specific to that shape (such as a draw method and properties to store the geometric details), but all inherit from the base class the common set of properties and methods that all shapes have.

In Swift, you derive one class from another by using this syntax:

```
class NewClassName: BaseClassName
{
```

```
    // property and method definitions for the new class
}
```

Overriding Superclass Entities

When one class is derived from another, the new class is called the *subclass*, and the class from which it is derived is called the *superclass*. Although much of the time a subclass will add its own properties and methods to those inherited from the superclass, a subclass also has the ability to *override* methods and properties of the superclass by redefining them itself.

To override something already defined in a superclass, you must precede its definition in the subclass with the `override` keyword. This is a signal to the Swift compiler that the redefinition is intentional, and that you haven't accidentally created a method or property with the same name.

In Swift, you can override methods, properties, and subscripts.

Accessing overridden superclass entities

At times it is necessary for a method in a subclass to access a method or property in its superclass, rather than an overridden version in the subclass. This is accomplished with the `super` prefix, as follows:

- To access an overridden method, call it with `super.methodName()`.
- To access an overridden property, refer to it as `super.propertyName` in the getter, setter, or observer definitions.
- To access an overridden subscript, use `super[indexValue]`.

Overriding properties

You can't actually override a property in a superclass with your own property (it wouldn't make sense to duplicate the storage

space), but you can override a property in order to provide your own custom getter and setter for the superclass instance, or add a property observer so you can be informed when the property value changes.

Earlier in this section, the `Rect` class was demonstrated as an example for storing arbitrary rectangles. The example that follows creates a derived class, `Square()`, that overrides the `width` and `height` properties with new getters and setters that ensure the `width` and `height` always match, and therefore that instances of the `Square` class are always in fact square:

```
class Square: Rect
{
    override var width: Double {
        get { return super.width }
        set {
            super.width = newValue
            super.height = newValue
        }
    }
    override var height: Double {
        get { return super.height }
        set {
            super.width = newValue
            super.height = newValue
        }
    }
}
```

Note that the getter and setter still access the properties that are stored in the superclass via the `super` prefix. Here is an example of the class in use:

```
var s = Square()
s.width = 20.0
s.height = 10.0
s.area
// returns 100.0
s.width
// returns 10.0 (not 20.0)
```

When overriding properties, note the following:

- You can override inherited read-only properties as read/write properties, by defining both a getter and setter.
- You *cannot* override inherited read/write properties as read-only properties.
- If you provide a setter, you must also provide a getter (even if it only returns the superclass property unmodified).
- You can override inherited mutable properties (declared with `var`) with property observers, but you cannot override inherited immutable properties (declared with `let`) in this way (because property observers are intended to observe *writes* to the property).
- You cannot override a property with both a setter and an observer (because the setter can act as the observer).

Overriding methods and subscripts

To override a method or a subscript that exists in the superclass, precede the method or subscript name in the derived class with the `override` keyword.

In the earlier section on subscripts, the class `Byte` was used to demonstrate how a subscript method can be used to provide access to each bit of a byte as though it were an array of bits, using subscript syntax.

A serious omission with this class is that it does not perform bounds-checking on either the subscript value or the value to be written. If you refer to a bit position higher than 7, the program will terminate because the mask assignment in the setter will generate an overflow. If you assign a bit value of something other than 0 or 1, the assignment will happen, but it will pollute other bits in the byte property presented as the array of bits.

The following example demonstrates a safer derived class that overrides the subscript definition to ensure these values are valid:

```
class SafeByte: Byte
{
    override subscript(whichBit: UInt8) -> UInt8 {
        get { return super[whichBit & 0x07] }
        set { super[whichBit & 0x07] = newValue & 1 }
    }
}
```

Note that this still uses the superclass implementation of the subscript function; it just sanitizes the bit value and bit position before doing so.

Preventing Overrides and Subclassing

Prepending the keyword `final` to a property, method, or subscript definition prevents that entity from being overridden in a derived class. Here is a modified version of the `Rect` class that uses the `final` keyword to prevent the `width` and `height` properties from being overridden:

```
class Rect4
{
    var x = 0.0, y = 0.0
    final var width = 0.0, height = 0.0
    // rest of definition
}
```

Note that the use of `final` in this context does not mean the values of `width` and `height` are locked; it just means the properties cannot be overridden in a subclass.

This change means the `Square` class from earlier would no longer compile, because it overrides these properties with a custom setter and getter.

You can also use the `final` keyword in front of a class definition to prevent that class from being subclassed.

Initialization

Initialization is the process of setting up appropriate default values for stored properties in a new instance of a class, structure, or enumeration. The process is similar to a constructor in C++, C#, or Java, or the `init` selector in Objective-C. It ensures a new instance is ready for use and does not contain random or uninitialized data.

Initialization happens automatically for a new instance of a class; you do not call the initializer explicitly, although you do need to call initializers in a superclass from the initializer of a derived class (this is discussed more in “[Initialization and Inheritance](#)” on page 137).

You can initialize stored properties by either assignment of default values in the class definition or by defining one or more `init()` functions in the class. The `Byte` class introduced earlier demonstrated initialization by assignment:

```
class Byte
{
    var d: UInt8 = 0
    // rest of class definition
}
```

For more complex classes, it is common to write one or more `init()` functions to manage the process of instantiating a new instance of an object. For classes, Swift supports two kinds of initialization: *designated initializers* and *convenience initializers*.

A designated initializer must initialize all of the properties of a class. In a subclass, it must initialize all of the properties defined in that subclass and then call a designated initializer in the superclass to continue the initialization process for any inherited properties.

A convenience initializer provides a way to call a designated initializer with some of the designated initializer’s parameters set to common default values.

A designated initializer is defined by using the following syntax:

```
class ClassName
{
    init(parameterList)
    {
        // statements to initialize instance
    }
    // rest of class definition
}
```

A convenience initializer is defined by using the following syntax:

```
class ClassName
{
    convenience init(parameterList)
    {
        // statements to initialize instance
    }
    // rest of class definition
}
```

Here are some important characteristics of the initialization process:

- If a property has a property observer, that observer is not called when the property is initialized.
- Properties whose type is *optional* are automatically initialized to `null` if you do not separately initialize them.
- Immutable properties (declared with `let`) can be modified during initialization, even if assigned a default value in the class definition.
- A designated initializer is the main initializer for a class. Most classes will only have one, but more than one is allowed: for example, one with no arguments that sets all properties to default values, and one with arguments that serve as initialization values for specific properties.
- A designated initializer *must* call a designated initializer for its superclass.

- Convenience initializers are optional secondary initializers; they *must* call another initializer in the same class.
- A convenience initializer’s execution *must* eventually lead to the execution of a designated initializer.

Swift also supports *deinitializers*, which are invoked automatically immediately before an object is deallocated (see the section “[Deinitialization](#)” on page 138 for more information).

Designated initializers

The `Rect` class described earlier demonstrated initialization by assignment in the class definition:

```
class Rect
{
    var x = 0.0, y = 0.0
    var width = 0.0, height = 0.0
    // rest of definition
}
```

This can be rewritten to use a designated initializer function instead, like this:

```
class Rect
{
    var x, y, width, height: Double
    init()
    {
        x = 0.0; y = 0.0
        width = 0.0; height = 0.0
    }
    // remainder of class definition
}
```

This default `init()` function, without parameters, is the initializer that is called when you create a new object with no initialization parameters, as in the following:

```
var q = Rect()
```

You can create additional initializers, each of which takes different numbers and/or types of parameters. The following extended version of the `Rect` class includes two different design-

nated initializer methods, either of which will be called depending on how the `Rect` is instantiated:

```
class Rect6
{
    var x, y, width, height: Double
    init()
    {
        x = 0.0; y = 0.0
        width = 0.0; height = 0.0
    }
    init(x: Double, y: Double,
        width: Double, height: Double)
    {
        self.x = x
        self.y = y
        self.width = width
        self.height = height
    }
}
```

Now, either `init()` method can be used to construct `Rect` instances:

```
var rectI1 = Rect6()
var rectI2 = Rect6(x: 2.0, y: 2.0, width: 5.0, height: 5.0)
```

Note that the second `init()` function has only defined local parameter names, but the instantiation of `rectI2` shows they are externally visible.

NOTE

For `init()` functions, Swift will always generate an external parameter name if one hasn't been defined. Moreover, external parameter names (whether explicitly defined or implicitly generated) *must* be used when the class is instantiated.

If you want to prevent the generation of an external parameter name, precede the local parameter name with an underscore, like this:

```

class Rect6
{
    var x, y, width, height: Double
    init()
    {
        x = 0.0; y = 0.0
        width = 0.0; height = 0.0
    }
    init(_ x: Double, _ y: Double,
        _ width: Double, _ height: Double)
    {
        self.x = x
        self.y = y
        self.width = width
        self.height = height
    }
}

```

Because there are now no external parameter names, new instances of the class can be created just by specifying the parameter values, as shown here:

```
var rectI3 = Rect6(2.0, 2.0, 5.0, 5.0)
```

Convenience initializers

Convenience initializers are secondary initialization functions that must call some other initializer within the same class, and ultimately they must cause the execution of a designated initializer.

In “[Defining a Base Class](#)” on page 108, a simple class called `Processor` was introduced to represent microprocessors. The use of this class might require frequent instantiation of a particular type of processor, and hence warrant the inclusion of a convenience initializer:

```

class Processor2
{
    var dataWidth = 0
    var addressWidth = 0
    var registers = 0
    var name = ""

    init (name: String, dWidth: Int, aWidth: Int, regs: Int)
    {

```

```

        self.name = name
        dataWidth = dWidth
        addressWidth = aWidth
        registers = regs
    }

    convenience init (eightbitName: String, regs: Int)
    {
        self.init(name: eightbitName, dWidth:8,
                  aWidth:16, regs: regs)
    }
}

```

Note that the convenience initializer defaults two of the four parameters required by the designated initializer, which it calls as `self.init()`.

The convenience initializer is called automatically when a new instance is created with parameters that match its signature, as in the following example:

```
var p = Processor2(eightbitName:"6502", regs:3)
```

Failable initializers

A failable initializer is one that can return `nil` if a class, structure, or enumeration is unable to properly initialize an instance of itself. For example, a class might require that a resource be available on disk, or a network connection be available, before it can be instantiated.

You indicate that an initializer is failable by defining it as `init?()`.

Following is a version of the `Employee` class that includes a failable initializer. The initialization will fail if the family name concatenated with the given name is an empty string:

```

class Employee3
{
    static var nextID = 1

    var familyName: String
    var givenName: String
    var employeeID = 0
}

```

```

init?(familyName: String, givenName: String)
{
    self.familyName = familyName
    self.givenName = givenName
    if familyName + givenName == "" {
        return nil
    }
    employeeID = Employee3.nextID++
}
}

```

When instances of this class are created *using this specific initializer*, they are optionals. Consider the following code:

```

var emp1 = Employee3(familyName: "Jones", givenName: "Bob")
var emp2 = Employee3(familyName: "", givenName: "")
emp1?.givenName // returns "Bob"
emp2?.givenName // returns nil

```

Both `emp1` and `emp2` are of type `Employee3?`—optional values that must be unwrapped in order to access the employee record. `emp1` contains a valid employee record, whereas `emp2` is `nil`.

Some other points to note include:

- For classes, a failable initializer must not fail (return `nil`) until after all stored properties for the class have been set to an initial value, and a designated initializer has been run.
- A failable initializer can call another failable initializer in the same class or a superclass. If that initializer fails, the entire initialization process fails immediately.
- A failable initializer can be overridden in a subclass with either a failable or a nonfailable initializer.
- A failable initializer cannot override a nonfailable initializer in a superclass.
- You can create an implicitly unwrapped failable initializer by defining it as `init!()`. If you do so, you must check that the created instance is not `nil` before accessing its

methods and properties, otherwise a runtime error will occur.

Initialization and Inheritance

Hierarchies of classes introduce additional complexity into the way initializers are defined and used, including the following:

- A designated initializer must set values for all properties introduced in its own class before calling a superclass initializer.
- A designated initializer must call a superclass initializer before setting the value of any inherited property.
- A convenience initializer must call another initializer in its class (convenience or designated) before setting the value of any property.
- Initializers cannot call instance methods, read instance properties, or refer to `self` until all properties introduced by the class and all properties of its superclass hierarchy have been initialized.

A derived class in Swift does not usually inherit initializers from a superclass, but there are two circumstances in which it does:

- If the derived class does not define any designated initializers of its own, it will automatically inherit all of the designated initializers of its superclass.
- If the derived class implements all of its superclass designated initializers, by any combination of defining them itself and inheriting them, it will automatically inherit all of its superclass convenience initializers.

Overriding initializers

You can override initialization functions in a derived class, but you must consider the following:

- To override a designated initializer, you *must* precede its definition with the keyword `override`.
- To override a convenience initializer, it must use the same number of parameters, with the same names and types, as the superclass initializer it is overriding, but you must *not* use the `override` keyword.

Required initializers

The `required` keyword, when used in front of an initializer, means the initializer must be implemented in a derived class. Here are two issues to keep in mind:

- A required designated initializer *must* be redefined in the derived class.
- A required convenience initializer does not need to be redefined in the derived class if it will be automatically inherited, unless the inherited behavior is not desirable.

Deinitialization

Deinitialization is the process of cleaning up anything related to an instance of a class immediately before that instance is deallocated. The process is similar to a destructor in C++ and C#, or a `finalize` method in Java.

A deinitializer is declared by using the `deinit` keyword, as shown here:

```
class SomeClass
{
    // other parts of class definition
    deinit
    {
```



```
        // code to tidy up before deallocation
    }
}
```

The `deinit` function is called automatically and cannot be called directly.

A derived class inherits its superclass deinitializer, and the superclass deinitializer is automatically called at the end of the derived deinitializer’s implementation.

Structures

In Swift, structures are closely related to classes (see “Classes” on page 106), which can be surprising for C and Objective-C programmers, but less surprising for those familiar with C++, in which classes and structs are also closely related.

Here are some notable similarities and differences:

- Like classes, structures can have properties, instance and type methods, subscripts, and initializers, and they can support extensions and protocols (see the sections “Extensions” on page 155 and “Protocols” on page 162).
- Structures can’t inherit from or be derived from other structures and can’t have deinitializers.
- Structures are *value types*, whereas classes are *reference types*. This means structures are always copied when assigned or used as arguments to functions; they don’t use reference counts.

The syntax for declaring a new structure is as follows:

```
struct StructureName
{
    // property, member and related definitions
}
```

Properties in Structures

Properties and property features in structures are largely identical to those of classes, and you should read the subsection “[Properties](#)” on page 110 to learn the basics.

Like classes, structures support stored and computed properties, property observers, and constant properties.

Structures also support *type properties*. While classes introduce these by using the keyword `class`, in structures they are introduced with the keyword `static`.

Methods in Structures

Structures can have instance methods, defined by using the same syntax as they are with classes. The following example demonstrates a structure for representing rectangles in terms of a corner point, a width, and a height, and includes a method `area()` that computes the area of the shape:

```
struct Rect1
{
    var x = 0.0, y = 0.0, width = 0.0, height = 0.0

    func area() -> Double
    {
        return width * height
    }
}
```

Mutating methods

By default, instance methods defined in a structure are not able to modify that instance’s properties. You can enable this behavior, however, by defining the instance method as a *mutating method*.

The following example modifies the earlier `Rect` structure to include a mutating method `embiggenBy()`, which modifies the `width` and `height` properties:

```
struct Rect2
{
```

```

var x = 0.0, y = 0.0, width = 0.0, height = 0.0

mutating func embiggenBy(size: Double)
{
    width += size
    height += size
}
}

```

A mutating method can also replace the current instance of the structure with a new instance by direct assignment to `self`.

Type Methods for Structures

When declaring a type method for a structure, precede the method's definition with the keyword `static` (in contrast to classes, for which the keyword `class` is used), as illustrated in this example:

```

struct someStructure
{
    static func someTypeMethod()
    {
        // implementation
    }
}

```

To call the type method for a structure, you precede it with the structure name using dot syntax, as shown here:

```
someStructure.someTypeMethod()
```

Note the following additional issues when using type methods:

- A type method cannot access instance properties or instance methods because it is not called in relation to an instance, but it can call other type methods and access type properties in the structure.
- To access a type property or call a type method from another type method within the same structure, you can omit the structure name because it is equivalent to `self`.

Initializers in Structures

As with classes, there are a number of different ways you can initialize a structure before it is used. The most direct way is by assigning default values to stored property members in the structure definition, as in this example:

```
struct Rect3
{
    var x=0.0, y=0.0, width=0.0, height=0.0
}
```

New instances of this structure can be instantiated without specifying any parameters:

```
var r3a = Rect3()
```

If no initializer methods are included in a structure definition, Swift will automatically create a *memberwise initializer* that allows each stored property to be specified during instantiation, such as the following:

```
var q2 = Rect3(x:2.0, y:2.0, width:2.0, height:5.0)
```

If you require more flexibility than what is provided by either the memberwise initializer or default values, you can write your own `init()` method (or methods) to do custom initialization. For example, here is a variation of the `Rect` class that includes an `init()` method for creating squares:

```
struct Rect4
{
    var x=0.0, y=0.0, width=0.0, height=0.0

    init (x: Double, y: Double, side: Double)
    {
        self.x = x
        self.y = y
        self.width = side
        self.height = side
    }
}
```

Initializer delegation in structures

For structures that include more than a few `init()` methods, you can use *initializer delegation*, by which one `init()` method calls another to carry out some part of the initialization process. Here is an example of a structure for implementing a `Rect` class that includes two different `init()` methods:

```
struct Rect5
{
    var x, y, width, height: Double

    init(_ x: Double, _ y: Double,
        _ width: Double, _ height: Double)
    {
        self.x = x
        self.y = y
        self.width = width
        self.height = height
    }

    init()
    {
        self.init(0.0, 0.0, 0.0, 0.0)
    }
}
```

The first `init()` method provides a way to initialize the structure so that each initial property value is specified when the structure is instantiated:

```
var r5a = Rect5(0.0, 0.0, 3.0, 4.0)
```

The second `init()` method provides another way to instantiate the structure, as in the following:

```
var r5b = Rect5()
```

In this case, the second `init()` function delegates all of its work to the first `init()` function by using the `self` prefix, with each parameter set to 0. Only initializers are allowed to call other initializers in this way.

Enumerations

An enumeration is a user-defined type that consists of a set of named values. With enumerations, algorithms can be more naturally expressed by using the language of the problem set rather than having to manually map values from another type (such as `Int`) to some representation of the problem set.

For example, you might want to store information about the class of travel for a ticket. Without enumerations, you might use “1” to represent first class, “2” to represent business class, and “3” to represent economy class.

Using an enumeration, you could instead represent the classes as named values such as `first`, `business`, and `economy`.

Enumerations in Swift are considerably enhanced compared to their counterparts in C-based languages, and they share many characteristics with classes and structures. Following are some of the notable similarities and differences:

- Enumerations can have computed properties (but not stored properties), instance and type methods, and initializers (see the relevant sections in [“Classes” on page 106](#)).
- Enumerations support extensions and protocols (see the sections [“Extensions” on page 155](#) and [“Protocols” on page 162](#)).
- Enumerations can’t inherit or be derived from other enumerations and can’t have deinitializers.

The syntax for declaring a new enumeration is as follows:

```
enum EnumName {  
    // list(s) of enumeration member values  
}
```

Using the travel class analogy, the example thus becomes:

```
enum TravelClass {  
    case First
```

```
    case Business
    case Economy
}
```

You can write the same definition more concisely like this:

```
enum TravelClass {
    case First, Business, Economy
}
```

Unlike C, enumerations in Swift are not assigned equivalent integer values, but such values can be optionally assigned to them (see the next section, “**Raw Member Values**”).

Once defined, enumerations are used much like any other type:

```
var thisTicket = TravelClass.First
var thatTicket: TravelClass
thatTicket = .Economy
```

Note that dot syntax is required when referring to a named value from an enumeration. In the second assignment in the preceding example, the enumeration name is omitted because it can be inferred from the variable type, but the dot is still required.

Raw Member Values

In C, each member of an enumeration has an underlying integer value, and you can use that value in place of the member name. Swift does not assign values to enumeration members by default, but you can include values in the definition. These are called *raw values*. Moreover, raw values aren’t limited to integer values; they can be strings, characters, or floating-point values, but all raw values for a given enumeration must be the same type.

This example declares an enumeration with raw values of Int type:

```
enum AtomicNumber: Int {
    case Hydrogen = 1
    case Helium = 2
    case Lithium = 3
}
```

```
    case Beryllium = 4
}
```

For enumerations where the raw value is of `Int` type, successive members will be given auto-incrementing raw values if no value is provided:

```
enum AtomicNumber: Int {
    case Hydrogen = 1, Helium, Lithium, Beryllium
}
// Helium = 2, Lithium = 3, Beryllium = 4
```

The `rawValue` property in this example gives you access to a member's raw value:

```
AtomicNumber.Lithium.rawValue
// returns 3

var mysteryElement = AtomicNumber.Helium
mysteryElement.rawValue
// returns 2
```

The `(rawValue: n)` initializer lets you translate a raw value back its enumeration value, if one exists. Because there might not exist a member with the specified raw value, this returns an optional value, which must be unwrapped:

```
if let r = AtomicNumber(rawValue: 2) {
    print(r)
} else {
    print("no match for raw value 2")
}
```

In the previous example, note that the text “Helium” will be output, but `r` is not a string. Swift 2.0 added reflection information to enumerations so the `print` statement displays something useful. You can convert an enumeration to a string using string interpolation. For example:

```
var s: String
if let r = AtomicNumber(rawValue: 3) {
    s = "\(r)"
} else {
    s = ""
}
```


Associated Values

Raw member values are invariant (i.e., they are constant values that are associated with each enumeration member). Enumerations in Swift support another kind of value called an *associated value*. These are more like properties of a class—you can set them differently for each instance of the enumeration.

You define associated values for an enumeration as follows:

```
enum EnumName {
    case MemberName(SomeType [, SomeType...])
    case AnotherMemberName(SomeType [, SomeType...])
}
```

The associated value(s)—expressed as a tuple—can have one or more values, each of which can be of a different type.

Let's consider a concrete example. The term “network address” can be generically used to mean an address for the type of protocol being considered, even though network addresses for two different protocols might look very different from each other.

For example, an Ethernet MAC address consists of six 2-digit hex values separated by colons (e.g., “00:01:23:45:CD:EF”), whereas an IPv4 consists of four 8-bit unsigned values (or octets). You can represent this as an enumeration with associated values like this:

```
enum NetworkAddress {
    case MAC(String)
    case IPv4(UInt8, UInt8, UInt8, UInt8)
}
```

When a variable of this enumeration type is defined, you can associate IP addresses with each IPv4 case, and MAC addresses with each MAC case:

```
var routerAddress = NetworkAddress.IPv4(192, 168, 0, 1)
var dnsServerAddress = NetworkAddress.IPv4(8, 8, 8, 8)
var ethernetIF = NetworkAddress.MAC("00:DE:AD:BE:EF:00")
```

Note that the associated value is stored with the instance; it is not part of the enumeration. You can even reassign a different

type of network address to an existing variable of that type and store a different type of associated value:

```
var someAddress = NetworkAddress.IPv4(192, 168, 0, 1)
someAddress = NetworkAddress.MAC("00:DE:AD:BE:EF:00")
someAddress = NetworkAddress.IPv4(10, 10, 0, 1)
```

To check for different types of network addresses, use a switch statement:

```
someAddress = NetworkAddress.IPv4(10, 10, 0, 1)
switch someAddress {
    case .MAC:
        print ("got a MAC address")
    case .IPv4:
        print ("got an IP address")
}
// prints "got an IP address"
```

To access the associated value, use a switch statement with let value binding:

```
someAddress = NetworkAddress.MAC("00:DE:AD:BE:EF:00")
switch someAddress {
    case let .MAC(theaddress):
        print ("got a MAC address of \(theaddress)")
    case let .IPv4(a, b, c, d):
        print ("got an IP address with" +
            "a low octet of \(d)")
}
// prints "got a MAC address of 00:DE:AD:BE:EF:00"
```

Methods in Enumerations

Enumerations can have instance methods, which you define using the same syntax as when defining classes. The following example extends the `NetworkAddress` enumeration to include a method `printable()` that returns the associated value of either enumeration type as a string:

```
enum NetworkAddress2 {
    case MAC(String)
    case IPv4(UInt8, UInt8, UInt8, UInt8)

    func printable() -> String
    {
        switch self {
```

```

        case let .MAC(theAddress):
            return theAddress
        case let .IPv4(a, b, c, d):
            return ("\(a).\(b).\(c).\(d)")
    }
}

```

You use this as follows:

```

var deviceAddress = NetworkAddress2.IPv4(192, 168, 0, 1)
deviceAddress.printable()
// returns "192.168.0.1"
deviceAddress = NetworkAddress2.MAC("00:DE:AD:BE:EF:00")
deviceAddress.printable()
// returns "00:DE:AD:BE:EF:00"

```

By default, instance methods defined in an enumeration are not able to modify the instance's value, but you can enable this behavior by defining the instance method as a *mutating method* (see the subsection “Mutating methods” on page 140).

Type Methods for Enumerations

To declare a type method for an enumeration, you precede the method's definition with the keyword `static` (in contrast to classes, where the keyword `class` is used), as shown in the following example:

```

enum SomeEnumeration {
    static func someTypeMethod()
    {
        // implementation
    }
}

```

Type methods defined in enumerations can access other type methods and type properties defined for the same enumerations (indicated with the keyword `static`).

To call a type method for an enumeration, precede it with the enumeration name using dot syntax, like so:

```

SomeEnumeration.someTypeMethod()

```

Note the following additional issues when using type methods in enumerations:

- A type method cannot access associated values or instance methods because it is not called in relation to an instance, but it can call other type methods in the enumeration.
- To call a type method from another type method within the same enumeration, you can omit the enumeration name because it is equivalent to `self`.

Recursive Enumerations

Swift 2.0 introduced support for *recursive* enumerations—i.e., enumerations that can contain cases with associated values that are instances of the same enumeration. Such enumerations must be annotated with the keyword `indirect`.

Following is an example that defines an enum `List`, which can either be empty, or can contain a head value (an `Int`) followed by a tail value (a `List`):

```
indirect enum List {
    case Empty
    case SubList(head: Int, tail: List)
}

let list1 = List.SubList(value: 4, tail: List.Empty)
let list2 = List.SubList(value: 1, tail: list1)
let list3 = List.SubList(value: 2, tail: list2)
print (list3)
// will output:
// SubList(2, List.SubList(1, List.SubList(4, List.Empty)))
```

Failable Initializers in Enumerations

Failable initializers are available in enumerations.

For more information, see [“Failable initializers” on page 135](#), which discusses failable initializers in the context of classes.

Access Control

Many object-oriented languages feature a method of access control for limiting access to members of classes. For example, C++ supports *public*, *protected*, and *private* access levels for data and function members of a class or structure.

Swift provides a similar mechanism, but it extends it to provide access control for higher-order types (classes, structures, and enumerations) as well as their members, and for globally defined values, such as functions and variables, type aliases, protocols, extensions, and generics.

The access control levels provided by Swift are *public*, *internal*, and *private*. Following are descriptions of each and some limitations of their use:

- Public entities are accessible from any source file in the module in which they are defined as well as any source file that imports that module.
- Internal entities are accessible from any source file in the module in which they are defined, but not from elsewhere. Internal access is the default access level applied in most cases for which access control is not otherwise specified.
- Private entities are only accessible within the source file in which they are defined (and thus are not even accessible from other source files that are part of the same module).
- It is not possible to specify an access level for an entity that is more open than the access level of its type. For example, if the class *SomeClass* has an access level of internal, it is not possible to define an instance of this class and assign it an access level of public.
- It is not possible to specify an access level for a member of an entity that is more open than the entity itself. For example, if the class *SomeClass* has an access level of

internal, it is not possible for a member of the class to have an access level of public.

Specifying Access Control Levels

You specify access control levels by preceding the entity to which they refer with one of the keywords, `public`, `internal`, or `private`, as in the following:

```
public let APIVersion = "1.0.0"
private struct HashingRecord { }
internal class Byte { }
```

The access level specified for a type affects the default access level for its members:

- If a type's access is marked as `private`, its members will default to `private`.
- If a type's access is marked as `public` or `internal`, its members will default to `internal`—you need to explicitly declare members as `public` if you want them visible outside the current module, even if the containing entity itself is marked as `public`.
- If a type's access is not specified, it, and its members, will default to `internal`.

In the earlier section on classes a `Byte` class was introduced to demonstrate the use of the `subscript` function, which provided access to `Byte` instances as though they were an array of bits. If this class was to be included in a framework for other programs to use, it would be necessary to make the class and its `subscript` member `public`, but it would also be reasonable to mark the variable that stores the byte value as `private`, so the implementation is opaque to callers, as in this example:

```
public class Byte
{
    private var d: UInt8 = 0

    public subscript(whichBit: UInt8) -> UInt8 {
```

```

    }
    // rest of subscript definition
}

```

Other code that imported this library module would be able to create instances of the `Byte` class, and could set and get their value by subscript, but could not access the `d` property directly.

Default Access Control Levels

Although the default access level for most entities is internal, there are some exceptions and caveats you might need to consider, which are presented in [Table 6](#).

Table 6. Access control restrictions

Type	Default/available access level(s)
Constants, variables, properties	Constants, variables, and properties must have either the same access level as their type, or a more restrictive level. For example, a variable of type <code>SomeClass</code> that is marked <code>internal</code> cannot itself be marked as <code>public</code> .
Enumeration cases	The access level of the cases of an enumeration is the same as the access level of the enumeration itself (enumeration cases are not members in the usual sense).
Enumeration values	The default access level is the same as the access level of the enumeration with which the values are associated and cannot be overridden with a more restrictive access level.
Extensions	When extending a class, structure, or enumeration, new members defined in the extension have the same default access level as members of the original entity: <ul style="list-style-type: none"> You can override the default for new members by specifying a different access level on the extension. The access level for new members can override the default but cannot be more open than the access level of the original entity. You cannot specify an access level on an extension that adds protocol conformance.

Type	Default/available access level(s)
Function	The default access level is the most restrictive of all of the function's parameter and return types. You can override the default but only with a more restrictive level.
Generics	The effective access level for generics is the most restrictive of the access level of the generic itself as well as the access level of any of its constraining types.
Getters, setters	The default access level for getters and setters is the same as the access level for the entity on which they are defined. The setter can have a more restrictive level than the getter (limiting modification of the entity without affecting its readability), which is specified by preceding the variable or property name with either <code>private(set)</code> or <code>internal(set)</code> .
Initializers	The default access level for initializers is the same as the access level of the class to which they belong. You can override the default for a custom initializer with a more restrictive level.
Nested types	<ul style="list-style-type: none"> • Types defined within private types will also be private. • Types defined within internal types will default to the internal access level but can be made private. • Types defined within public types will default to the internal access level, and you can override with any access level.
Protocols	The default access level is internal, but you can override this with any level. Each requirement within the protocol has the same access level as the protocol itself, and this cannot be overridden.
Subclasses	The default access level is the same as that of the superclass, but you can override this with a more restrictive level. A subclass can override the implementation of an inherited class member and override that member's access level as long as it is not more open than the access level of the subclass.
Subscripts	The default access level for a subscript is the most restrictive of its index and return types. You can override this only with a more restrictive level.

Type	Default/available access level(s)
Tuple	The only available access level is the most restrictive level of all of the types that comprise the tuple. You cannot override this because tuples are not explicitly defined like other types.
Type aliases	The default access level for a type alias is the same as that of the type that it aliases. You can override the default with a more restrictive level.

Extensions

Swift's extensions mechanism makes it possible for you to add new functionality to existing classes, structures, and enumerations, even if you did not create them yourself, and you do not have their source code. This is a hugely powerful feature and one that opens opportunities for extending Swift itself—if the language is missing a feature you need, you can often add it yourself as an extension.

The basic syntax of an extension is as follows:

```
extension Type
{
    // code that extends Type
}
```

Extensions can only add new functionality to existing types; they cannot override existing properties, methods, subscripts, or any other functionality already defined in a type.

Computed Property Extensions

You can use extensions to add computed type properties and computed instance properties to an existing type, but you cannot use them to add stored properties or property observers.

Here is a simple extension that adds a computed property to the `Int` class that returns a hex representation of the integer as a string:

```
extension Int
{
```

```

var asHex: String {
    var temp = self
    var result = ""
    let digits = Array("0123456789abcdef".characters)
    while (temp > 0) {
        result = String(digits[Int(temp & 0x0f)])
            + result
        temp >>= 4
    }
    return result
}
}

```

With this extension in place, you can query the `asHex` property of any `Int` type to get its hex equivalent as a string:

```

45.asHex
// returns "2d"
var s = 100.asHex
// stores "64" in a new String variable s

```

Initializer Extensions

Extensions can be used to add convenience initializers to a class, but you cannot add designated initializers or deinitializers.

Extensions can also be used to add initializers to structures. If the structure does not define its own initializers, your extension initializer can call the default member-wise initializer if required to set all default property values.

Method Extensions

You can use extensions to add instance methods and type methods to an existing type. The following example extends the `Int` type to provide a facility for converting an integer to a fixed-width string with leading spaces:

```

extension Int
{
    func asRightAlignedString(width: Int) -> String
    {
        var s = "\(self)"
        while (s.characters.count) <= width) {

```

```

        s = " " + s
    }
    return s
}

let x = -45
x.asRightAlignedString(5)
// returns "  -45"

```

An instance method added with an extension can modify the instance with the `mutating` keyword. This example extends the `Double` type with a method that truncates a double to its nearest integer value that is not larger than the original value:

```

extension Double
{
    mutating func trunc()
    {
        self = Double(Int(self))
    }
}

var d = 45.5
d.trunc()
// d is now 45.0

```

Subscript Extensions

Here is an example that extends the `String` class to support subscripted character referencing:

```

extension String
{
    subscript (i: Int) -> Character {
        return Array(self.characters)[i]
    }
}

```

To use this, follow a string with a subscript:

```

"Hello"[4]
// returns "o"
var a = "Alphabetical"
a[0]
// returns "A"

```

Earlier, the section on classes presented a `Byte` class that included a subscript definition with which you could treat a `Byte` object as an array of bits. Using extensions, you can apply the same feature to the `UInt8` type, as demonstrated here:

```
extension UInt8
{
    subscript(whichBit: UInt8) -> UInt8 {
        get { return (self >> whichBit) & 1 }
        set {
            let mask = 0xFF ^ (1 << whichBit)
            let bit = (newValue & 1) << whichBit
            self = self & mask | bit
        }
    }
}

var b: UInt8 = 0
b[0] = 1
b[7] = 1
b
// returns 129
```

Checking and Casting Types

Swift is a strongly typed language, but there are times when some relaxation of type rules is warranted, and there are times when you want to check what an object's type is or downcast a reference to a subclass type.

Using Swift's `is`, `as`, `as!`, and `as?` type casting operators, you can test types and protocol conformance, and downcast types and protocols.

Any and AnyObject

One Swift mechanism that provides type flexibility comes via the keywords `AnyObject` and `Any`, which are special built-in type aliases. `AnyObject` can represent an instance of any class, whereas `Any` can represent an instance of any type except for function types.

You can use these two type aliases to create complex entities. For example, an array that can store any type of data:

```
var v = [Any]()
v.append(2)
v.append(3.4)
v.append("crunch")
```

Or you can create a function that can take an instance of any class type:

```
func someFunc(t: AnyObject)
{
    // do something with t
}
```

Another example of where `AnyObject` is required is when calling Objective-C APIs that return an `NSArray`. Because an Objective-C array can contain arbitrary object types, it must be represented in Swift as an array of type `[AnyObject]`. To work with such an array, you will likely need to use Swift's type-casting operators to cast references to the array entries to an appropriate Swift class, such as a `String`.

Checking Types

You use the `is` operator to check whether an instance is of a specific type. Consider the following example of three classes A, B, and C. Note that B is a subclass of A:

```
class A { }
class B: A { }
class C { }

var a = A()
var b = B()
var c = C()
```

The following function takes an instance of any object as a parameter and uses the `is` operator to check whether it is an instance of class A:

```
func typeCheck(t: AnyObject) -> Bool
{
    return t is A
}
```

```
}  
  
typeCheck(a) // true  
typeCheck(b) // true  
typeCheck(c) // false
```

The first `typeCheck` call returns `true` because `a` is an instance of class `A`. The second call also returns `true`, because `b` is an instance of `B`, which in turn is a subclass of `A`. The third call returns `false` because `c` is not of type `A` or a subclass of `A`.

Downcasting Types

Using downcasting, you can treat an instance of a class as an instance of one of its subclasses.

Consider the scaffolding in the example that follows for a system for representing geometric shapes. The example starts with a generic base class called `Shape` and then defines specific shapes as subclasses of the base class to represent circles and squares. Note that the subclasses have different functionality. In this contrived example, one has a `describe` method, while the other has an `identify` method—a small difference purely for the sake of demonstration:

```
class Shape { }  
class Square: Shape  
{  
    func describe()  
    {  
        print("I am a square")  
    }  
}  
class Circle: Shape  
{  
    func identify()  
    {  
        print("I am a circle")  
    }  
}
```

Now, an array is defined for storing shapes, some of which are then added to it:

```
var shapes = [Shape]()

let sq = Square()
let ci = Circle()

shapes.append(sq)
shapes.append(ci)
```

The array is defined to store generic shapes, but because both `Circle` and `Square` are subclasses of `Shape`, they too can be stored in the array.

With this structure in place, you now want to create a general-purpose function to do something with the array of shapes (e.g., drawing them on a display or calling some other subclass-specific method). As you walk through the array, you might want to know the type of each member (whether it's a circle or a square, or something else), but you might also want to be able to treat each member as its subclass type (rather than the generic type of the array) so you can use features unique to each subclass. For you can use the `as` and `as?` downcast operators.

The `as!` operator forcibly downcasts to a specific subclass type. If the object you're trying to downcast is not actually of the specified class or is not a subclass of the specified class, Swift will terminate with a runtime error.

The `as?` operator attempts to downcast to a specific subclass type and returns an optional value. If the downcast fails (i.e., the object you're trying to downcast is not of the specified class or a subclass of the specified class), the value returned is `nil`. If the downcast succeeds, the returned value is the type to which it was downcast.

Suppose you wanted to walk through the shape array searching for a specific type of entry. Even though you could do that by using the `is` operator, here's how you would do it using the `as?` operator and `let` binding:

```
for s in shapes
{
    if let c = s as? Circle {
```

```

        // c is now a reference to an array entry downcast
        // as a circle instead of as a generic shape
    } else {
        // downcast failed
    }
}

```

Alternatively, you could use a `switch` statement to achieve a similar goal:

```

for s in shapes
{
    switch s {
        case let cc as Circle:
            cc.identify()
        case let ss as Square:
            ss.describe()
        default:
            break;
    }
}

```

Note that the preceding example again used `let` binding so that you have a reference to the array entry, but the reference is cast to the subclass type so that `cc` refers to a `Circle`. This allows you to call methods unique to the `Circle` type, whereas `ss` refers to a `Square`, and lets you call methods unique to the `square` type.

If you're sure a downcast won't fail and you don't want to use `let` binding, you can use the `as!` operator to forcibly downcast from the generic class to the subclass like this:

```

for s in shapes {
    if s is Circle {
        let c = s as! Circle
        c.identify()
    }
}

```

Protocols

A protocol defines a standard set of features, expressed via properties, methods, operators, and subscripts, that embody a specific role or provide a specific set of functionality. A proto-

col isn't the implementation of this functionality; it just describes what must be implemented.

A class, structure, or enumeration can subsequently *adopt* the protocol, which implies the class, structure, or enumeration will include its own definitions of the features the protocol declares, thereby *conforming* to the protocol.

NOTE

Swift 2.0 allows you to use protocol extensions to provide default implementations of methods and properties for all conforming types. See “[Protocol Extensions](#)” on page 173 for more information.

The syntax for declaring a new protocol is as follows:

```
protocol ProtocolName
{
    // protocol definition
}
```

To restrict a protocol such that it can only be adopted by classes (and not by structures or enumerations), add a `class` annotation, as in this example:

```
protocol ProtocolName: class
{
    // protocol definition
}
```

The syntax for adopting one or more protocols (in this case, for a class) is as follows:

```
class ClassName : [SuperClassName]
    ProtocolName [, ProtocolName...]
{
    // class definition and code that
    // implements the requirements of ProtocolName(s)
}
```

If the class is derived from an existing class, the superclass name appears before any protocol names.

The protocol body consists of a series of declarations that define the requirements that must be implemented in order for the adopter to conform to the protocol. This includes a list of the required properties and methods the adopter must define and any constraints on them (e.g., whether a property is read-only or read/write).

Required Properties

A property requirement for a protocol specifies both the name of the property and its type as well as whether the property is read-only or read/write. It does not specify whether the property is implemented as a stored property or as a computed property, because that is up to the adopter.

You declare properties by using `var` in this fashion:

```
protocol SomeProtocol
{
    var aWritableProperty: Double { get set }
    var aReadOnlyProperty: Int { get }
    static var aTypeProperty: String { get set }
}
```

This example also demonstrates how the type of a property is specified and whether it must be implemented as a read-only property (by using just the `get` keyword) or read/write (by using both the `get` and `set` keywords).

To conform to this protocol, an adopter must define a read/write property named `aWritableProperty`, a read-only property named `aReadOnlyProperty`, and a read/write static (type) property named `aTypeProperty`.

Note that the `static` keyword is used to specify a type property (as opposed to an instance property), even if the protocol will be adopted by a class (see [“Static properties” on page 116](#) for more information).

Required Methods

A method requirement for a protocol specifies the name, parameters, and return type of the method. You write it in the same way as you would define an ordinary method in a class, except for the method body. This next example defines a protocol that requires adopters to implement a method that returns a (presumably) printable string:

```
protocol Printable
{
    func printable() -> String
}
```

A class adopting this protocol must include a method that returns a printable representation of the instance.

A required method in a protocol can be an instance method (as shown in the example) or a type method (by preceding it with the keyword `class`).

If a required method needs to be able to mutate the instance that it refers to, precede it with the keyword `mutating` (see the subsection “[Mutating methods](#)” on page 140).

Optional Methods and Properties

You can use the keyword `optional` in front of a method or property name in a protocol to indicate the method or property does not have to be implemented by a class that adopts that protocol.

There is an important restriction, though, in the use of the `optional` keyword: the protocol definition must be prefixed with the `@objc` keyword, even if there is no intention to interact with Objective-C code or data. This immediately places further restrictions on what you can do with the protocol, notably the following:

- `@objc` protocols can only be adopted by classes, not by structures or enumerations.

- The module must import the Foundation framework.
- You cannot use generic types in the protocol (using the `typealias` keyword, as described in the section “[Generic Protocols](#)” on page 200).
- You cannot use any Swift data type in the protocol that has no Objective-C equivalent (so, for example, the protocol cannot define anything that uses or requires a tuple).

The example that follows demonstrates a protocol that defines an optional property and an optional method. The property is defined as a read-only string (because only the `get` keyword is present), and the method is defined as taking an integer parameter and returning an integer string:

```
@objc protocol Optionals
{
    optional var optProperty: String { get }
    optional func optMethod(i: Int) -> String
}
```

The next step is to define two classes that adopt the protocol. The first only implements the optional property; the second only implements the optional method. Note that in both cases, the implementation of the optional entity must be annotated with a preceding `@objc` keyword:

```
class ImplementsProperty: Optionals
{
    @objc let optProperty = "I'm a property!"
}

class ImplementsMethod: Optionals
{
    @objc func optMethod(i: Int) -> String
    {
        return "I'm a method and was passed \(i)"
    }
}
```

Finally, the following code creates an instance of each class to demonstrate how to access the optionally defined features.

In doing so, note that the variables `a` and `b` are declared as `Optionals` (the protocol), not of either of the class types:

```
var optA: Optionals = ImplementsProperty()
var optB: Optionals = ImplementsMethod()
```

Looking first at the instance `optA`, you might expect that you can reference it directly, and, technically, you can:

```
optA.optProperty
// returns an optional String? "I am a property!"
optB.optProperty
// returns nil
```

Notice, though, that the returned value is an optional string, even though the property is declared in the protocol as a non-optional string, and the class actually implements the property. Because optional methods and properties might not have been implemented in an adopting class, they *always* return an optional value. Thus, you must test for a non-`nil` value and then unwrap it to safely use the property value.

Because you can access the property, you might wonder what happens if you try to access the (unimplemented) method:

```
optA.optMethod(45)
// compiler error
```

Rather than returning an optional value of `nil`, this code generates an error because the method hasn't actually been implemented. Instead, you must call the method with a question mark immediately following its name, as in this example:

```
optA.optMethod?(45)
// returns nil
```

When called this way, you do get a `nil` value—in this case, indicating the method has not been defined.

Finally, consider the instance `optB`, created earlier, that implements the optional method but not the optional property:

```
optB.optMethod?(1)
// returns an optional String?
// "I am a method and was passed 1"
```

```
optB.optProperty
// returns nil
```

This time the method return (defined as a nonoptional string) is wrapped in an optional because the method implementation itself is optional, whereas the property, which has not been implemented, returns `nil`.

When a protocol is defined with an optional method or property, you can also use optional chaining to access the optional entities. See also the sections “Optionals” on page 81 and “Optional Chaining” on page 85 for more information.

Initializers in Protocols

Initialization methods, including failable initializers, can be included in protocol definitions, i.e., conforming types must implement those initializers.

When implementing a protocol-defined initializer in a class that adopts that protocol, note the following:

- The initializer can be implemented as either a convenience or a designated initializer.
- The initializer definition must be preceded by the keyword `required`, and must be overridden in subclasses to ensure they conform.
- A failable initializer requirement can be implemented as failable or nonfailable.

Adopting Protocols with Extensions

Protocol use isn't limited to classes, structures, and enumerations you define yourself: you can use Swift's extension mechanism (see “Extensions” on page 155) to conform any existing type to a protocol, including simple built-in types such as `Int`, `Double`, and `String`.

The basic syntax for defining such an extension is as follows:

```

extension Type : ProtocolName [, ProtocolName...]
{
    // code that extends Type and implements
    // the requirements of ProtocolName(s)
}

```

Following is an example of using an extension to conform the Bool type to the Printable protocol that was introduced earlier. This example is somewhat contrived because the global print function will print Bool types as either “true” or “false,” so this implementation is defined to return “YES” and “NO” instead:

```

extension Bool: Printable
{
    func printable() -> String
    {
        return self ? "YES":"NO"
    }
}

```

Here’s how to use this with Bool types:

```

var a = false
a.printable()
// returns "NO"

```

Inheritance and Protocols

In the same way you can use inheritance with classes, you can use inheritance with protocols—one protocol inherits all of the requirements of another and then adds further requirements of its own.

The syntax for basing one protocol on another is as follows:

```

protocol ProtocolName : ProtocolName [, ProtocolName...]
{
    // protocol definition
}

```

The following example demonstrates a simple protocol that requires that adopters implement the method asHex() to return a hex representation of the type, but because it inherits from the Printable protocol (described earlier), adopters must also implement a method named printable():

```
protocol Hexable : Printable
{
    func asHex() -> String
}
```

And here is an extension to the Bool type that adopts the Hexable protocol and implements the required methods:

```
extension Bool: Hexable
{
    func asHex() -> String
    {
        return self ? "1":"0"
    }

    func printable() -> String
    {
        return self ? "YES":"NO"
    }
}

var b = true
b.asHex()
// returns "1"
b.printable()
// returns "YES"
```

NOTE

For the purpose of demonstration, we'll assume the reader is happy to forgive the contrivance that "1" is the hex equivalent of true and that "0" is the hex equivalent of false.

Using a Protocol as a Type

A protocol is a type in the same way a class is a type, and you can use protocols in most places you can use types. This is a hugely powerful feature, especially when you begin to think of types not in terms of what they can store (Int, String, data) but in terms of the functionality they can provide (methods, actions, *conformed behavior*).

With this mindset, you can think of a variable not as a place to store a specific type of value but as a place to store anything that implements specific *behavior*. Similarly, you can think of an array not as a place to store a collection of one type of data but as a place to store a collection of anything that implements a specific behavior.

An earlier example demonstrated an extension to the Bool type that adopted the Hexable protocol. Here's a similar extension for the Int type:

```
extension Int: Hexable
{
  func asHex() -> String
  {
    var temp = self
    var result = ""
    let digits = Array("0123456789abcdef".characters)
    while (temp > 0) {
      result = String(digits[Int(temp & 0x0f)])
        + result
      temp >>= 4
    }
    return result
  }

  func printable() -> String
  {
    return "\(self)"
  }
}
```

With both Bool and Int conforming to the Hexable protocol, you can create some interesting behavior with variables:

```
var c: Hexable = true
c.printable() // returns "YES"
c.asHex() // returns "1"
c = 45
c.printable() // returns "45"
c.asHex() // returns "2d"
```

The variable `c` is of type Hexable; it can store anything that conforms to the Hexable protocol, which means it can store both Bool and Int data.

This next example defines an array of type `Hexable` and populates it with some values:

```
var ar = [Hexable]()
ar.append(true)
ar.append(45)
ar[0].asHex()    // returns "1"
ar[1].asHex()    // returns "2d"
```

Again, the array is not limited to being able to store just `Bool` data or just `Int` data: it can store `Hexable` data, which, due to protocol conformance, includes both `Bool` and `Int` types.

Checking Protocol Conformance

You can use the `is`, `as`, and `as?` type casting operators to check for protocol conformance and to downcast protocols in the same way you can use them to check and downcast class types, because you can define protocols in terms of other protocols (inheritance) in the same way you can define classes in terms of other classes.

There is an important restriction, though, in the use of these operators in checking and downcasting protocols: the protocol definition must be prefixed with the `@objc` keyword, even if there is no intention to interact with Objective-C code or data. This is the same restriction that was described earlier in the section [“Optional Methods and Properties” on page 165](#), and it imposes the same limitations when checking for protocol conformance.

The `@objc` keyword requirement also means you cannot check if an instance conforms to a built-in Swift protocol, because those protocols aren’t defined with the `@objc` keyword.

If you can work within these restrictions, you can use the type checking and downcasting operators in the same way you would use types. For example, the following defines a basic protocol, a class that adopts that protocol, and a general function to check if any instance conforms to the protocol using the `is` keyword:

```

@objc protocol DemoProto
{
}

class DemoClass: DemoProto
{
}

func protoCheck(t: AnyObject) -> Bool
{
    return t is DemoProto
}

```

Next, create some objects that can be passed to the protocol checking function:

```

var s = DemoClass()
var i = 4
protoCheck(s) // returns true
protoCheck(i) // returns false

```

When `protoCheck()` is called with `s` as a parameter, it returns the value `true` because `s` is an instance of `DemoClass`, which adopts the `DemoProto` protocol.

When `protoCheck()` is called with `i` as a parameter, it returns `false` because `i` has been created as an `Int` type through type inferencing, and the `Int` type does not conform to the `DemoProto` protocol.

Protocol Extensions

Swift 2.0 allows you to use *protocol extensions* to provide default implementations of methods and properties for all conforming types. In other words, a protocol extension can implement functionality, and conforming types automatically gain the implementation with no further code.

A protocol extension is defined as follows:

```

extension ProtocolName
{
    // function and/or property definitions
}

```

Following is an example of a reworked version of the `asHex()` converter for unsigned integers—this time implemented as a read-only property. This version determines the storage size of the type using the `sizeof()` global function to ensure an appropriate number of leading zeros are included:

```
extension UnsignedIntegerType
{
    var asHex: String {
        var temp: UInt = numericCast(self)
        var result = ""
        let digits = Array("0123456789abcdef".characters)
        for _ in 0..sizeof(self.dynamicType) * 2 {
            result = String(digits[Int(temp & 0x0f)])
                        + result
            temp >>= 4
        }
        return result
    }
}
```

Since this type is an extension on the `UnsignedIntegerType` protocol, it is automatically available for `UInt8`, `UInt16`, and any other unsigned integer types that adopt the protocol. Without implementing any extensions on those types, we can access the property, as follows:

```
let u1: UInt8 = 16
let u2: UInt16 = 38
let u3: UInt32 = 32767
u1.asHex // returns "10"
u2.asHex // returns "0026"
u3.asHex // returns "00007fff"
```

Here are some other important aspects of protocol extensions:

- If a type implements a requirement that is already satisfied through a protocol extension, that type's implementation will override the default version in the extension.
- A protocol extension can be constrained with a `where` clause. See [“The “where” type constraint clause” on page 198](#) for more information.

Built-In Protocols

There are many built-in protocols in Swift: some are used to define the language itself, while others are useful to adopt for your own classes so you can use them in many of the same contexts as Swift's built-in types.

The first group of these to consider are the *literal convertible* protocols, which includes the following:

- `ArrayLiteralConvertible`
- `BooleanLiteralConvertible`
- `DictionaryLiteralConvertible`
- `ExtendedGraphemeClusterLiteralConvertible`
- `FloatLiteralConvertible`
- `IntegerLiteralConvertible`
- `NilLiteralConvertible`
- `StringLiteralConvertible`
- `UnicodeScalarLiteralConvertible`

An instance conforming to a literal convertible protocol can be initialized with a corresponding literal value. For example:

```
var x: Float = 21.0
// x is a float, which conforms to FloatLiteralConvertible,
// and so x can be initialized from 21.0

var freeDays: [String] = ["Sunday", "Saturday"]
// freeDays is an array of strings, which conforms to
// ArrayLiteralConvertible, so freeDays can be initialized
// from the array literal

var b: Bool = 3
// 'Int' is not convertible to 'Bool'
// b is a Bool, which does not conform to
// IntegerLiteralConvertible so b cannot be initialized from 3
```

Swift's other built-in protocols are listed in [Table 7](#).

Table 7. Built-in protocols

Protocol	Description
<code>AbsoluteValuable</code>	Conforming types implement an <code>abs()</code> (absolute value) function (implying they are signed numbers).
<code>Any</code>	A protocol to which all types implicitly conform.
<code>AnyClass</code>	A protocol to which all class types implicitly conform.
<code>AnyObject</code>	A protocol to which all classes implicitly conform.
<code>BidirectionalIndexType</code>	Inherits from <code>ForwardIndexType</code> and adds the requirement that conforming types implement the <code>predecessor()</code> method to step backwards from one value to the previous.
<code>BitwiseOperationsType</code>	A protocol that defines bitwise operations <code>&</code> (AND), <code> </code> (OR), <code>^</code> (XOR), and <code>~</code> (NOT), which is adopted by Swift's integer types.
<code>BooleanType</code>	Conforming types represent Boolean values, and can be used as conditions (such as in <code>if</code> and other control statements).
<code>CollectionType</code>	Conforming types are sequences of elements that can be addressed by position using an <code>Index</code> type (e.g., arrays).
<code>Comparable</code>	For types that can be magnitude-compared using relational operators. Conforming types need to implement at least <code><</code> and <code>==</code> . If the other relational operators are not implemented, default versions defined by the standard library will be used.
<code>CustomDebugStringConvertible</code>	Replaces the <code>DebugPrintable</code> protocol from Swift 1, but otherwise unchanged. Conforming types implement a <code>debugDescription</code> property that returns a textual representation of the type that can be written to an output stream (such as by <code>print</code>).

Protocol	Description
CustomLeaf Reflectable	Used to build a custom mirror of an instance (like CustomReflectable). Descendant classes are not represented in the mirror unless they override customMirror().
CustomPlayground QuickLookable	Used to implement custom Quick Look views in Xcode Playgrounds. Conforming types implement the method customPlaygroundQuickLook(), which returns a member of the QuickLookObject enum.
CustomReflectable	Used to build a custom mirror of an instance, which allows you create alternate views of your data. Conforming types implement a customMirror() function that returns a Mirror object. (Mirrors are used by Playgrounds and by the Xcode debugger).
CustomStringConvertible	Replaces the Printable protocol from Swift 1, but otherwise unchanged. Conforming types implement a description property that returns a textual representation of the type that can be written to an output stream (such as by print).
Equatable	For types that can be compared for equality using == and !=.
ErrorType	Conforming types can be used as arguments to throw and catch.
FloatingPointType	Defines a set of common requirements for Swift's floating-point types.
ForwardIndexType	Instances of conforming types can represent discrete values in a series, and implement the successor() method to step from one value to the next.

Protocol	Description
GeneratorType	Conforming types can be used to iterate over a sequence, and must implement a function <code>next()</code> that returns the next element in the sequence, or <code>nil</code> if no further elements exist. See “The GeneratorType Protocol” on page 180 .
Hashable	Conforming types can be used as keys in a Dictionary, and must implement an <code>Int</code> property <code>hashValue</code> , which returns a hashed version of an instance.
Indexable	Conforming types can be accessed via a subscripted <code>Index</code> value.
IntegerArithmeticType	Defines arithmetic operations <code>+</code> (add), <code>-</code> (subtract), <code>*</code> (multiply), <code>/</code> (divide), and <code>%</code> (modulus), which is adopted by Swift’s integer types.
IntegerType	Defines a set of common requirements for Swift’s integer types.
IntervalType	Defines an interval, comprised of a starting bound and an ending bound, and associated with the idea of containment. See also “Intervals” on page 207 .
LazyCollectionType	A collection on which methods (like <code>map</code> and <code>filter</code>) are implemented lazily. They compute elements on demand using the underlying storage, rather than all at once and by creating a mutated copy of the original collection.
LazySequenceType	A sequence on which methods (like <code>map</code> and <code>filter</code>) are implemented lazily. They compute elements on demand using the underlying storage, rather than all at once and by creating a mutated copy of the original sequence.
MutableCollectionType	Conforming types are collections that support assignment via subscripts.
MutableIndexable	Conforming types can be read and written via a subscripted index value (e.g., the <code>Array</code> type).

Protocol	Description
<code>MutableSliceable</code>	Conforming types are collections with mutable slices (e.g., the <code>Array</code> type). See “ Slices ” on page 56.
<code>OptionSetType</code>	Defines conformance to the <code>SetAlgebraType</code> protocol for types whose <code>RawValue</code> is a <code>BitwiseOperationsType</code> . See “ Option Sets ” on page 67.
<code>OutputStreamType</code>	Conforming types can be a target of a text stream. <code>String</code> adopts this protocol, meaning you can <code>print()</code> to a string.
<code>RandomAccessIndexType</code>	Conforming types are indexes that can be offset by arbitrary amounts in constant time. This includes all of the integer types.
<code>RangeReplaceableCollectionType</code>	Conforming types are collections that support replacement of an arbitrary range of their elements with the elements of another collection.
<code>RawRepresentable</code>	A type that can be converted to an associated raw type, and then back to an instance equivalent of the original. Used to implement <code>Option Sets</code> .
<code>ReverseIndexType</code>	An extension of <code>BidirectionalIndexType</code> that reverses the direction of <code>successor()</code> and <code>predecessor()</code> .
<code>SequenceType</code>	Conforming types are sequences of elements that can be iterated over (e.g., in a <code>for-in</code> loop), and must implement the <code>generate()</code> method that returns a generator over the elements in the sequence. Swift types that adopt this protocol include <code>Array</code> , <code>Dictionary</code> , <code>Range</code> , <code>Set</code> , and <code>strides</code> (and many others). See “ The SequenceType Protocol ” on page 184.
<code>SetAlgebraType</code>	Defines a set of common set algebra operations, such as <code>union</code> and <code>intersect</code> . See “ Sets ” on page 62.

Protocol	Description
<code>SignedIntegerType</code>	Defines a set of common requirements for Swift's signed integer types.
<code>StringInterpolationConvertible</code>	Conforming types can be initialized with string interpolation, i.e., " <code>\(. . .)</code> ".
<code>SignedNumberType</code>	Conforming types that can be subtracted, negated, or initialized from 0.
<code>Streamable</code>	Conforming types can be written to an output stream (this includes the <code>Character</code> , <code>String</code> , and <code>UnicodeScalar</code> types).
<code>Strideable</code>	Conforming types represent a continuous sequence of values that can be offset and measured.
<code>UnicodeCodecType</code>	Defines requirements for translating between sequences of Unicode code units and Unicode scalar values.
<code>UnsignedIntegerType</code>	Defines a set of common requirements for Swift's unsigned integer types.

The GeneratorType Protocol

A *generator* is an instance that conforms to the `GeneratorType` protocol. It carries state (i.e., a “current” value) and includes a method `next()`, which advances the current state to the next state, and returns it.

Generators are used to build *sequences*, and there exist many generators in Swift that implement basic aspects of the language, including ranges, strides, intervals, enumeration, and iteration.

One way to create a generator is with the `anyGenerator()` global function. This takes a trailing closure, and returns an instance of type `AnyGenerator`, which you save in a variable or constant, or pass around as a parameter. The closure typically captures an in-scope variable and implements the functionality

of the `next()` method using it. The closure returns either the updated value, or `nil` to indicate that the sequence has ended.

Following is an example of a very basic generator:

```
var state = 0
let intSeq = anyGenerator { state++ }

intSeq.next() // returns 0
intSeq.next() // returns 1
intSeq.next() // returns 2
state         // returns 2
```

This creates a generator (in this case, of type `AnyGenerator<Int>`), which is saved as the constant `intSeq`, and which produces a sequence of integers. The closure used to define the behavior of the generator's `next()` method captures the integer variable `state`, and increments and returns it every time it is called.

Used in this way, generators are of limited use, but since closures capture state, they can be used even after the scope that created them has exited. Consider this slightly more complex example:

```
var intSeq2: AnyGenerator<Int>
do {
    var n = 10
    intSeq2 = anyGenerator { n++ }
}

intSeq2.next() // returns 10
intSeq2.next() // returns 11
intSeq2.next() // returns 12
```

In this example, the generator is created inside a `do` scope using a local variable `n` defined inside that scope as the generator state value. Even though `n` has gone out of scope when the generator is actually used, the closure keeps it alive and the generator continues to return new values in the sequence each time it is called.

The following example shows a more complex generator that advances its state by 7 on each iteration, and returns `nil` to

indicate the end of the sequence after 12 iterations. This example intentionally shows a more verbose closure definition (it could be simplified). The closure indicates that it takes no parameters and returns an optional `Int?`, and includes two return statements to return either the next value in the sequence, or `nil`.

```
var stepState = 0
let stepBy7 = anyGenerator {
    () -> Int? in
    if (stepState < (12*7))
    {
        stepState += 7
        return stepState
    }
    return nil
}
```

Since this generator is bound (it stops producing values after 12 iterations), it can be used in more interesting contexts, like this:

```
let a = Array(stepBy7)
// a = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84]
```

This works because one of the initializers for the `Array()` type takes a generator as a parameter. It's important that the generator has some notion of ending; otherwise the assignment would fill up all available memory as it tried to create an infinitely large array.

After the array has been initialized, the generator is exhausted—it can't be used to produce a second identical sequence because the state variable now has the value 84, which is the stopping condition. If the state variable is still in scope, you can reset it to refresh the generator:

```
stepState = 0
let b = Array(stepBy7)
// b = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84]
```

Following is an example function that can produce a “times table.” If you are of a certain age, it's likely you rote-learned these in the earlier years of your schooling!

```

func timesTable(which: Int) -> [Int]
{
    var i = 0
    return Array(anyGenerator
        { return i++ < 12 ? i * which : nil })
}
timesTable(1)
// returns [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
timesTable(2)
// returns [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]

```

Another way to create a generator is to define a class, struct, or enum that adopts the `GeneratorType` protocol, which has only two requirements:

```

protocol GeneratorType {
    typealias Element
    mutating func next() -> Element?
}

```

The `typealias` requirement identifies the type the generator will work with, and the `next()` method requirement advances and returns the generator state, or returns `nil` to indicate no more values are available. Following is an example of a structure that adopts this protocol to implement “times tables” in a different way than the earlier examples:

```

struct timesTable2 : GeneratorType
{
    var table: Int
    var state: Int = 0

    init(_ table: Int) { self.table = table }

    typealias Element = Int
    mutating func next() -> Element?
    {
        return (state++ < 12) ? state * table : nil
    }
}

var ttGen = timesTable2(6)
while let i = ttGen.next() {
    print (i, terminator: ", ")
}
// output
// 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72,

```

The struct defines an initializer to establish which particular table is being generated, and the `while` loop uses optional binding to end once the generator is exhausted.

The SequenceType Protocol

A *sequence* is an instance that conforms to the SequenceType protocol, and can be iterated over (e.g., with a `for-in` loop). Many Swift types adopt this protocol, including arrays, dictionaries, ranges, sets, and strides. Sequences are dependent on generators.

To conform to the SequenceType protocol you need to implement the following requirements:

- A `generate()` method that returns an instance of a generator.
- A type alias `Generator` for the generator's type. This can often be inferred from the `generate()` method, so it is not always required.

Following is a reimplement of the earlier “times table” structure that now conforms to SequenceType:

```
struct timesTable3 : SequenceType
{
    var table: Int
    init(_ table: Int) { self.table = table }

    func generate() -> AnyGenerator<Int>
    {
        var state = 0
        return anyGenerator(
            { state++ < 12 ? state * self.table : nil }
        )
    }
}
```

With a sequence defined, you can now create instances and do things such as:

```
var f = timesTable3(7)
for i in f {
    print (i, terminator: ", ")
}
```

```

}
// outputs 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84,

for i in timesTable3(4) {
    print (i, terminator: ", ")
}
// outputs 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48,

```

This isn't substantially different than what you can do more directly with generators, but the utility of sequences comes with the many default method implementations the protocol defines. These include methods like `filter()`, `map()`, `reduce()`, and `sort()`. For example:

```

let c = timesTable3(3).map({$0*2}).filter({$0%4==0})
// c is [12, 24, 36, 48, 60, 72]

timesTable3(9).contains(18) // returns true
timesTable3(9).contains(12) // returns false

```

See [“Array Inherited Functionality” on page 52](#) for more examples of some of the functionality available to sequences.

Note that `SequenceType` imposes no requirements about the behavior of a sequence that is used after it reaches its terminating condition, or that is aborted before reaching its terminating condition.

The CollectionType Protocol

A *collection* is an instance that conforms to the `CollectionType` protocol. It is defined as a multipass sequence with positions that can be addressed by an associated `Index` type.

`CollectionType` adopts the `SequenceType` protocol, so much of the functionality of sequences can also be applied to collections.

In order to conform to `CollectionType`, you need to implement the following requirements:

- A type alias named `Index` for the type to be used to index the collection. This type must conform to `ForwardIndexType` (this includes the integer types).

- A property `startIndex` that represents the first valid index for the collection. This can be a constant or a variable.
- A property `endIndex` that represents the “past the end” position of the collection. This can be a constant or a variable.
- A `generate()` method that returns an instance of a generator.
- A `subscript(position)` method that returns an element from the collection based on the index value `position`.

Following is a reimplementaion of the earlier “times table” structure that now conforms to `CollectionType`:

```

struct timesTable4 : CollectionType
{
    var table: Int
    init(_ table: Int) { self.table = table }

    typealias Index = Int
    let startIndex = 1
    let endIndex = 13

    func generate() -> AnyGenerator<Int>
    {
        var state = startIndex
        return anyGenerator(
            { state < self.endIndex ?
              state++ * self.table : nil }
        )
    }

    subscript(i: Index) -> Int {
        return i * self.table
    }
}

```

With this definition in place, the structure exhibits all of the functionality that it had as a sequence, as well as features unique to collections, such as subscript access and default methods defined on `CollectionType` such as `reverse()`. For example:


```
let d = timesTable4(9).reverse() as Array
// d is [108, 99, 90, 81, 72, 63, 54, 45, 36, 27, 18, 9]

print(timesTable4(6)[2])
// outputs "12" - the second element of the 6 times table
```

Memory Management

Swift, like Objective-C, uses *reference counting* as the main technique for keeping track of when dynamically allocated memory is no longer being used and can be released for other purposes.

For many years, Objective-C used (and can still use) manual reference counting, but this requires diligence on the part of the developer, and even though it can be mastered, it is a challenge for those new to the language to understand the nuances of when to use certain methods associated with reference counting, and when not to use them.

After a brief experiment with an Objective-C garbage collector for automatic memory management, Apple announced ARC in 2011. ARC—which stands for Automatic Reference Counting—uses the same approach a programmer would use, but it does so in a rigorous and deterministic manner.

How Reference Counting Works

The principle underpinning reference counting is quite simple. Every object (i.e., every instance of a class) has a built-in *reference count* property that is set to 1 when the object is instantiated.

Whenever a piece of code wants to express an interest in or ownership over the object (i.e., when a pointer is created that points to the object), it must increment the reference counter. When it has finished with the object and has no further interest in it (i.e., when its pointer to the object is no longer needed), it must decrement the reference counter. (In Objective-C, the way to express interest in an object is to call its `retain` method, and

the way to express no further interest is to call the object's release method.)

When the reference counter for an object is decremented to zero, it means there are no current references to the object and the object can be destroyed and the memory allocated to it can be released.

As long as an object's instantiation and release happens in the one function, the process of reference counting is very simple to master. The intricacies of reference counting don't really become apparent until the instantiation of an object is disconnected (code or execution-wise) from its release. At this point, it is relatively easy for a novice programmer to not release an object to which they created a reference (which results in a memory leak) or to release an object they don't actually own (which can result in a crash).

ARC manages the entire process of reference counting, automatically determining where to add retain and release calls, and thus relieves the programmer from the responsibility of doing so.

Retain Cycles and Strong References

One of the main problems with a reference counting approach to memory management is that of *retain cycles*. At their simplest, these occur when two objects contain *strong references* to each other. Consider the following code:

```
class A { }  
var a = A()
```

The variable `a` stores a strong reference to the newly created instance of class `A`.

Strong references are also created by default when objects store references to each other. This next example has been trimmed to the bare minimum, but this situation can occur in many places where complex interlinked data structures are used:

```

class A
{
    var otherObject: B?
}

class B
{
    var otherObject: A?
}

var a = A() // retain count for new instance of A set to 1
var b = B() // retain count for new instance of B set to 1
a.otherObject = b
// B instance retain count incremented to 2
b.otherObject = a
// A instance retain count incremented to 2

```

The code demonstrates two instances, referred to by `a` and `b`, that also refer to each other. After this code has executed, the retain count for each of the instances will be 2.

When the section of code that instantiates these two classes goes out of scope, `a` and `b`, being local variables, also go out of scope and are deleted. When `a` is deleted, the retain count for the instance it refers to is decremented (to 1), but because it is not zero, the instance itself is not deleted. Similarly, when `b` is deleted, the retain count for the instance it refers to is decremented (again, to 1), but, again, because it is not zero, that instance is also not deleted.

As a consequence, there exist two instances (one of class `A`, one of class `B`) that refer to each other and thus keep each other “alive.” Because the retain count for both instances never reaches zero, neither object can be deleted, and both take up memory.

This situation is referred to as a *memory leak*, and if the situation that created it occurs repeatedly during the execution of the program, the amount of memory allocated to the program will continue to grow, possibly causing performance issues or termination by the operating system if it limits the amount of memory an application can claim.

Although retain cycles are something an experienced coder might consider when manually managing memory, they are not something ARC can prevent without some assistance from the programmer. Essentially, if ARC is to manage memory automatically, it requires some additional information about the nature of references to other objects—they need to be classified as either *weak* or *unowned*.

Weak References

One way to prevent a retain cycle is to change one of the strong references to a weak reference. You do this by preceding the var declaration with the keyword `weak`, as in this example:

```
class A
{
    var otherObject: B?
}

class B
{
    weak var otherObject: A?
}
```

Weak references have two effects:

- There is no assertion that the referrer “owns” the instance it refers to, and it can deal with the fact that the instance might go away (in practical terms, this means that when a weak reference is established, the retain count is not incremented).
- When an instance referred to by a weak reference is deallocated, ARC sets the weak reference value to `nil` (thus, weak references must be declared as variables, not constants).

With a weak reference in place, consider in this next example how the retain counts change as the code executes:

```
var a = A() // retain count for new instance of A set to 1
var b = B() // retain count for new instance of B set to 1
a.otherObject = b
```

```
// B instance retain count incremented to 2
b.otherObject = a
// A instance retain count remains at 1
```

Because the reference that the instance of B holds to A is a weak reference, the retain count for the A instance remains at 1.

As before, when the local variables a and b go out of scope and are deleted, the retain count for the instances they refer to are both decremented, which means the retain count for the instance of A drops to zero, and the retain count for the instance of B drops to 1.

Because the instance of A now has a retain count of zero, it is no longer required in memory, and the process of its deinitialization and deallocation can begin. During deallocation, two things to happen:

- The weak reference to the instance of A that is stored in the instance of B is set to `nil`.
- Because the instance of A holds a strong reference to the instance of B (and A is being deallocated), the deallocation process sends B a release message, decrementing its retain count to zero.

At this point, the instance of A has been removed from memory, and what remains is an instance of B that has a retain count of zero. The process of its deinitialization and deallocation now begins.

When using weak references, consider the following:

- Use a weak reference if your code and data model allows for a reference to have no value (i.e., be `nil`) at times during the execution of your program.
- Weak references must always be defined as optionals.
- Weak references must be declared as variables (not constants).

Unowned References

An unowned reference is similar to a weak reference in that there is no assertion that the referrer “owns” the instance to which it refers, and when an unowned reference is established, the instance’s retain count is not incremented.

The main difference between a weak reference and an unowned reference is that while a weak reference may at times validly be `nil` without causing errors, an unowned reference, once established, must always have a value.

An unowned reference is defined by preceding the `var` declaration with the keyword `unowned`, as in this example:

```
class B
{
    unowned var otherObject: A
}
```

When using unowned references, consider the following:

- Use an unowned reference if your code or data model expects that the reference, after creation, will always exist and be valid (at least until such time as the referrer goes out of scope or is deleted).
- Unowned references must always be defined as nonoptionals.
- If you try to access an instance that has been deallocated via an unowned reference, Swift will terminate with a runtime error.

Retain Cycles and Closures

Like classes, closures are actually reference types. If you assign a closure to a property of an instance and that closure captures the instance, either by reference to a property of the instance or by a method call on the instance, you will have created a retain cycle between the instance and the closure.

The solution to this is to use either a weak or an unowned reference in the closure to the instance or method that is being captured, but the syntax is not the same as for references, as described earlier. Instead, references are specified in a *capture list* as part of the definition of the closure.

A capture list is defined inside the closure definition either immediately prior to the parameter list or immediately prior to the `in` keyword if the closure has no parameters. The capture list is a series of one or more reference types (unowned or weak, followed by the property or method it refers to) and separated by commas, as follows:

```
{
    [referenceType propertyOrMethod [, ...] ]
    (parameters) -> returnType in
        statements
}
```

For example, a closure being stored in a property `aClosure`, and referencing `self`, would look like:

```
var aClosure: (parameters) -> returnType =
{
    [unowned self]
    (parameters) -> returnType in
        statements
}
```

The rules as to which type of reference you should use remain the same as they do for instance-to-instance references. Use a weak reference (defined as an optional) if the reference may validly become `nil`; use an unowned reference if the closure and the instance it captures refer to each other, and the reference will remain valid until both objects can be deallocated.

Generics

Swift's *generics* feature provides you with the ability to write generic code that can work with any type of data. Similar features exist in C++ (as *templates*) and C# (as *generics*). Parts of the Swift standard library are implemented as generics. For

example, the `Array` type and the `Dictionary` type are generic collections that can store any type of data.

In Swift, you can write generic code in a number of ways, including generic functions, generic types, and generic protocols.

Generic Functions

To see a classic example of where generics are useful, consider the Swift standard library function `swap`, which is defined as follows:

```
func swap<T>(inout a: T, inout b: T)
```

From this function definition, you can see `swap` takes two `inout` parameters, of some type `T`, but there is nothing to indicate what type `T` actually is. (The two parameters are declared as `inout` parameters because the function must swap two existing items, not copies of them.)

`swap` is a generic function that can swap a pair of `Ints`, `Doubles`, or even a pair of user-defined type instances. It can swap any two variables, as long as they are the same type. To get a better understanding of what `swap` is doing, take a look at this next example to consider how it would be implemented:

```
func swap<T>(inout a: T, inout b: T)
{
    let temp = a
    a = b
    b = temp
}
```

Nothing in the body of the function definition is type specific. As long as the constant `temp` is the same type as `a` (which it will be due to type inferencing), and `a` is the same type as `b` (which it must be according to the types specified in the parameter list of the function), this function can swap any two same-typed values.

You define generic functions by using the following syntax:


```
func someFunc<Placeholder, [Placeholder...]>(parameterList)
{
    // function body
}
```

The key parts of the definition that indicate this is a generic function are the angle brackets immediately following the function name; these contain one or more *type placeholders*. These placeholders, called *type parameters*, stand in for actual types throughout the body of the function.

In much the same way that “i” has become a *de facto* loop variable, “T” is commonly used as the name for a type parameter in generic functions, but you can use any valid identifier.

Generic Types

In the section “Arrays” on page 48, it was noted that the preferred way to refer to an array of a specific type is `[SomeType]`, but that the formal way is `Array<SomeType>`. The angle brackets reveal the `Array` type is actually implemented as a generic type, and you can create your own generic types in the same way, using classes, structures, or enumerations.

This next example is a generic struct-based implementation of a queue:

```
struct Queue<T>
{
    var entries = [T]()

    mutating func enqueue(item: T)
    {
        entries.append(item)
    }

    mutating func dequeue() -> T
    {
        return entries.removeAtIndex(0)
    }
}
```

The queued data is stored in an array, `entries`, of type `T`, and defines two methods: `enqueue` (to add an item to the end of

the queue) and dequeue (to pull an item from the beginning of the queue).

So defined, Queue is now a new, generic type, and you can create queues for integers, strings, and any other data type to which you have access. For example, you can create and use a queue for Int data as follows:

```
var q1 = Queue<Int>()
q1.enqueue(45)
q1.enqueue(39)
q1.enqueue(61)
q1.enqueue(98)
dump(q1)
// dumps the equivalent of [45, 39, 61, 98]
q1.dequeue()
// returns 45
dump(q1)
// dumps the equivalent of [39, 61, 98]
```

Constraining Types

In designing a generic function or type, you might want to place some limits on what types it can support. You can constrain types based on their class (or subclass) or by protocol conformance, and with an optional where qualifier. A basic constraint is specified in the angle brackets immediately following the type parameter you want to constrain, as follows:

```
<T where T:SomeClass>
<T where T:SomeProtocol>
```

For the simplest of cases, these can be abbreviated as:

```
<T: SomeClass>
<T: SomeProtocol>
```

In the first example, T can act only as a placeholder for instances of *SomeClass* (or its subclasses). In the second example, T can act only as a placeholder for types that conform to *SomeProtocol*.

You can specify multiple types with a protocol composition that can include as many protocols as necessary:

```
<T: protocol<SomeProtocol, SomeOtherProtocol>>
```

Revisiting the Queue example from earlier, a constraint can be added so that it will only work with signed integers as follows:

```
struct SignedQueue<T: SignedIntegerType>
{
    // existing definition
}
```

`SignedIntegerType` is a protocol built into Swift the signed integer types (`Int`, `Int8`, `Int16`, `Int32`, and `Int64`) conform to, but which the unsigned types (`UInt`, `UInt8`, etc.) do not.

With this constraint in place, you can no longer create a queue for `UInt` data:

```
var q = SignedQueue<UInt>()
// error - Type 'UInt' does not conform to
// protocol 'SignedIntegerType'
```

The example that follows is a generic function that can merge two sorted arrays into a third. Because this function compares entries from each array, it is necessary that comparison is a defined operation for the type of data stored in the arrays, which is accomplished by constraining the type `T` to the `Comparable` protocol. Types that conform to this protocol can be compared with the relational operators `<`, `<=`, `>=`, and `>`, and many of Swift's built-in types (such as `Double`, `Int`, and `String`) conform:

```
func merge<T:Comparable>(a:[T], _ b:[T]) -> [T]
{
    var output = [T]()
    var i = 0, j = 0

    let sizea = a.count
    let sizeb = b.count
    output.reserveCapacity(sizea + sizeb)

    while (i < sizea) && (j < sizeb) {
        if a[i] < b[j] { output.append(a[i++]) }
        else { output.append(b[j++]) }
    }

    while i < sizea { output.append(a[i++]) }
}
```

```

        while j < sizeb { output.append(b[j++]) }

        return output
    }

```

Here's a simple example of using the `merge()` function with arrays of strings:

```

let s = ["allan", "fred", "mike"]
let t = ["brenda", "geraldine", "ruth"]
let u = merge(s, t)
u
// returns ["allan", "brenda", "fred", "geraldine",
// "mike", "ruth"]

```

Since the function is generic, it can also be used to merge two arrays of integers:

```

let v = [3, 9, 17, 21]
let w = [1, 2, 12, 36]
let x = merge(v, w)
x
// returns [1, 2, 3, 9, 12, 17, 21, 36]

```

The “where” type constraint clause

More complex constraints can be specified with a `where` clause, which can include multiple comma-separated conditions that must all be matched so the constraint is satisfied.

The sorts of constraints that can follow the `where` clause can include:

- that a type conforms to a protocol or class, e.g., `<T where T:CollectionType>`
- that a type matches the type of another type's property, e.g., `<U where U.Generator.Element == T>`
- that a type conforms to several protocols through a protocol composition, e.g., `<T where T:protocol<aProtocol, anotherProtocol>>`

Following is a more complex example that partially implements an array whose contents are always stored in sorted order. This

implementation includes an initializer that allows instances to be constructed from an existing type. That type (identified by the placeholder `U`) is constrained with a `where` clause such that it must conform to the `SequenceType` protocol (which includes the `Array` type) and furthermore that its elements are of the same type as the sorted array (identified by the placeholder `T`):

```
struct SortedArray<T: Comparable>
{
    private var elements: [T] = []

    init<U where U:SequenceType,
        U.Generator.Element == T>(_ sequence: U) {
        elements = sequence.sort()
    }

    // binary search elements for specific value
    private func bsearch(value: T) -> Int
    {
        guard elements.count > 0 else { return 0 }
        var middle = 0, lower = 0
        var upper = elements.count - 1
        while lower < upper {
            middle = (lower + upper) >> 1
            if elements[middle] < value
                { lower = middle + 1 }
            else { upper = middle }
        }
        if (elements[lower] < value) { return lower+1 }
        return lower
    }

    mutating func insert(value: T)
    {
        elements.insert(value, atIndex: bsearch(value))
    }
}
```

With this definition in place, it is possible to create an instance from an unsorted array, and start inserting new elements in sorted order:

```
var sa = SortedArray([15, 3, 9, 7])
sa.insert(13)
sa.elements
// returns [3, 7, 9, 13, 15]
```

Since the structure is generic, it can also be used with non-integer types, such as strings:

```
var sb = SortedArray(["Wilma", "Albert"])
sb.insert("Geraldine")
sb.elements
// returns ["Albert", "Geraldine", "Wilma"]
```

Generic Protocols

With Swift, you can also write generic protocols, although the way this works is a little different than the way generic types are expressed in function and type definitions. Instead of a `<T>` placeholder, unknown types in protocols are identified by using the `typealias` keyword, which was introduced earlier in the section “Types” on page 18. When type aliases are used in this way in protocol definitions, they are known as *associated types*.

To use an associated type, the protocol definition takes this form:

```
protocol SomeProtocol
{
    typealias SomeName
    // remainder of protocol definition with the generic
    // type references expressed as SomeName
}
```

The actual type the type alias `SomeName` refers to is defined when the protocol is adopted, in the same way the other required parts of the protocol must be defined to provide conformance:

```
class SomeClass : SomeProtocol
{
    typealias SomeName = SomeActualType
    // rest of class definition
}
```

Here’s an example of a generic protocol, `Queueable`, that demonstrates the use of an associated type:

```
protocol Queueable
{
    typealias NativeType
    mutating func enqueue(item: NativeType)
```

```
    mutating func dequeue() -> NativeType
}
```

A class or structure that adopts this protocol must implement the enqueue and dequeue methods (thus, behaving like a queue), but the type of data these methods use is defined in the adopting class at the same place the methods themselves are defined—in the protocol definition, it's referred to as `NativeType`.

Following is an example of a structure that stores a list of strings and adopts the `Queueable` protocol so the list can also be treated as a queue. You can see that the `NativeType` from the protocol is defined as a `String` type for this particular adopter:

```
struct StringList: Queueable
{
    var list = [String]()

    typealias NativeType = String

    mutating func enqueue(item: NativeType)
    {
        list.append(item)
    }

    mutating func dequeue() -> NativeType
    {
        return list.removeAtIndex(0)
    }
}
```

You could use the structure as follows:

```
var sl = StringList()
sl.enqueue("Joshua")
sl.enqueue("Nadia")
sl.enqueue("Paul")
sl.dequeue()
// returns "Joshua"
dump(sl)
// dumps the equivalent of ["Nadia", "Paul"]
```

Operator Overloading

Operator overloading is the ability to define custom behavior for standard operators (+, /, =, etc.) when they are used with

custom types. Overloading is a controversial feature because it can lead to ambiguous code.

For example, all programmers understand that the `+` operator is traditionally associated with addition when both operands are numeric. A slightly smaller group generally associates `+` with concatenation when both operands are strings, so `+` is already overloaded in languages that support concatenation with it.

A language that supports programmer-defined operator overloading takes this further, allowing the programmer to add new custom behavior to any of the standard operators when they are applied to custom types. For example, you could define a struct type to represent vectors. Adding vectors is a natural operation for people who think in terms of motion, and overloading `+` to add two vector types allows vector addition to be expressed naturally in code.

You overload binary infix operators in Swift by using the following syntax:

```
func + ([inout] left: SomeType, right: SomeType)
    -> SomeType
{
    // code that returns a value of SomeType
}
```

Let's look closer at this syntax:

- The parameter names are shown as `left` and `right`, but you can use any other parameter names. The first parameter is the one that appears on the lefthand side of the operator, and the second parameter is the one that appears on the righthand side.
- This example overloads `+`, but you can overload any existing binary operator (including compound assignment operators and comparison operators) except for assignment (`=`).
- When overloading compound assignment operators (such as `+=`), the first (left) parameter must be prefixed

with `inout` because the body of the function will directly modify the left parameter.

- The two input types are both shown as *SomeType*, but these do not have to be of the same type.
- The return value does not have to be the same type as either of the operands.

In the section “Structures” on page 139, a simple structure `Rect` was used to represent rectangular shapes, as shown in this example:

```
struct Rect
{
    var x = 0.0, y = 0.0, width = 0.0, height = 0.0

    func area() -> Double
    {
        return (width * height)
    }
}
```

Here is an overloaded version of the `+` operator that returns a new `Rect` that represents the smallest rectangle that would contain the two operand rectangles (assuming the origin is at the upper left):

```
func + (left: Rect, right: Rect) -> Rect
{
    return Rect (
        x: min(left.x, right.x),
        y: min(left.y, right.y),
        width: max(left.width, right.width),
        height: max(left.height, right.height)
    )
}
```

This new operator could be used as follows:

```
var a = Rect (x:5, y:5, width:5, height:5)
var b = Rect (x:6, y:6, width:10, height:10)
var c = a + b
// c is now a Rect where
// (x=5.0, y=5.0, width=10.0, height=10.0)
```

The following example overloads the < operator so that two Rects can be compared in terms of area:

```
func < (left: Rect, right: Rect) -> Bool
{
    return left.area() < right.area()
}
```

You could use this as follows:

```
var e = Rect(x:0, y:0, width:4, height:5)
var f = Rect(x:5, y:5, width:5, height:5)
e<f
// returns true
```

Overloading Unary Operators

The incrementing and decrementing unary operators (++ , --) are overloaded by preceding the function definition with either the prefix or postfix keyword. The general pattern is as follows:

```
prefix func ++ (someName: someType) -> someType
{
    // code that returns a value of SomeType
}
```

Here's an alternative:

```
postfix func -- (someName: someType) -> someType
{
    // code that returns a value of SomeType
}
```

The parameter names are shown as *someName*, but you can use any parameter names.

Note that the return value does not have to be the same type as the operand.

The following example defines a ++ postfix operator for the Rect type that adds 1.0 to the x- and y-coordinates but leaves the width and height unmodified:

```
postfix func ++ (inout r: Rect) -> Rect
{
    let temp = r;
```

```

    r.x += 1.0
    r.y += 1.0
    return temp
}

```

Note that this function copies the operand so that the value returned is the original, unmodified value (thus mimicking the expected behavior of a postfix ++ operator). In use, this would behave as follows:

```

var d = Rect(x:5, y:5, width:5, height:5)
d++
// returns a Rect where
// (x=5.0, y=5.0, width=5.0, height=5.0)
// but d is a Rect where
// (x=6.0, y=6.0, width=5.0, height=5.0)

```

Custom Operators

As well as overloading the built-in operators, you can create custom operators that can begin with any of the ASCII characters +, -, *, /, =, !, %, <, >, &, |, ^, and ~, as well as a range of Unicode character blocks, including the math, symbol, arrow, dingbat, line drawing, and box drawing sets. The second and subsequent characters can be any of those listed as well as any of the Unicode combining characters (which are characters that modify other characters, such as diacritical marks and accents).

Unusually, custom operators in Swift need to be declared before they are defined, using this syntax:

```
[prefix|postfix|infix] operator symbols {}
```

For example, you could declare a prefix operator that used the square root symbol ($\sqrt{\quad}$) to calculate square roots, as follows:

```
prefix operator √ {}
```

With the declaration in place, you could then define the function:

```

prefix func √ (operand: Double) -> Double
{
    return sqrt(operand)
}

```

So defined, you can then use the operator as follows:

```
print (√25)
// outputs 5.0
```

Custom Operator Precedence

When you define custom infix operators, you can also specify optional precedence and associativity values. These values are specified when the custom operator is declared (not when it is subsequently defined) as follows:

```
operator symbols { associativity someValue
                   precedence someValue }
```

Precedence is specified as a numeric value, and defaults to 100 if not provided.

Associativity is specified as `left`, `right`, or `none`, and defaults to `none` if not provided.

See the subsection “[Operator Precedence](#)” on page 37 for more information.

Ranges, Intervals, and Strides

An earlier section of this book introduced the closed range operator (`x...y`) and the half-open range operator (`x..). These operators represent two of the more commonly used types of ranges (they’re used frequently in iteration), but Swift supports two other range types: intervals and strides.`

Let’s look at all three of these types a little more closely.

Ranges

A range is a collection of consecutive discrete values. The end of the range must be reachable from the start by a process of repeated incrementation (so the start can’t be a value that is later in the series than the end).

Typically, you will use ranges with integer types, but you can use them with any type that conforms to the `ForwardIndexType` protocol.

Ranges are types (so, for example, a variable can be of type `Range`) and include the properties `startIndex` and `endIndex`, as demonstrated in this example:

```
var r = 1...5
r
// returns "1..<6"
r.startIndex
// returns 1
r.endIndex
// returns 6

for x in r {
    print(x, terminator: ", ")
}
// prints:
// 1, 2, 3, 4, 5,
```

Observe that even though the closed range `1...5` was assigned to `r`, Swift converted this internally to the half-open range `1..<6`. This is because it always represents ranges internally in half-open format.

The `endIndex` represents the end of a range but is not a value in the range, which is why the `for` loop only outputs five values.

Intervals

Like a range, an interval consists of a start value and an end value, and the start must be less than the end, but intervals are not associated with indexing or the concept of advancing progressively from the start to the end by incrementation. Instead, they are associated with the idea of *containment*—for example, checking whether a value is contained within the interval.

You can use intervals with any type that conforms to the `Comparable` protocol, which includes the integer and floating-point types.

Intervals are types, and a variable or constant can be of type `HalfOpenInterval` or `ClosedInterval`.

Intervals include as properties `start` and `end` (both of type `Bound`, which is a type alias for `Comparable`), and `isEmpty`, which has the value `true` if the interval is empty.

Intervals include the following instance methods:

`someInterval.contains(someValue)`

Returns `true` if `someValue` lies within the `someInterval`.

`someInterval.clamp(someOtherInterval)`

Returns a new interval that is essentially the intersection of `someInterval` and `someOtherInterval`. The start of the returned interval will never be less than the start of `someInterval`, and the end of the returned interval will never be greater than the end of `someInterval`.

`someInterval.overlaps(someOtherInterval)`

Returns `true` if the intersection of `someInterval` and `someOtherInterval` is nonempty.

Here are some examples of creating and using intervals:

```
var i = 1.0...2.0
i.start // returns 1.0
i.end   // returns 2.0
i.isEmpty // returns false

// contains
i.contains(3.4) // returns false
i.contains(1.6) // returns true

// clamping
let c = i.clamp(1.5...2.5) // 1.5...2.0
let d = i.clamp(0.1...0.2) // 1.0...1.0
let e = i.clamp(3.0...4.0) // 2.0...2.0

// overlaps
i.overlaps(1.2...1.4) // true
i.overlaps(2.0...3.0) // true
i.overlaps(4.0...5.0) // false
```

Like ranges, you can define intervals as either half-open or closed. Since the same operators are used to create ranges and intervals, it is important to understand the default behavior:

- When a type used with the half-open or closed range operator conforms to only the `Comparable` protocol (this includes the floating-point types) the operator will always return an interval.
- When a type used with the half-open or closed range operator conforms to both the `Comparable` and `ForwardIndexType` protocols (this includes the integer types), the operator will return an interval when it is used in a pattern-matching context (e.g., a `switch` case), but it will return a range in any other context.

If you need to create an integer interval for use outside of pattern matching, you can specify the type as follows:

```
let j = HalfOpenInterval(2..<3)
let k: ClosedInterval = 4...14
```

Strides

Like ranges and intervals, a stride consists of a start and an end, but it also includes a distance to step as the sequence progresses. You can use strides with any type that conforms to the `Strideable` protocol, which includes floating-point and integer types.

Strides are initialized with the `.stride()` method, which takes one of two forms, as follows:

```
type.stride(to:endValue, by:increment)
type.stride(through:endValue, by:increment)
```

The first form is said to be *exclusive*—the last value of the stride will always be less than the value of the `to` parameter.

The second form is said to be *inclusive*—the last value of the stride *may* equal the `to` value, but will never exceed it.

Unlike ranges and intervals, the `to/through` and `by` values are not accessible as properties of the stride.

The following example creates an exclusive stride from 2 to 8. The type of `s` in this example is `StrideTo<Int>`.

```
var s = 2.stride(to:8, by:2)
for x in s {
    print (x)
}
// prints:
// 2
// 4
// 6
```

In this next example, a stride is used directly, but this time it is initialized by using the `through` parameter so the end value (2.8) is included in the sequence. The type of the stride in this example is `StrideThrough<Double>`:

```
for x in 2.2.stride(through:2.8, by:0.3) {
    print (x)
}
// prints:
// 2.2
// 2.5
// 2.8
```

Note that since it is impossible to exactly represent all floating-point values, a stride initialized with a floating-point to parameter may appear to return the end value as the last value in a sequence. This is not an issue with strides, but a reminder that floating-point values should not be relied on for certain values.

Global Functions

All of Swift's global functions are listed here. While this may seem like a small list, remember that a substantial amount of Swift's functionality is defined using protocols and extensions.

`abs(x)`

Returns the absolute value of `x`, which must conform to the `AbsoluteValuable` protocol.

- `alignof(type)`
Returns the minimum memory alignment of the type *type*.
- `alignofValue(value)`
Returns the minimum memory alignment of the instance *value*.
- `anyGenerator(body)`
Returns an instance of a `GeneratorType`. *body* is a closure that typically captures a value in scope, and returns the next value in a sequence every time it is called, or `nil` if the sequence has ended.
- `assert(condition: Bool [, message: String])`
C-style assert with optional message. Tests condition. If it evaluates to false, the program is terminated and message (if provided) is displayed as an error. Ignored in Release builds.
- `assertionFailure([message: String])`
Terminates the program and displays message (if provided) as an error. Ignored in Release builds.
- `debugPrint(items... [, separator: String] [, terminator: String])`
Writes a debug representation of the list of comma-separated items to the output stream. *separator* defaults to a space, and is output between each item. *terminator* defaults to a new line, and is output at the end of the sequence.
- `dump(x)`
Dumps the contents of *x* using its mirror to standard output. Useful for debugging and exploring the contents of instances.
- `fatalError([message: String])`
Unconditionally terminates the program and displays message (if provided) as an error.
- `getVAList(args: CVarArgType)`
Returns a `CVAListPointer` built from *args* that's backed by autoreleased storage. Use `withVAList()` in preference to this where possible.

`isUniquelyReferenced(someObject)`

Returns a `Bool` value of `true` if there is a single strong reference to *someObject*, which must be derived from the `NonObjectiveCBase` class. *someObject* is not modified, despite the `inout` annotation. Useful for implementing copy-on-write optimization of value types.

`isUniquelyReferencedNonObjC(&someObject)`

Returns a `Bool` value of `true` if there is a single strong reference to *someObject*. If *someObject* is an instance of an Objective-C class, always returns `false`. *someObject* is not modified, despite the `inout` annotation. Useful for implementing copy-on-write optimization of value types.

`max(list)`

Returns the greatest argument in *list*, which consists of comma-separated arguments of the same type. That type must conform to the `Comparable` protocol, which includes `Int` and `Double` (and their variants), and `String`.

`min(list)`

Returns the lesser argument in *list*, which consists of comma-separated arguments of the same type. That type must conform to the `Comparable` protocol, which includes `Int` and `Double` (and their variants), and `String`.

`numericCast(x)`

Used to convert one integer type to another contextually-deduced integer type, trapping on overflow. For example, `func f(x: Int)` cannot be called with any other integer parameter type (such as `UInt`), but if called as `f(numericCast(UInt))`, then the `UInt` parameter will be cast to an `Int`.

`precondition(condition: Bool[, message: String])`

The *condition* must evaluate to `true` for execution to continue; otherwise execution terminates and the optional *message* is displayed. Unlike `assert()`, this will terminate Release builds as well as Debug builds.

`preconditionFailure([message: String])`

Terminates the program and displays `message` (if provided) as an error. Unlike `assertionFailure()`, this will terminate Release builds as well as Debug builds.

`print(items... [, separator: String] [, terminator: String] [, toStream: &target])`

Writes a textual representation of the list of comma-separated items to the output stream. `separator` defaults to a space, and is output between each item. `terminator` defaults to a new line, and is output at the end of the sequence. `toStream` defaults to `stdout`.

`readLine([stripNewLine: Bool])`

Returns standard input as an optional `String`, through to the end of the current line or until EOF is encountered, or `nil` if EOF has been reached. If the optional `stripNewLine` parameter is `true` (the default), newline characters and character combinations are removed from the result.

`sizeof(type)`

Returns the contiguous storage space for an instance of `type`, but does not include dynamically allocated storage associated with that type. The size returned includes space allocated for stored properties if the type is a struct, but does not count stored properties if the type is a class.

`sizeofValue(x)`

Returns the contiguous storage space for `x`, but does not include dynamically allocated storage associated with `x`. The size returned includes space allocated for stored properties if `x` is a struct, but does not count stored properties if `x` is a class.

`strideof(type)`

Returns the least possible interval between instances of `type` in memory.

`strideofValue(value)`

Returns the least possible interval between separate instances of `value` in memory.

`swap(&x, &y)`

Swaps `x` and `y`, which must be the same type.

`transcode(inputEncoding: InputEncoding.Type, outputEncoding: outputEncoding.Type, input: Input, output: Output, stopOnError: Bool)`

Transcodes `input` (in the given *InputEncoding*) to output (in the given *OutputEncoding*). If `stopOnError` is true, and an encoding error is detected, the function will stop and return `false`. If `stopOnError` is false, the Unicode REPLACEMENT CHARACTER U+FFFD is inserted into the output for each encoding error detected.

`unsafeAddressOf(x)`

Returns the address of the storage allocated for `x`.

`unsafeBitCast(x, t)`

Returns the bits of `x`, recast as type `t`. Considered dangerous, as it breaks Swift's type checking.

`unsafeDowncast(x)`

Returns `x` recast as type `t` where `t` is the type of the receiver. If `x` is not of type `t`, will generate a runtime error. Trades type safety for performance.

`unsafeUnwrap(x)`

Returns an unwrapped copy of `x`. If `x` is `nil`, will generate a runtime error. Trades safety for performance.

`withExtendedLifetime(x, f)`

Evaluates the function `f()` and returns its result, ensuring `x` is not destroyed before `f()` returns. `f()` can either take no parameters, or can take `x` as a single parameter. `f()` has a `throws` annotation—this function will rethrow any error thrown by it.

`withUnsafeMutablePointer(&x, f)`

Evaluates the function `f()` with an `UnsafeMutablePointer` to `x`, and returns the result. `f()` has a `throws` annotation—this function will rethrow any error thrown by it.

`withUnsafeMutablePointers(&x, &y [, &z], f)`

Performs the same function as `withUnsafeMutablePointer()` but calls `f()` with pointers to `x`, `y`, and (if provided), `z`.

`withUnsafePointer(&x, f)`

Evaluates the function `f()` with an `UnsafePointer` to `x`, and returns the result. `f()` has a `throws` annotation—this function will rethrow any error thrown by it.

`withUnsafePointers(&x, &y [,&z], f)`

Performs the same function as `withUnsafePointer()` but calls `f()` with pointers to `x`, `y`, and (if provided), `z`.

`withVaList(args: [CVarArgType], f)`

Invokes the function `f()` with a C-language `va_list` argument derived from `args`, and returns the result of `f()`.

`zip(SequenceType1, SequenceType2)`

Returns a sequence of tuples, with each tuple consisting of one item from `SequenceType1`, and the matching item from `SequenceType2`. If one sequence is longer than the other, any unmatched items are discarded.

Changes From Swift 1.0

Swift 2.0 includes a number of syntax changes from earlier versions that will prevent programs you have already written from compiling. Xcode 7 offers an option to convert existing code to the latest Swift syntax (click `Edit` → `Convert` → `To Latest Swift Syntax...` to use this feature).

Here is a list of the major changes (note that some were actually added in Swift 1.1 or Swift 1.2):

- The `println()` function has been removed.
- The `print()` function has been extended to support printing a list of comma-separated items, with an optional separator and terminator strings.
- The overflow division (`&/`) and overflow remainder (`&%`) operators have been removed.
- The `as` keyword, which forcibly downcasts an instance reference to a specific subclass type, or an instance reference to a specific protocol type, has been replaced with `as!`.

- In Swift 1.0, a `String` was an `Array of Character` (much like an array in C). In Swift 2.0, this is no longer the case, but the characters that comprise the string are now accessible via the `.characters` property.
- The `do-while` loop construct has been replaced with `repeat-while`.
- The `do` keyword now opens a new scope.
- The `defer` statement has been added, which allows you to specify a block of code that is guaranteed to be executed as the current scope exits, regardless of how it exits.
- The `guard-else` statement has been added, for testing that a condition is met before continuing with the rest of the current scope.
- A new error-handling model has been introduced. Functions annotated with `throws` can throw errors. Such functions are called with `do-try` and errors are caught with `catch`.
- The `if`, `guard`, and `while` statements can now take a case pattern as would appear in a `switch` statement as the condition to be tested, including optional `let` variable binding and an optional `where` clause.
- Optional binding (e.g., `if-let` and `guard-let`) now supports a `where` clause, as well as compound `let` and `var` assignments, separated by commas.
- The index value in `for-in` loops can be filtered with a `where` clause as part of the loop construct.
- Enumerations can now be self-referential using the `indirect` keyword.
- Failable initializers can now be defined, allowing an initialization function to return `nil` if it cannot properly initialize a structure, class, or enumeration.
- A new collection type `Set` has been added, which includes support for set algebra operations.
- Option sets can be now used, providing set-type behavior for bit-field Boolean values.
- Extensions can now be defined for protocols and provide default implementations of methods and properties.

Symbols

- ! exclamation mark
 - as logical operator, 35
 - custom operators and, 205
 - unwrapping optionals and, 82
- != comparison operator, 34
 - strings, 41
- !== comparison operator, 34
- #include statements, 16
- % percent sign
 - custom operators and, 205
 - remainder operator, 31
- %= assignment operator, 33
- & operator (bitwise AND), 32
 - custom operators and, 205
- && logical operator, 35
- &* overflow multiplication operator, 35
- &+ overflow addition operator, 35
- &- overflow subtraction operator, 35
- &= assignment operator, 33 (double quotes), 20
- * operator, 31
 - custom operators and, 205
- *= assignment operator, 33
- + operator, 31
 - custom operators and, 205
 - string concatenation, 16
- ++ operator, 32
- += assignment operator, 33
- operator, 31
 - custom operators and, 205
- operator (decrement), 32
- = assignment operator, 33
- > closure expression, 75
- ..< half-open range operator, 37
- / operator, 31
 - custom operators and, 205
- /* */ (multiline comment delimiter), 15
- // (comment marker), 15
- /= assignment operator, 33
- ;(semicolons), 15
- < comparison operator, 34
 - custom operators and, 205
 - strings, 41
- << operator (bitwise left-shift), 32
- <<= assignment operator, 33
- <= comparison operator, 34
 - strings, 41
- = assignment operator, 33
 - custom operators and, 205

- == comparison operator, 34
 - strings, 41
- === comparison operator, 34
- > comparison operator, 34
 - custom operators and, 205
 - strings, 41
- >= comparison operator, 34
 - strings, 41
- >> operator (bitwise right-shift), 32
- >>= assignment operator, 33
- ? (question mark), 82
- [] syntax
 - for arrays, 49
 - for dictionaries, 57
- \" (double quote) escape sequence, 42
- ' (single quote) escape sequence, 42
- \n (line feed) escape sequence, 42
- \r (carriage return) escape sequence, 42
- \t (tab) escape sequence, 42
- \u{n} arbitrary Unicode scalar escape sequence, 42
- \\ (backslash) escape sequence, 42
- ^ operator
 - bitwise XOR, 32
 - custom operators and, 205
- ^= assignment operator, 33
- _ underscore character, 99
- ` (back ticks), 24
- | operator
 - bitwise OR, 32
 - custom operators and, 205
- |= assignment operator, 33
- || logical operator, 35
- ~ operator
 - bitwise NOT, 32
 - custom operators and, 205
- ~= comparison operator, 34
- ... range operator, 36
- √ (square root), 205

A

- abs() global function, 210
- AbsoluteValuable protocol, 176
- access control, 151-155
 - default, 153-155
 - of class members, 125
 - specifying level of, 152
- addWithOverflow() function, 35
- alignof() global function, 211
- alignofValue() global function, 211
- Any
 - protocol, 176
 - type alias, 158
- AnyClass protocol, 176
- AnyGenerator type, 180
- anyGenerator() global function, 180, 211
- AnyObject
 - protocol, 176
 - type alias, 158
- append() method
 - in arrays, 50
 - in strings, 47
- appendContentsOf() method
 - in strings, 47
- Apple Worldwide Developers Conference, 1
- ARC, 2, 187
- arguments, automatic names, 77
- arithmetic operators, 31
- arrays, 48-57
 - append method, 50
 - appending two, 50
 - assigning value to element, 50
 - assigning value to range of elements, 50
 - capacity property, 50
 - capacity, reserving, 51
 - contains() method, 52
 - count property, 50
 - dropFirst() method, 52
 - dropLast() method, 53

- elements, accessing, 49
 - elementsEqual() method, 53
 - filter() method, 53
 - flatMap() method, 53
 - forEach() method, 53
 - indexOf() method, 54
 - inherited functionality, 52
 - inserting values into, 50
 - isEmpty property, 50
 - iterating over, 51
 - joinWithSeparator() method, 54
 - map() method, 54
 - maxElement() method, 49
 - minElement() method, 50
 - modifying, 50
 - mutable, 50
 - prefix() method, 54
 - properties of, 50
 - reduce() method, 54
 - remove and return last element of, 51
 - remove and return single elements from, 51
 - removing all elements from, 51
 - reverse() method, 55
 - slices of, 56
 - sort() method, 55
 - sorting, 51
 - split() method, 55
 - startsWith() method, 55
 - suffix() method, 56
 - as operator
 - checking for protocol conformance, 172
 - type casting, 36, 158
 - as! operator
 - downcasting with, 161
 - type casting, 36
 - as? operator
 - checking for protocol conformance, 172
 - downcasting with, 161
 - type casting, 36, 158
 - assert() global function, 211
 - assertionFailure() global function, 211
 - assignment operators, 33-34
 - associated types, 200
- ## B
- base classes, 125
 - BidirectionalIndexType protocol, 176
 - binary operators, 30
 - bitwise operators, 32
 - BitwiseOperationsType protocol, 176
 - blocks, 75
 - Bool values, 35
 - BooleanType protocol, 176
 - break statements, 101
- ## C
- capture list, 193
 - case
 - in for-in statement, 91
 - in guard-else statement, 95
 - in if statement, 94
 - in switch statement, 96
 - in while statement, 92
 - matching ranges in, 98
 - using tuples in, 98
 - catch statement, 103
 - characters, 39
 - literals, 20
 - clamp() method (intervals), 208
 - Clang, 2
 - classes, 107-139
 - computed properties, 112-114
 - computed type properties, 117
 - constant properties, 23, 119
 - defining, 108

- deinitialization, 138
 - inheritance, 125
 - initialization, 130-139
 - instances, 108
 - member protection, 125
 - methods, 120-125
 - nested types and, 21
 - overrides, preventing, 129
 - overriding superclass entities, 126-129
 - properties of, 110
 - properties, instance vs. type, 115
 - property observers, 114
 - self property, 121
 - static properties, 116
 - stored properties, 110
 - subclassing, preventing, 129
 - closures, 75-81
 - automatic argument names, 77
 - capturing values by reference, 80
 - capturing values with, 79-80
 - retain cycles and, 192
 - trailing, 78
 - Cocoa Framework, 9
 - collections, 185
 - CollectionType protocol, 176, 185
 - command line access, 6-8
 - comments, 15
 - Comparable protocol, 176, 197
 - comparison operators, 34, 197
 - computed properties, 112-114
 - extensions, 156
 - computed type properties, 117
 - computed variables, 24
 - conditionals, 93
 - guard-else statement, 94
 - if-else statement, 93
 - switch statements, 96-102
 - constant properties, 23
 - in classes, 119
 - constants, 23
 - default access level of, 153
 - tuples, 28
 - constraining types (where clause), 196
 - contains() method
 - in arrays, 52
 - in intervals, 208
 - in sets, 63
 - continue statements, 101
 - convenience initializers, 130, 134
 - overriding, 138
 - count property
 - in arrays, 50
 - in dictionaries, 58
 - in sets, 63
 - in strings, 40, 45
 - curly braces, 93
 - custom operators, 205
 - precedence, 206
 - CustomDebugStringConvertible protocol, 176
 - CustomLeafReflectable protocol, 177
 - CustomPlaygroundQuickLookable protocol, 177
 - CustomReflectable protocol, 177
 - CustomStringConvertible protocol, 177
- ## D
- data types, 18-22
 - debugPrint() global function, 211
 - default clause (switch statements), 96
 - defer statement, 102
 - deferred execution, 102
 - deinitialization, 138
 - designated initializers, 130, 132-134
 - overriding, 138
 - developer resources, 5
 - Developer Tools Access prompt, 7

- dictionaries, 57-62
 - accessing elements of, 58
 - count property, 58
 - dropFirst() method, 61
 - dropLast() method, 61
 - endIndex property, 61
 - forEach() method, 61
 - indexOfKey() method, 61
 - inherited functionality, 60
 - isEmpty property, 59
 - iterating over, 60
 - keys property, 59
 - modifying, 59
 - mutable, 59
 - pop and return first element from, 59
 - properties of, 58
 - remove all elements from, 59
 - remove specified elements from, 60
 - removeAtIndex method, 61
 - setting values for specified elements, 59
 - startIndex property, 62
 - updating values for specified elements, 59
 - values property, 59
- didSet keyword
 - property observers, 114
 - variable observers, 26
- divideWithOverflow() function, 35
- do statement, 101, 103
- do-try-catch sequence, 103
- downcasting, 160-162
- dropFirst() method
 - in arrays, 52
 - in dictionaries, 61
- dropLast() method
 - in arrays, 53
 - in dictionaries, 61
- dump() global function, 211

E

- elementsEqual() (arrays), 53
- else clause, 94
- endIndex property
 - in dictionaries, 61
 - in sets, 66
 - in strings, 46
- enumerations, 144-150
 - associated values, 147-148
 - default access level of, 153
 - methods in, 148
 - raw member values, 145
 - recursive, 150
 - type methods in, 149
 - using switch statements with, 100
- Equatable protocol, 177
- ErrorType protocol, 104, 177
- escaped characters in strings, 42
- extended grapheme cluster, 43
- extensions, 155-158
 - adopting protocols with, 168
 - computed properties, 156
 - default access level of, 153
 - initializers, 156
 - methods, 156
 - subscripts, 157
- external parameter names
 - in functions, 71
 - in init() methods, 133
 - in methods, 121-121

F

- failable initializers, 135, 168
- fatalError() global function, 211
- filter() (arrays), 53
- final keyword, 129
- flatMap() (arrays), 53
- floating point literals, 19
- FloatingPointType protocol, 177
- for-condition-increment loops, 88
- for-in loops, 89

- case pattern, 91
 - index value filtering, 90
 - iterating over arrays with, 51
 - iterating over dictionaries
 - with, 60
 - iterating over sets with, 64
 - where clause, 90, 91
 - forEach() method
 - in arrays, 53
 - in dictionaries, 61
 - in sets, 66
 - ForwardIndexType protocol, 177
 - Foundation, 47
 - Foundation framework, 166
 - functions, 68-75
 - computed variables, 24
 - default access level of, 154
 - default parameter values, 72
 - external parameter names, 71
 - generic, 194
 - global, 210-215
 - local parameter names, 71
 - parameter types, 68
 - returning multiple values, 70
 - returning optional values, 69
 - returning tuples, 70
 - types, 74
 - variadic parameters for, 73
- G**
- generators, 180
 - GeneratorType
 - protocol, 180
 - GeneratorType protocol, 178
 - generics, 193-201
 - constraining types, 196-200
 - default access level of, 154
 - functions, 194
 - protocols, 200-201
 - types, 195
 - getter functions
 - computed properties, 112
 - computed variables, 24
 - default access level of, 154
 - getVAList() global function, 211
 - guard statements, 84
 - guard-else statement, 94
 - case pattern, 95
 - optional binding and, 84
 - where clause, 95
- H**
- half-open range operator, 37
 - Hashable protocol, 178
- I**
- if-else statement, 93
 - case pattern, 94
 - optional binding and, 84
 - where clause, 94
 - import statements, 16
 - in-out parameters, 69
 - Indexable protocol, 178
 - indexOf() method
 - in arrays, 54
 - in sets, 66
 - inheritance, 106, 125
 - initializers and, 137-138
 - protocols and, 169
 - inherited functionality
 - arrays, 52
 - dictionaries, 60
 - sets, 66
 - strings, 46
 - initializer delegation, 143
 - initializers, 130-139
 - convenience, 134
 - default access level of, 154
 - designated, 132-134
 - extensions, 156
 - failable, 135
 - for structures, 142
 - in protocols, 168
 - inheritance and, 137-138
 - overriding, 138

- required, 138
- instances, 108
- instantiation, 109
- Int types, 18
- integer types, 18
 - overflow operators, 35
- IntegerArithmeticType protocol, 178
- IntegerType protocol, 178
- internal access control level, 151
- intervals, 207
- IntervalType protocol, 178
- iOS 7, 5
- is operator
 - checking for protocol conformance, 172
 - checking types with, 161
 - type casting, 36, 158
- isEmpty property
 - in arrays, 50
 - in dictionaries, 59
 - in sets, 63
- isUniquelyReferenced() global function, 212
- isUniquelyReferencedNonObjC() global function, 212

J

- joinWithSeparator() method (arrays), 54

K

- keys property (dictionaries), 59

L

- lambdas, 75
- Lattner, Chris, 1
- lazy evaluation, 111
- lazy initialization of stored properties, 111
- LazyCollectionType protocol, 178
- LazySequenceType protocol, 178

- let keyword
 - declaring constants, 23
- let statement
 - optional binding with, 84
 - value binding, 161
 - value binding with, 99
- literals
 - array, 48, 62
 - character, 20
 - dictionary, 57
 - floating point, 19
 - numeric, 19
 - string, 20
- LLDB, 2
- LLVM, 2
- local parameter names, 121-121
 - in functions, 71
 - in methods, 121-121
- logical operators, 35
- loops, 88-93
 - early termination of, 93
 - for-condition-increment, 88
 - for-in, 89
 - repeat-while, 92
 - while, 91

M

- map() method (arrays), 54
- max() global function, 212
- memberwise initializer, 142
- memory leaks, 189
- memory management, 187-193
 - closures and, 192
 - reference counting, 187
 - retain cycles, 188, 192
 - strong references, 188
 - unowned references, 192
 - weak references, 190
- methods, 120-125
 - extensions, 156
 - in enumerations, 148
 - in structures, 140
 - optional, in protocols, 165

- overriding, 128
- parameter names, local/external, 121-121
- required in protocols, 165
- self property and, 121
- subscripts, 123-125
- type, 122

min() global function, 212

multiplyWithOverflow() function, 35

mutable dictionaries, 59

mutable sets, 63

MutableCollectionType protocol, 178

MutableIndexable protocol, 178

MutableSliceable, 179

mutating methods, 140, 149, 157

N

naming conventions, 16

nested types

- default access level of, 154

null escape sequence, 42

numeric literals, 19

numericCast() global function, 212

O

Objective-C, 1

operators, 30-39

- arithmetic, 31
- assignment, 33-34
- binary, 30
- binary, overloading, 202
- bitwise, 32
- comparison, 34
- custom, 205
- implicit type conversion, 30
- overflow, 35
- overloading, 201-205
- precedence, 37-39
- range, 36

- ternary, 30
- ternary conditional, 37
- type casting, 36
- unary, 30
- unary, overloading, 204

option sets, 67

optional binding, 84

optional tuple return type, 71

optionals, 81-87

- as return value, 69
- binding, 84
- implicitly unwrapped, 83
- Objective-C pointers vs., 81
- testing value, 84
- unwrapping, 82

OptionSetType protocol, 179

OS X 10.10 (Yosemite), 5

OS X 10.11 (El Capitan), 5

OutputStreamType protocol, 179

overflow operators, 31, 35

overlaps() method (intervals), 208

overridden superclass entities, 126-129

- accessing, 126
- initializers, 138
- methods, 128
- properties, 126-128
- subscripts, 128

P

parameters, function, 68

- default values for, 72
- external names for, 71
- local names for, 71
- variadic, 73

patterns, 96

Perl, 8

playground, 8-12

- creating, 5

pointers

- in Objective-C, 81

precedence

- custom operators, 206

- operators, 37-39
- preconditionFailure() global function, 212, 213
- predecessor() method, 46
- prefix() method (arrays), 54
- prefixes, finding in strings, 42
- print, 210-215
- print() global function, 213
- private access control level, 151
- program flow, 88
 - conditional execution, 93-102
 - loops, 88-93
- properties
 - computed, 112-114, 156
 - computed type, 117
 - constant, 119
 - default access level of, 153
 - in classes, 110
 - in structures, 140
 - instance vs. type, 115
 - optional, in protocols, 165-168
 - overriding, 126-128
 - required in protocols, 164
 - static, 116
 - stored, 110
- property observers, 114
- protocols, 162-186
 - adopting with extensions, 168
 - built-in, 175
 - checking conformance of, 172
 - default access level of, 154
 - default implementations with extensions, 173
 - generic, 200-201
 - inheritance and, 169
 - initializers, 168
 - literal convertible, 175
 - optional methods, 165-168
 - optional properties, 165-168
 - required methods, 165
 - required properties, 164
 - using as types, 170-172

- public access control level, 151
- Python, 8

Q

- qsort() (C standard library), 75
- Quick Look view (Xcode), 10

R

- RandomAccessIndexType protocol, 179
- range operators, 36
- RangeReplaceableCollectionType protocol, 179
- ranges, 206
 - matching, in case clauses, 98
 - value binding with, 99
- raw values (enumerations), 145
- RawRepresentable protocol, 179
- readLine() global function, 213
- recursive enumerations, 150
- reduce() method (arrays), 54
- reference counting, 187
- remainderWithOverflow() function, 35
- removeAll() method
 - in strings, 47
- removeAtIndex() method
 - in arrays, 51
 - in sets, 66
 - in strings, 47
- removeRange() method
 - in strings, 47
- repeat-while loops, 92
- required initializers, 138
- required properties, 164
- reserveCapacity() method
 - in arrays, 51
 - in strings, 47
- reserved words, 24
- retain cycles, 188
- rethrows annotation, 105
- return values

- multiple, 70
- optional, 69
- tuples as, 29

reverse() method (arrays), 55

ReverseIndexType protocol, 179

Ruby, 8

Run-Evaluate-Print-Loop (REPL), 6-8

- starting, 7

S

scope

- capturing values by reference, 80
- closures and, 79-80
- creating new, with do statement, 101

self property, 121

sequences, 184

SequenceType protocol, 179, 184

SetAlgebraType protocol, 179

sets, 62-68

- accessing, 63
- comparing, 62
- contains(), 63
- count property, 63
- endIndex property, 66
- exclusiveOr() method, 65
- exclusiveOrInPlace() method, 63
- forEach() method, 66
- indexOf() method, 66
- inherited functionality, 66
- insert() method, 63
- intersect() method, 65
- intersectInPlace() method, 64
- isDisjointWith() method, 65
- isEmpty property, 63
- isStrictSubsetOf() method, 65
- isStrictSupersetOf() method, 65
- isSubsetOf() method, 65
- isSupersetOf() method, 66

- iterating over, 64
- modifying, 63
- mutable, 63
- operations on, 65
- properties, 63
- remove() method, 64
- removeAll() method, 64
- removeAtIndex() method, 66
- removeFirst() method, 64
- startIndex property, 67
- subtract() method, 66
- subtractInPlace() method, 64
- union() method, 66
- unionInPlace() method, 64

setter functions

- computed properties, 112
- computed variables, 24
- default access level of, 154

SignedIntegerType protocol, 180

SignedNumberType protocol, 180

sizeof() global function, 174, 213

sizeofValue() global function, 213

slices, 56

sort() method (arrays), 55, 76

split() method (arrays), 55

startIndex property

- in dictionaries, 62
- in sets, 67
- in strings, 46

startsWith() method

- in arrays, 55

statement labels, 101

static properties, 116

stored properties, 110

Streamable protocol, 180

Strideable protocol, 180

strideof() global function, 213

strideofValue() global function, 213

strides, 209

StringInterpolationConvertible protocol, 180

strings, 39-47

- append() method, 47
- appendContentsOf() method, 47
- characters property, 40, 44
- comparing, 41
- concatenation of, 40
- converting to numeric types, 43
- count property, 45
- endIndex property, 46
- escaped characters in, 42
- hasPrefix() method, 42
- hasSuffix() method, 42
- inherited functionality, 46
- interpolation, 43
- isEmpty property, 40
- length of, 40
- literals, 20
- lowercase property, 40
- NSString extensions, 47
- properties of, 40
- removeAll() method, 47
- removeAtIndex() method, 47
- removeRange() method, 47
- reserveCapacity() method, 47
- startIndex property, 46
- unicodeScalars property, 41
- unicodeScalars property, 45
- uppercase property, 40
- utf16 property, 41, 45
- utf8 property, 41, 44
- strong references, 188
- structures, 139-143
 - initializers delegation in, 143
 - initializers for, 142
 - methods in, 140
 - mutating methods, 140
 - properties in, 140
 - type methods for, 141
- subclass, 106
- subclasses, 126-129
 - default access level of, 154
- subscripts, 123-125
 - default access level of, 154
 - extensions, 157
 - overriding, 128
- subtractWithOverflow() function, 35
- successor() method, 46
- suffix() method (arrays), 56
- suffixes, finding in strings, 42
- super prefix, 126
- superclasses, 106
 - deinitializer, inheritance of, 139
 - initialization and, 130
 - overriding entities, 126-129
 - protocol inheritance and, 164
- swap() global function, 213
- Swift
 - access control, 151-155
 - arrays, 48-57
 - as scripting language, 8
 - classes, 107-139
 - constants, 23
 - data types, 18-22
 - dictionaries, 57-62
 - functions, 68-75
 - generics, 193-201
 - importing modules in, 16
 - loops, 88-93
 - memory management, 187-193
 - operators, 30-39
 - playground, 8-12
 - program flow, 88
 - protocols, 162-186
 - reserved words, 24
 - sets, 62-68
 - simple program in, 12-14
 - structures, 139-143
 - tuples, 27-30
 - variables, 23-27
 - Xcode, 5-12
- switch statements, 96-102

- matching ranges in case clauses, 98
 - statement labels, 101
 - using tuples in case clause, 98
 - using with enumerations, 100
 - value binding, 99
 - where qualifier, 100
- T**
- ternary conditional operator, 37
- ternary operator, 30
- throw statement, 103
- throws annotation, 103
- transcode() global function, 214
- try statement, 103
- try! statement, 106
- try? statement, 106
- tuples, 27-30
 - as return type, 29
 - as return values, 70
 - constants, 28
 - default access level of, 155
 - extracting components of, 28
 - naming components, 28
 - using type aliases with, 29
 - value binding with, 99
 - variables, 28
- type aliases, 21
 - default access level of, 155
 - using with tuples, 29
- type casting
 - as! operator, 36, 36
 - as? operator, 36
- type casting operators, 36
- type inferencing, 23
 - tuples and, 29
- type methods
 - for structures, 141
 - in enumerations, 149
- type placeholders, 195
- type properties
 - in structures, 140
- typealias keyword, 21, 200
- types, 18-22
 - character literals, 20
 - checking, 159
 - downcasting, 160-162
 - generic, 195
 - integer, 18
 - numeric literals, 19
 - string literals, 20
 - using protocols as, 170-172
- U**
- UIKit Framework, 9
- UInt types, 18
- unary operators, 30
 - overloading, 204
- Unicode, 1
 - \u{n} arbitrary Unicode scalar character, 42
- UnicodeCodecType protocol, 180
- UnicodeScalars format of strings, 41
- unowned references, 192
- unsafeAddressOf() global function, 214
- unsafeBitCast() global function, 214
- unsafeDowncast() global function, 214
- unsafeUnwrap() global function, 214
- UnsignedIntegerType protocol, 180
- unwrapping optionals, 82
- UTF-16, view of string in, 41
- UTF-8, view of string in, 41
- V**
- value types
 - arrays as, 49
 - dictionaries as, 58
 - sets as, 63
 - structures as, 139

- values property (dictionaries), 59
- var keyword
 - declaring variables, 23
 - function parameters and, 68
 - value binding with, 99
- variable parameters, 68
- variables, 23-27
 - computed, 24
 - default access level of, 153
 - observers, 25-27
 - tuples, 28
- variadic parameters, 73

W

- weak references, 190
- where
 - as type constraint, 198
 - in for-in statement, 91
 - in guard-else statement, 95
 - in if-else statement, 94
 - in switch statement, 100
 - in while statement, 92
- while loops, 91
 - case pattern, 92
 - where clause, 92
- whitespace, 15
- willSet keyword
 - property observers, 114

- variable observers, 25
- withExtendedLifetime() global function, 214
- withUnsafeMutablePointer() global function, 214
- withUnsafeMutablePointers() global function, 214
- withUnsafePointer() global function, 215
- withUnsafePointers() global function, 215
- withVaList() global function, 215
- Worldwide Developers Conference (2014), 12

X

- Xcode, 5-12
 - multiple installs of, 6
 - new projects, creating, 5
 - playground, 8-12
 - playground, creating, 5
 - Swift REPL, 6-8
- xcode-select command, 6

Z

- zip() global function, 215

About the Author

Anthony Gray (you can call him Tony) has over 25 years of experience working in tertiary education, where he's provided technical and systems support for academic and research staff, and for some very smart students. He loves to teach, with his favorite subjects being operating systems, computer graphics and animation with OpenGL, and most recently mobile development for iOS. In his spare time, he writes software to scratch his own itch, some of which is available at squidman.net. Secretly he pines for the days when you could handcode assembler for your 6502 and occasionally writes emulators so he can do just that.

Colophon

The animal on the cover of *Swift Pocket Reference* is an African palm swift (*Cypsiurus parvus*). This bird seeks palm trees for dwelling in the savannas and grasslands of sub-Saharan Africa and of the Arabian Peninsula. 16 centimeters in length, with a thin body and a long tail, the African palm swift is mostly brown with a gray throat and a black bill. Differences in coloring between genders (mostly in the tail) lessen with age. To avoid the ground, these birds use their short purple legs to cling to vertical surfaces.

The species's population appears to be on the rise, thanks largely to growth in the planting of the Washington palm tree.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Wood's *Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.