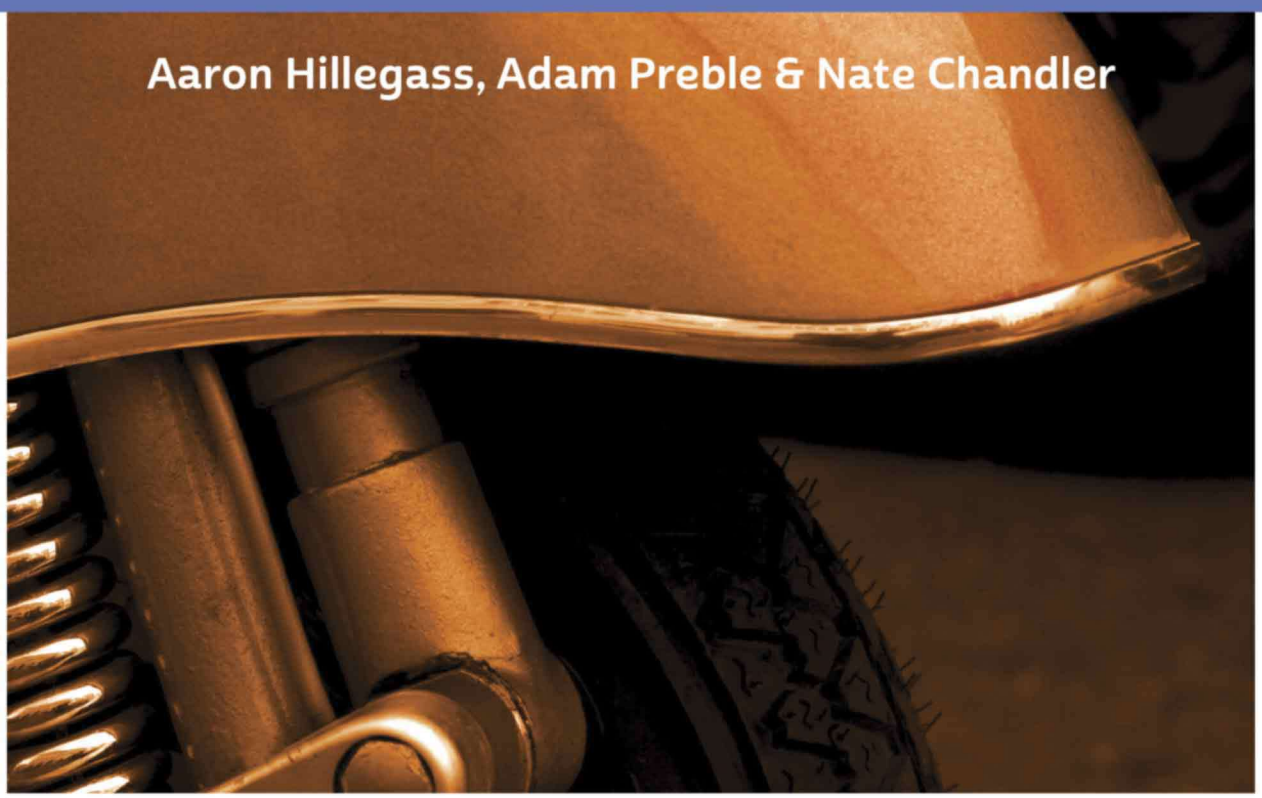
Big Nerd Ranch

5TH EDITION

Cocoa Programming for OS X

THE BIG NERD RANCH GUIDE

Aaron Hillegass, Adam Preble & Nate Chandler



Cocoa Programming for OS X: The Big Nerd Ranch Guide

by Aaron Hillegass, Adam Preble and Nate Chandler

Big Nerd Ranch, LLC.
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0134077113
ISBN-13 978-0134077116

Fifth edition, first printing, April 2015
Release D.5.1.1

For Aaron's sons, Walden and Otto

~

For Adam's daughters, Aimee and Leah

~

For Nate's nieces and nephews

This page intentionally left blank

Acknowledgments

Creating this book required the efforts of many people. We want to thank them for their help. Their contributions have made this a better book than we could have ever written alone.

Thanks to the students who took the Cocoa programming course at the Big Nerd Ranch. They helped us work the kinks out of the exercises and explanations that appear here. Their curiosity inspired us to make the book more comprehensive, and their patience made it possible.

Thank you to all the readers of the first four editions who made such great suggestions on our forums (http://forums.bignerdranch.com/).

Thank you to our technical reviewers, Juan Pablo Claude, Chris Morris, Nick Teissler, Pouria Almassi, and John Gallagher, who made great additions and caught many of our most egregious errors.

Finally, a very big thank you to our support team in writing this book: Liz Holaday, for copy-editing; Chris Loper, whose excellent tool chain made writing this book that much easier; and most of all Susan Loper, whose collaboration helped us write the kind of book we believe this technology deserves.

This page intentionally left blank

Table of Contents

Introduction

If you are developing applications for OS X, or are hoping to do so, this book will be your foundation and will help you understand Cocoa, the set of frameworks for developing applications for OS X. You, the developer, are going to love developing for OS X because Cocoa will enable you to write full-featured applications in a more efficient and elegant manner.

About This Book

This book covers the major design patterns of Cocoa and includes an introduction to the Swift language. It will also get you started with the most commonly-used developer tools: Xcode and Instruments. After reading this book, you will understand these major design patterns which will enable you to understand and use Apple's documentation – a critical part of any Cocoa developer's toolkit – as well as build your own Cocoa applications from scratch.

This book teaches ideas and provides hands-on exercises that show these ideas in action. Each chapter will guide you through the process of building or adding features to an application.

Often, we will ask you to do something and explain the details or theory afterward. If you are confused, read a little more. Usually, the help you seek will be only a paragraph or two away.

Because of the hands-on nature of the book, it is essential that you do the exercises and not just read the words. Doing the exercises will help build the kind of solid understanding that will enable you to develop on your own when you are finished with this book. You will also learn a great deal from making mistakes, reading error messages, and figuring out what went wrong – practical experience you can't get from reading alone. At first, you may want to stick with what we show you, but later in the book when you are more comfortable with the environment, you should feel free to experiment with the exercises and add your own ideas.

Most chapters end with one or two challenge exercises. These exercises are important to do as well. Taking on these challenges gives you the opportunity to test your skills and problem-solve on your own.

You can get help with this book at bignerdranch.com/books, where you will find errata and downloadable solutions for the exercises. You can also post questions and find relevant conversations on the Big Nerd Ranch forums at forums.bignerdranch.com.

We ask that you not use the downloadable solutions as a shortcut for doing the exercises. The act of typing in code has far more impact on your learning than most people realize. By typing the code yourself (and, yes, making mistakes), you will absorb patterns and develop instincts about Cocoa programming, and you will miss out on these benefits if you rely on the solutions or copy and paste the code instead.

There is a lot of code in this book. Through that code, we will introduce you to the idioms of the Cocoa community. Our hope is that by presenting exemplary code, we can help you to become more than a Cocoa developer – a stylish Cocoa developer.

Most of the time, Cocoa fulfills the following promise: Common things are easy, and uncommon things are possible. If you find yourself writing many lines of code to do something rather ordinary, you are probably on the wrong track. There is a popular adage in the community which you should bear in mind: *Don't fight the framework*. Cocoa is opinionated and you will benefit greatly from adapting your way of doing things to its way of doing things.

Prerequisites

This book is written for programmers and assumes that you are familiar with basic programming concepts (like functions, variables, and loops) as well as object-oriented concepts (like classes, objects, and inheritance). If you do not fit this profile, you will find this book tough going. You are not expected to have any experience with Mac programming.

One of the challenges of learning Cocoa programming is learning the Swift language. If you have a basic foundation in programming and know something about objects, you will find learning Swift to be easy. This book includes three chapters to introduce to you to the language. Then you will learn more Swift as you build Cocoa applications throughout the book. If you would prefer a gentler introduction, start with Apple's *The Swift Programming Language*, available in the iBooks store or from developer.apple.com/swift, offers a more gentle introduction. Or, if you can wait until Summer 2015, you can read *Swift Programming: The Big Nerd Ranch Guide* first.

This is a hands-on book and assumes that you have access to OS X and the developer tools. The book requires OS X Yosemite (10.10) or higher. The exercises are written for Xcode 6.3 and Swift 1.2.

We strongly recommend that you join Apple's Mac Developer Program at developer.apple.com/programs. Joining the program gives you access to pre-release versions of Xcode and OS X. These can be very useful when trying to stay ahead of Apple's development curve. In addition, you must be a member of the developer program to distribute your apps on the App Store.

Typographical conventions

To make the book easier to follow, we have used several typographical conventions.

In Swift, class names are always capitalized. In this book, we have also made them appear in a monospaced bold font. In Swift, method names start with a lowercase letter. Here, method names will also appear in a monospaced bold font. For example, you might see "The class **`NSWindowController`** has the method **`showWindow(_:)`**."

Other literals, including instance variable names that you would see in code, will appear in a regular monospaced font. Also, filenames will appear in this same font. Thus, you might see "In `MyClass.swift`, set the optional `favoriteColor` to `nil`."

Code samples in this book appear in the regular monospaced font. New portions, which you will need to type yourself, will appear in bold. Code that you should delete is struck-through.

What's new in the fifth edition?

This fifth edition includes technologies introduced in OS X 10.8, 10.9, and 10.10. It is updated for Xcode 6.3 and Swift 1.2. It includes coverage of Swift basics, Auto Layout, unit testing, view controllers and expanded coverage of view swapping, storyboards, modernized localization and web services APIs, JSON parsing, Key-Value Validation, and a strong emphasis on demonstrating best practices for application architecture.

The Story of Cocoa

Once upon a time, two guys named Steve started a company called Apple Computer in their garage. The company grew rapidly, so they hired an experienced executive named John Sculley to be its CEO.

After a few conflicts, John Sculley moved Steve Jobs to a position where he had no control over the company. Steve Jobs left to form another computer company, NeXT Computer.

NeXT hired a small team of brilliant engineers. This small team developed a computer, an operating system, a printer, a factory, and a set of development tools. Each piece was years ahead of competing technologies. Unfortunately, the computer and the printer were commercial failures. In 1993, the factory closed, and NeXT Computer, Inc. became NeXT Software, Inc. The operating system and the development tools continued to sell under the name NeXTSTEP.

NeXTSTEP and OpenStep

NeXTSTEP was very popular with scientists, investment banks, and intelligence agencies. These groups found that NeXTSTEP enabled them to turn their ideas into applications faster than any other technology. In particular, NeXTSTEP had three important features:

a Unix-based operating system

NeXT decided to use Unix as the core of NeXTSTEP. It relied on the source code for BSD Unix from the University of California at Berkeley. Why Unix? Unix crashed much less frequently than Microsoft Windows or Mac OS and came with powerful, reliable networking capabilities.

a powerful window server

A *window server* takes events from the user and forwards them to the applications. The application then sends drawing commands back to the window server to update what the user sees. One of the nifty things about the NeXT window server is that the drawing code that goes to the window server is the same drawing code that would be sent to the printer. Thus, a programmer has to write the drawing code only once, and it can then be used for display on the screen or printing.

If you have used Unix machines before, you are probably familiar with the X window server. The window server for OS X is completely different but fulfills the same function as the X window server: It gets events from the user, forwards them to the applications, and puts data from the applications onto the screen.

an elegant set of libraries and tools

NeXTSTEP came with a set of libraries and tools to enable programmers to deal with the window server in an elegant manner. The libraries were called frameworks. In 1993, the frameworks and tools were revised and renamed OpenStep.

Programmers loved OpenStep because they could experiment more easily with new ideas. In fact, Tim Berners-Lee developed the first web browser and web server on NeXTSTEP using the OpenStep libraries and tools. Securities analysts could code and test new financial models much more quickly. Colleges could develop the applications that made their research possible. We do not know what the intelligence community was using it for, but they bought thousands of copies of OpenStep.

From NeXTSTEP to OS X to iOS

For many years, Apple Computer had been working to develop an operating system with many of the same features as NeXTSTEP. This effort, known as Project Copland, gradually spun out of control,

and Apple finally decided to pull the plug and buy the next version of Mac OS instead. After surveying the existing operating systems, Apple selected NeXTSTEP. Because NeXT was small, Apple simply bought the whole company in December 1996. In 1997, Steve Jobs returned to Apple.

NeXTSTEP became Mac OS X, and OpenStep became Cocoa. In 2001, the first desktop version of Mac OS X was released with several more to follow. In 2012, Apple dropped the "Mac," and the operating system became known as OS X.

The mutation of NeXTSTEP didn't stop with OS X. iOS, the operating system for iPhones and iPads, is based on OS X, and iOS's Cocoa Touch is built on the same foundations as Cocoa. As a developer you will find that your knowledge transfers well between the two: the design patterns are identical, and many of the APIs are very similar if not the same.

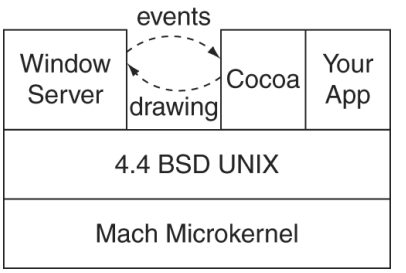
OSX, Unix, and Cocoa

OS X is Unix underneath, and you can get all the standard Unix programs (such as the Apache Web server) on OS X. It is extremely stable, and the user interface is spectacular.

(Apple has made the source code to the Unix part of OS X available under the name Darwin. A community of developers continues to work to improve Darwin. You can learn more about Darwin at www.macosforge.org.)

As shown in Figure 1, the window server and your application are Unix processes. Cocoa is your application's interface to the window server to receive events and draw to the screen. At the same time it has access to the Unix layer where it can make lower level calls.

Figure 1 Where is Cocoa?



Introducing the Swift Language

Programming in Cocoa was initially done in a language called *Objective-C*. Objective-C is an extension of the C programming language that adds constructs for object-oriented programming. In that respect it bears a superficial resemblance to C++, but the two are extremely different. Unlike C++, Objective-C is weakly typed and extremely powerful. With power comes responsibility: Objective-C also allows programmers to make ridiculous errors.

Over the past several years, Apple's engineers have gone to heroic lengths to make Objective-C faster and add more modern features, but in order to move forward, a new language was needed, free of the limitations of the past. Swift, developed by a small team led by Chris Lattner, was the answer. Apple introduced Swift in 2014.

Swift maintains the expressiveness of Objective-C while introducing a syntax that is significantly more rich, succinct, and – in the opinion of some – readable. It emphasizes type safety and introduces advanced features such as optionals and generics. Swift is much stricter than Objective-C and will not allow you to make as many ridiculous errors.

Although we will focus on Swift, you can still write Cocoa code in Objective-C, even alongside Swift, compiling the two in the same project.

Most importantly, Swift allows the use of these new features while relying on the same tested, elegant Cocoa frameworks that developers have built upon for years and years.

The Cocoa Frameworks

A *framework* is a collection of classes that are intended to be used together. That is, the classes are compiled together into a reusable library of binary code. Any related resources are put into a directory with the library. The directory is given the extension `.framework`. You can find the built-in frameworks for your machine in `/System/Library/Frameworks`. Cocoa is made up of three frameworks:

- *Foundation*: Every object-oriented programming language needs the standard value, collection, and utility classes. Strings, dates, lists, threads, and timers are in the Foundation framework. All Cocoa apps, from command-line tools to fully-featured GUI apps, use Foundation. Foundation is also available on iOS.
- *AppKit*: All things related to the user interface are in the AppKit framework. These include windows, buttons, text fields, events, and drawing classes. AppKit is built on top of Foundation and is used in practically every graphical application on OS X.
- *Core Data*: Core Data makes it easy to save objects to a file and then reload them into memory. It is a *persistence* framework.

In addition to the three Cocoa frameworks, over a hundred frameworks ship with OS X. The frameworks offer a wide variety of features and functionality. For example, AVFoundation is great for working with audio and video, AddressBook provides an API to the user's contacts (with their permission), and and SpriteKit is a full-featured 2D game engine with physics. You can pick and choose from these frameworks to suit the needs of your application. You can also create your own frameworks from the classes that you create. Typically, if a set of classes is used in several applications, you will want to turn them into a framework.

This book will focus on the Cocoa frameworks and especially Foundation and AppKit because they will form the basis of most Cocoa applications that you will write. Once you have mastered these, other frameworks will be easier to understand.

Tools for Cocoa Programming

Xcode is the IDE (integrated development environment) used for Cocoa development.
Xcode is available for free on the Mac App Store. Pre-release versions can be downloaded at developer.apple.com/mac. (You will need to join Apple's Mac Developer Program to access these.) We strongly recommend using Xcode 6.3 with Swift 1.2 or later for the exercises in this book.

Xcode tracks all the resources that go into an application: code, images, sounds, and so on. You edit your code in Xcode, and Xcode compiles and launches your application. Xcode can also be used to

invoke and control the debugger. Behind the scenes, swiftc (Apple's Swift compiler) will be used to compile your code, and LLDB (Low Level Debugger) will help you find your errors.

Inside Xcode, you will use the *Interface Builder* editor as a GUI builder to lay out windows and add UI elements to those windows. But Interface Builder is more than a simple GUI builder. In Interface Builder, you can create objects and edit their attributes. Most of those objects are UI elements from the AppKit framework such as buttons and text fields, but some will be instances of classes that you create.

You will use *Instruments* to profile your application's CPU, memory, and filesystem usage. Instruments can also be used to debug memory-management issues. Instruments is built on top of dtrace, which makes it possible to create new instruments.

Some Advice on Learning

All sorts of people come to our class: the bright and the not so bright, the motivated and the lazy, the experienced and the novice. Inevitably, the people who get the most from the class share one characteristic: they remain focused on the topic at hand.

The first trick to maintaining focus is to get enough sleep: ten hours of sleep each night while you are studying new ideas. Before dismissing this idea, try it. You will wake up refreshed and ready to learn. *Caffeine is not a substitute for sleep.*

The second trick is to stop thinking about yourself. While learning something new, many students will think, "Damn, this is hard for me. I wonder if I am stupid." Because stupidity is such an unthinkably terrible thing in our culture, they will then spend hours constructing arguments to explain why they are intelligent yet having difficulties. The moment you start down this path, you have lost your focus.

Aaron used to have a boss named Rock. Rock earned a degree in astrophysics from Cal Tech, but never had a job that used his knowledge of the heavens. When asked if he regretted getting the degree, he replied, "Actually, my degree in astrophysics has proved to be very valuable. Some things in this world are just hard. When I am struggling with something, I sometimes think 'Damn, this is hard for me. I wonder if I am stupid,' and then I remember that I have a degree in astrophysics from Cal Tech; I must not be stupid."

Before going any further, assure yourself that you are not stupid and that some things are just hard. Armed with this affirmation and a well-rested mind, you are ready to conquer Cocoa.

3
Structures and Classes

At this point you should be somewhat familiar with using Swift's standard types: strings, arrays, enums, etc. It is time to move on to bigger and better things: defining your own types. In this chapter, you will build a simple 2D physics simulation. You will create your own structure and a few classes, and you will learn about the differences between them.

Structures

In Cocoa, structures are typically used to represent groupings of data. For example, there is `NSPoint`, which represents a point in 2D space with an X and a Y value. As your first structure you will create a 2D vector structure.

Create a new playground. From Xcode's File menu, select New... → Playground. Name the playground `Physics` and save it with the rest of your projects.

Start by defining the **`Vector`** structure:

```
import Cocoa

struct Vector {
    var x: Double
    var y: Double
}
```

Much like C structures, Swift structures are composite data types. They are composed of one or more fields, or *properties*, each of which has a specified type. A few lines down, create an instance of **`Vector`** and access its properties:

```
let gravity = Vector(x: 0.0, y: -9.8)       // {x 0, y -9.800000000000001}
gravity.x                                   // 0
gravity.y                                   // -9.800000000000001
```

You just used Swift's *automatic initializer* to create an instance of this structure. The automatic initializer has a parameter for each property in the structure. If you were to add a `z` field, this code would cause a compiler error because it lacks a `z` parameter. (Do not worry about the zeros; that is just typical floating point fun.)

You can provide your own initializers, but when you do, the automatic initializer is no longer provided. Go back to **`Vector`** and add an initializer that takes no parameters and initializes `x` and `y` to `0`.

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
    }
}
```

Initializers in Swift use the `init` keyword, followed by the parameter list, and then the body of the initializer. Within the body, the `x` and `y` properties are assigned directly.

An initializer *must* initialize all of the properties of its structure.

As we warned, defining this initializer has caused the automatic one to vanish, causing an error in the playground. You can easily define it manually, however:

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
    }

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```

A Swift programmer would say that this initializer takes two parameters, `x` and `y`, both of type `Double`.

What is `self`? It represents the instance of the type that is being initialized. Using `self.`*`propertyName`* is usually unnecessary (you did not use it in **`init()`**), but because the initializer's parameter names match the names of the properties you must use `self` to tell the compiler that you mean the property and not the parameter.

Before continuing, let's make an improvement. As the **`Vector`** structure stands, its two initializers have independent code paths. It would be better to have them use one code path by having the parameterless initializer call the initializer which takes both `x` and `y`.

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
        self.init(x: 0, y: 0)
    }

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```

A single code path for initialization is not required for structures, but it is a good habit to get into as you will use it when working with classes.

Instance methods

Methods allow you to add functionality to your data types. In Swift, you can add methods to structures as well as classes (and enums!). *Instance* methods operate within the context of a single instance of the type. Add an instance method for multiplying a vector by a scalar:

```
struct Vector {
    ...

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }

    func vectorByAddingVector(vector: Vector) -> Vector {
        return Vector(x: self.x + vector.x,
                      y: self.y + vector.y)
    }
}
```

The `func` keyword in the context of a structure indicates that this is a method. It takes a single parameter of type `Double` and returns an instance of **`Vector`**.

Try this new method out:

```
let gravity = Vector(x: 0.0, y: -9.8)       // {x 0, y -9.800000000000001}
gravity.x
gravity.y
let twoGs = gravity.vectorByAddingVector(gravity)  // {x 0, y -19.6}
```

What is the name of this method? In conversation you would call it `vectorByAddingVector`, but in this text we include parameters, like this: **`vectorByAddingVector(_:)`**. By default, the first parameter of a method is not named – thus the underscore.

Why not name the first parameter? Because the convention – inherited from Objective-C and Cocoa – is that the base name of the method includes the name of the first parameter, in this case `Vector`. Suppose you added another parameter to that method. What would it look like?

```
    func vectorByAddingVector(vector: Vector, numberOfTimes: Int) -> Vector {
        var result = self
        for _ in 0..<numberOfTimes {
            ...
```

This method would be called **`vectorByAddingVector(_:numberOfTimes:)`**. Note that there is a colon for each parameter.

This can lead to verbose method names, but the code actually becomes very readable. No guessing or relying on the IDE to tell you what the third parameter is!

By default, each parameter's internal name is the same as its external name (except the first parameter, that is). In **`vectorByAddingVector(_:numberOfTimes:)`**, the second parameter is named

`numberOfTimes`. That is certainly very descriptive, but you might prefer to use a shorter name (like `times`) within the method. In that case you would explicitly set the internal parameter name like this:

```
    func vectorByAddingVector(vector: Vector, numberOfTimes times: Int) -> Vector {
        var result = self
        for _ in 0..<times {
            ...
```

The method's signature has not changed. For those calling it, its name is still **`vectorByAddingVector(_:numberOfTimes:)`**, but internally you have the satisfaction of using the name you want.

Using self in instance methods

As in initializers, `self` represents the instance that the method is being called on. As long as there is no conflict with named parameters or local variables, however, it is entirely optional, so we prefer to leave it off. Make this change to **`vectorByAddingVector(_:)`**.

```
struct Vector {
    ...

    func vectorByAddingVector(vector: Vector) -> Vector {
        return Vector(x: self.x + vector.x,
                      y: self.y + vector.y)
        return Vector(x: x + vector.x,
                      y: y + vector.y)
    }
}
```

Operator Overloading

By overloading operators you can make your own types work with common (and even uncommon) operators. This ability falls deep beyond the "with great power comes great responsibility" line. However, vectors are a natural and respectable application for this technique.

To define an operator overload you simply add a function that takes the appropriate types. To start with, instead of calling **`vectorByAddingVector(_:)`**, it would be nice to use the + operator. Overload + and * for adding and scaling vectors, respectively.

```
struct Vector {
    ...
}

func +(left: Vector, right: Vector) -> Vector {
    return left.vectorByAddingVector(right)
}
func *(left: Vector, right: Double) -> Vector {
    return Vector(x: left.x * right , y: left.y * right)
}
```

Now you can very succinctly manipulate vectors:

```
let twoGs = gravity.vectorByAddingVector(gravity)
let twoGs = gravity + gravity
let twoGsAlso = gravity * 2.0
```

Note that the order of types for binary operators like `*` and `+` is important. In order to write `2.0 * gravity` you will need to implement another operator overload function:

```
func *(left: Double, right: Vector) -> Vector {
    return right * left
}
```

Classes

Now that you have the beginnings of a robust vector type, let's put it to work. Your 2D physics simulation will consist of two classes: **`Particle`**, which represents a single moving object within the simulation, and **`Simulation`**, which contains an array of **`Particle`** instances.

Classes are very similar to structures. They have a lot of the same features: initializers, properties, computed properties, and methods. They have a significant difference, however, which we will discuss once the simulation is up and running.

Start by defining the **`Particle`** class in your playground. The position is not important, as long as it is above or below (but not inside!) the **`Vector`** structure. A **`Particle`** has three **`Vector`** properties: `position`, `velocity`, and `acceleration`.

```
struct Vector {
    ...
}

class Particle {

    var position: Vector
    var velocity: Vector
    var acceleration: Vector

}
```

Classes and structures differ significantly in terms of initializers. Most noticeably, classes do not have automatic initializers, so you will see a compiler error: Class 'Particle' has no initializers.

Fix this by adding an initializer to **`Particle`**:

```
class Particle {

    var position: Vector
    var velocity: Vector
    var acceleration: Vector

    init(position: Vector) {
        self.position = position
        self.velocity = Vector()
        self.acceleration = Vector()
    }
}
```

You do not need to provide a parameter for every property in a class like you did in **`Vector`**'s **`init(x:y:)`**. You just need to initialize everything. As with initializers for structures, a class's

initializer must initialize all of its properties before returning or performing any other tasks. By requiring this of initializers the Swift compiler guarantees that every instance is fully initialized before it is put to work.

Another approach is to give properties default values:

```
class Particle {

    var position: Vector
    var velocity: Vector = Vector()
    var acceleration: Vector = Vector()

    init(position: Vector) {
        self.position = position
    }
}
```

In a simple case like this, there is not a clear benefit to either approach.

Designated and convenience initializers

Like structures, classes can have multiple initializers. At least one of them will be the *designated initializer*. Remember how you refactored **`Vector`**'s **`init()`** to call **`init(x:y:)`**? A designated initializer is an initializer which other, non-designated initializers – *convenience initializers* – must call. The rule of thumb with designated initializers is that they are typically the one with the most parameters. Most classes will only have one designated initializer.

The **`init(position:)`** initializer is the **`Particle`** class's designated initializer. Add a convenience initializer:

```
class Particle {
    ...

    init(position: Vector) {
        self.position = position
        self.velocity = Vector()
        self.acceleration = Vector()
    }

    convenience init() {
        self.init(position: Vector())
    }

}
```

There is an exception to these designated initializer rules: required initializers, which you will see in Chapter 12.

Add an instance method

A particle has a position, velocity, and acceleration. It should also know a little about particle dynamics – specifically, how to update its position and velocity over time. Add an instance method, **`tick(_:)`**, to perform these calculations.

```
class Particle {
    ...

    convenience init() {
        self.init(position: Vector())
    }

    func tick(dt: NSTimeInterval) {
        velocity = velocity + acceleration * dt
        position = position + velocity * dt
        position.y = max(0, position.y)
    }

}
```

The **`tick(_:)`** method takes an `NSTimeInterval` parameter, `dt`, the number of seconds to simulate. `NSTimeInterval` is an alias for `Double`.

Below the definition of **`Particle`**, define the **`Simulation`** class, which will have an array of **`Particle`** objects and its own **`tick(_:)`** method:

```
class Particle {
    ...
}

class Simulation {

    var particles: [Particle] = []
    var time: NSTimeInterval = 0.0

    func addParticle(particle: Particle) {
        particles.append(particle)
    }

    func tick(dt: NSTimeInterval) {
        for particle in particles {
            particle.acceleration = particle.acceleration + gravity
            particle.tick(dt)
            particle.acceleration = Vector()
        }
        time += dt
    }

}
```

The **`Simulation`** class has no initializers defined since all of its properties have default values. The `for-in` loop iterates over the contents of the `particles` property. The **`tick(_:)`** method applies constant acceleration due to gravity to each of the particles before simulating them for the time interval.

Before you warm up the simulator and add a particle, add a line to evaluate `particle.position.y`. You will use this shortly with the playground's Value History. Additionally, add some code to remove particles once they drop below `y = 0`:

```
class Simulation {
    ...

    func tick(dt: NSTimeInterval) {
        for particle in particles {
            particle.acceleration = particle.acceleration + gravity
            particle.tick(dt)
            particle.acceleration = Vector()
            particle.position.y
        }
        time += dt
        particles = particles.filter { particle in
            let live = particle.position.y > 0.0
            if !live {
                println("Particle terminated at time \(self.time)")
            }
            return live
        }
    }

}
```

The last chunk of code filters the `particles` array, removing any particles that have fallen to the ground. This is a *closure*, and it is OK if you do not understand it at this point. You will learn more about closures in Chapter 15.

Now you are ready to run the simulator. Create an instance of the simulator and a particle, add the particle to the simulation, and see what happens.

```
class Simulation {
    ...
}

let simulation = Simulation()

let ball = Particle()
ball.acceleration = Vector(x: 0, y: 100)
simulation.addParticle(ball)

while simulation.particles.count > 0 && simulation.time < 500 {
    simulation.tick(1.0)
}
```

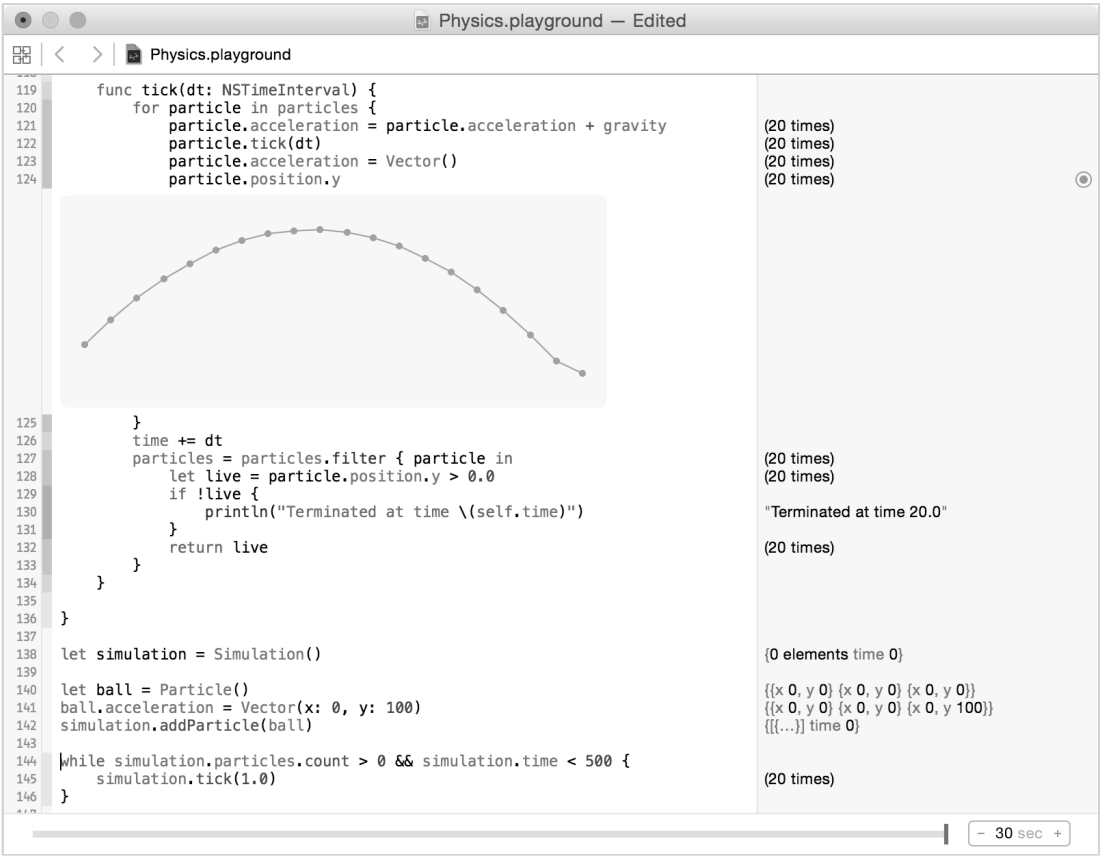
You should see the playground tally up (20 times) on a number of lines. If the playground runs the simulation continuously, you can stop it by commenting out the `while` loop. Select the three lines and hit Command-/ to toggle the comment marks:

```
// while simulation.particles.count > 0 && simulation.time < 500 {
//     simulation.tick(1.0)
// }
```

Double-check your code against the listings above, in particular the lines that filter the `particles` array and the line that increments `time`.

Once you have the simulation running as expected, click the Variables View circle in the playground sidebar on the line that reads `particle.position.y`, as shown in Figure 3.1. A graph will appear, showing the Y values of the particle over time. The X axis on this graph represents iterations over time and not the X coordinate of the particle.

Figure 3.1 Graph data history of particle.position.y



Inheritance

Suppose you wanted to simulate a particle that had different behavior than the **`Particle`** class you have already implemented: a rocket that propels itself with thrust over a certain period of time. Since **`Particle`** already knows about physics, it would be natural to extend and modify its behavior through subclassing.

Define the **`Rocket`** class as a subclass of **`Particle`**.

```
class Rocket: Particle {

    let thrust: Double
    var thrustTimeRemaining: NSTimeInterval
    let direction = Vector(x: 0, y: 1)

    convenience init(thrust: Double, thrustTime: NSTimeInterval) {
        self.init(position: Vector(), thrust: thrust, thrustTime: thrustTime)
    }

    init(position: Vector, thrust: Double, thrustTime: NSTimeInterval) {
        self.thrust = thrust
        self.thrustTimeRemaining = thrustTime
        super.init(position: position)
    }

}
```

The `thrust` property represents the magnitude of the rocket's thrust. `thrustTimeRemaining` is the number of seconds that the thrust will be applied for. `direction` is the direction that the `thrust` will be applied in.

Take a minute to go through the initializers you just typed in. Which is the designated initializer? (Remember the rule of thumb about designated initializers?)

In order to guarantee that a class's properties are initialized, initializers are only inherited if a subclass does not add any properties needing initialization. Thus, **`Rocket`** provides its own initializers and calls the superclass's designated initializer.

Next you will override the **`tick(_:)`** method, which will do a little math to calculate the acceleration due to thrust and apply it before calling the superclass's – **`Particle`**'s – **`tick(_:)`** method.

```
class Rocket: Particle {
    ...

    init(position: Vector, thrust: Double, thrustTime: NSTimeInterval) {
        self.thrust = thrust
        self.thrustTimeRemaining = thrustTime
        super.init(position: position)
    }

    override func tick(dt: NSTimeInterval) {
        if thrustTimeRemaining > 0.0 {
            let thrustTime = min(dt, thrustTimeRemaining)
            let thrustToApply = thrust * thrustTime
            let thrustForce = direction * thrustToApply
            acceleration = acceleration + thrustForce
            thrustTimeRemaining -= thrustTime
        }
        super.tick(dt)
    }

}
```

Finally, create an instance of **`Rocket`** and add it to the simulation in place of the ball:

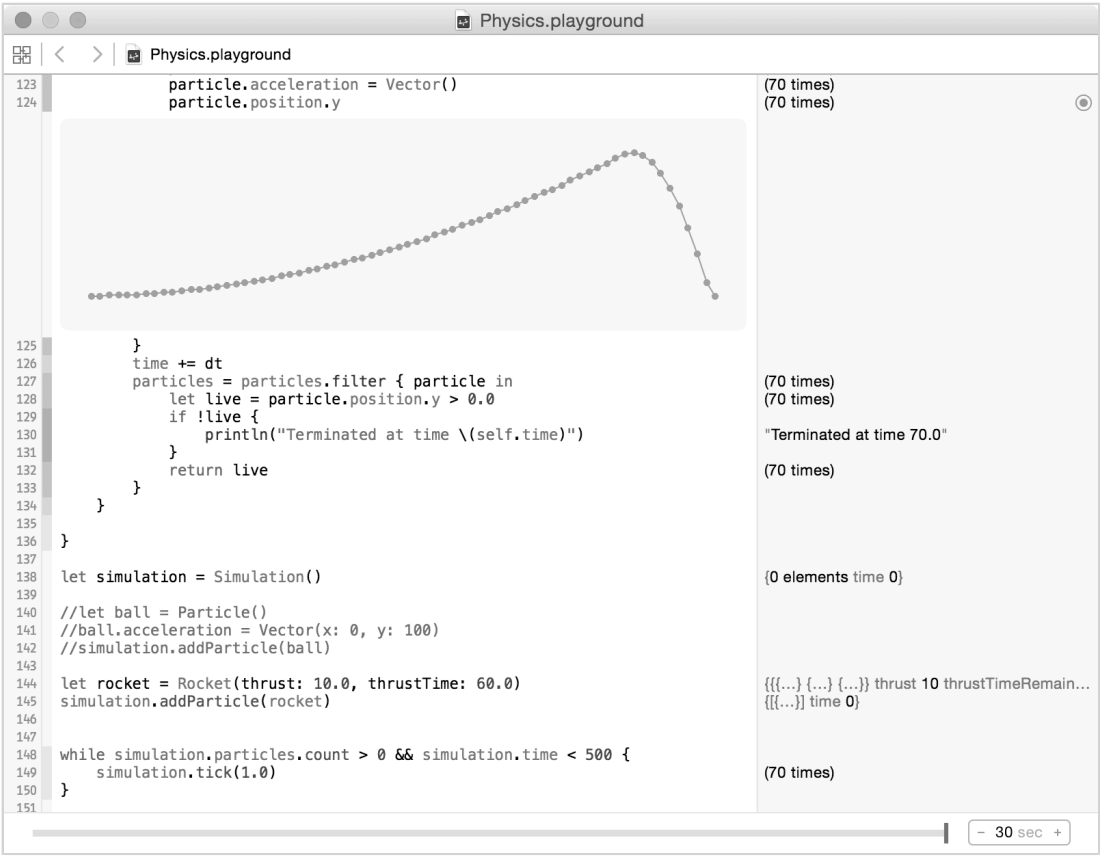
```
let simulation = Simulation()

~~let ball = Particle()~~
~~ball.acceleration = Vector(x: 0, y: 100)~~
~~simulation.addParticle(ball)~~
// let ball = Particle()
// ball.acceleration = Vector(x: 0, y: 100)
// simulation.addParticle(ball)

let rocket = Rocket(thrust: 10.0, thrustTime: 60.0)
simulation.addParticle(rocket)
```

The simulation will run for 70 “seconds” with these parameters. The Value History shows quite a different profile! (Figure 3.2)

Figure 3.2 The rocket’s Y position over time



Note that inheritance is one key differentiator between classes and structures: structures do not support inheritance.

Computed Properties

It is frequently useful to find a vector’s length or magnitude. You could do this by adding a function returning a `Double`:

```
struct Vector {
    ...

    func length() -> Double {
        return sqrt(x*x + y*y)
    }
}
```

However, it is much more natural to think of this as a read-only property. In Swift the general term for this is *computed property*, which is in contrast to the *stored properties* you have been using so far. A read-only computed property version of `length` would look like this:

```
struct Vector {
    ...

    var length: Double {
        get {
            return sqrt(x*x + y*y)
        }
    }
}
```

This read-only computed property pattern (called a "getter") is so common, in fact, that Swift provides a shorthand means of expressing it. Add this to **`Vector`**:

```
struct Vector {
    ...

    var length: Double {
        return sqrt(x*x + y*y)
    }
}
```

At other times it is useful to have a getter *and* setter for a computed property. This tends to be used to alias other properties or to transform a value before it is used elsewhere. For example, you could abstract the setting of the `textField` from the RandomPassword with a computed property:

```
class MainWindowController: NSWindowController {

    @IBOutlet weak var textField: NSTextField!

    var generatedPassword: String {
        set {
            textField.stringValue = newValue
        }
        get {
            return textField.stringValue
        }
    }

    ...

    @IBAction func generatePassword(sender: AnyObject) {
        let length = 8
        generatedPassword = generateRandomString(length)
    }
}
```

Computed properties do not have any storage associated with them. If you need to store a value, you must create a separate stored property for it.

Reference and Value Types

Structures and classes are far more alike in Swift than they are in most languages. However, there is one major difference in how they operate: classes are *reference types*; structures, enums, and tuples are *value types*.

What does it mean to be a value type? For one thing, a value type is always treated as *a single value*, even if it is composed of several individual values via its properties.

In practical terms, this means that when a value type is assigned or passed as a parameter, a copy is made. The following code demonstrates the effect with the **`Vector`** structure:

```
var vector0 = Vector(x: 0, y: 0)        vector0 = {x 0, y 0}
var vector1 = vector0                   vector0 = {x 0, y 0}, vector1 = {x 0, y 0}
vector0.x = 1                           vector0 = {x 1, y 0}, vector1 = {x 0, y 0}
```

When `vector0` is assigned to `vector1`, the entire value of `vector0` is copied into the memory represented by `vector1`. When `vector0` is changed, `vector1` is unaffected.

Contrast this behavior with classes, which, again, are reference types:

```
let ball0 = Particle()        ball0        -> Particle: {x 0, y 0} ...
let ball1 = ball0             ball0, ball1 -> Particle: {x 0, y 0} ...
ball0.particle.x = 1          ball0, ball1 -> Particle: {x 1, y 0} ...
```

Even though you assign `ball0` to `ball1`, there is still only one **`Particle`** instance in existence; no copies are made. The `ball0` constant is a *reference* to the instance, and when `ball0` is assigned to `ball1`, `ball1` is then a reference to the same instance.

(A reference is similar to a pointer in C-based languages. However, a pointer stores the actual memory address of the object and you can access that address directly. A Swift reference does not provide direct access to the address of the object being referenced.)

There is another reference type. Functions are types so that a function can be passed in to other functions as a defined parameter, or even assigned to a property. This is the basis of closures, which you saw briefly earlier in this chapter, and which you will see again in Chapter 15.

Implications of reference and value types

Passing by reference instead of by value has two main implications. The first has to do with mutability. With a value type the code manipulating that value has complete control over it.

Reference types, however, are much different: any part of the software that has a reference to an instance of a reference type can change it. In object oriented programming this can be desirable, but in complex (and especially multithreaded) software it is a liability. An object being changed “behind your back” can cause crashes at best and strange or difficult-to-debug behavior at worst.

Swift constants help further illustrate this point. A constant value type cannot be changed once it is defined, period:

```
let vector: Vector
if someCondition {
    vector = Vector(x: 0, y:  1)
}
else {
    vector = Vector(x: 0, y: -1)
}
vector.x = 1       // Error: Immutable value 'vector' may only be initialized once
```

A constant reference provides no such protection. Only the reference itself is constant.

```
let cannonball = Particle()
cannonball.velocity = Vector(x: 100, y: 5)     // No error!
```

Note that constants within a class or structure are constant. The **`Rocket`** class's `thrust` property, defined with `let` and given an initial value in the initializer, cannot be changed:

```
let rocket = Rocket(thrust: 10.0, thrustTime: 60.0)
rocket.thrust = 3                 // Error: Cannot assign 'thrust' in 'rocket'
```

Choosing between reference and value types

How does a Cocoa programmer decide whether to use a structure or a class for their new type? In order to answer that you will need to know how the type will be used.

The vast majority of Cocoa is built on reference types: subclasses of the Objective-C base class **`NSObject`**, which provides a lot of important functionality for Cocoa. As such, large portions of your app, namely the controller and view layers, will also need to descend from **`NSObject`**.

The model layer is where the answer gets fuzzy. Model-oriented Cocoa technologies such as KVC, KVO, and Bindings also depend on **`NSObject`**, so many app's models will also. For other apps whose models are perhaps more heavy on logic and computation, and less about binding to the UI, you are free to choose the Swift type that makes the most sense for the problem you are trying to solve. Do you want the shared mutable state provided by reference types, or do you prefer the safety of value types? Both have their advantages and costs.

Cocoa, with its deep roots in MVC and Objective-C, will always rely heavily on reference types. In comparison to Objective-C, however, Swift takes great strides in making value types powerful. As a Cocoa programmer, both will be important tools in your belt.

Making Types Printable

If you use a **`Vector`** value in string interpolation, you will not get a very pleasing result:

```
println("Gravity is \(gravity).")            "Gravity is __lldb_expr_247.Vector."
```

You can improve this by conforming to the `Printable` protocol, which looks like this:

```
protocol Printable {
    var description: String { get }
}
```

We will cover protocols in more detail in Chapter 6, but the short version is that a protocol defines a set of properties or methods. In order to conform to a protocol, your type must implement the required properties and methods.

To conform to `Printable`, you must implement a read-only computed property called `description` to return a `String`. Start by declaring that **`Vector`** conforms to `Printable`:

```
~~struct Vector {~~
struct Vector: Printable {
    var x: Double
    var y: Double
```

Finally, implement `description`:

```
struct Vector: Printable {
    ...

    var description: String {
        return "(\(x), \(y))"
    }
}
```

Your **`Vector`**s now look great in strings:

```
println("Gravity is \(gravity).")            "Gravity is (0.0, -9.8)."
```

Swift and Objective-C

Although you will write your classes in Swift, the classes in the Cocoa frameworks are written in Objective-C. Swift was designed to work seamlessly with Objective-C classes. While you can write Cocoa apps in pure Swift, without a line of Objective-C, it is important to have a basic understanding of how Objective-C works.

Objective-C methods (which are only available on classes, not structures) are not called like functions or like Swift methods. Instead of calling a method on an object, Objective-C sends the object a *message*.

A message consists of a receiver, selector, and any parameters. The *selector* is the name of the method you want executed. The *receiver* is the object that you want to execute that method. Here is an example of sending a message in Objective-C:

```
NSString *newString;
newString = [originalString stringByReplacingOccurrencesOfString: @"Mavericks"
                                                       withString: @"Yosemite"];
```

In this example, the receiver is `originalString`, an instance of **`NSString`**, and the selector is `stringByReplacingOccurrencesOfString:withString:`. The parameters are the two **`NSString`** literals.

Note that the selector in the message is “the name of the method.” It is not the method itself or even a reference to it. You can think of a selector as a glorified string.

Objective-C classes know how to receive a message, match the selector with a method of the same name, and execute the method. Or they can do something else with the selector, like forward it in a message to another class. Relying on selectors and message-passing is relatively unique among languages in modern use, and its dynamic nature made the powerful design patterns of Cocoa, and later iOS, possible.

Calling a method is a cut-and-dried process. Either the object implements the method or it does not, and this can be determined at compile time. Passing a message, on the other hand, is dynamic. At runtime, the object is asked, "Does your class implement a method with this name?" If yes, the method with that name is executed. If no, the message is run up the inheritance hierarchy: The superclass is asked, "Do you have a method with this name?" If that class does not have the method, then its superclass is asked, and so on. If the message reaches **`NSObject`** at the top of the hierarchy, and **`NSObject`** says, "No, I do not have a method with that name," then an exception occurs, and your app will halt.

You are developing in Swift, which means that you are not writing message sends in code; you are calling methods. These Swift methods have to be named in such a way that the Swift compiler can turn a method call into a message send when the receiver is an Objective-C object.

If you were to write the above message send in Swift, it would look like this:

```
let newString = originalString.stringByReplacingOccurrencesOfString("Mavericks"
                                                        withString: "Yosemite")
```

Remember, the Swift method has two parameters in the parameter list, but only one has a name. This is why you see methods named in this text and in Apple's documentation with underscores where you expect a parameter name. For example, this method is listed as **`stringByReplacingOccurrencesOfString(_:withString:)`**.

Working with Foundation Types

In Objective-C, a number of familiar types are implemented as classes as part of the Foundation framework: **`NSString`**, **`NSNumber`**, **`NSArray`**, **`NSDictionary`**, and **`NSSet`**. Because Cocoa was built for Objective-C, you will often run into these classes when working with the Cocoa APIs. The good news is that Apple has made transitioning between the Swift and Foundation (Objective-C) counterparts relatively painless. They are *toll-free bridged*, meaning that there is minimal computational cost in converting between the two.

Basic bridging

Swift types are automatically bridged to their Foundation counterparts:

```
let string = "Howdy"
let objcString: NSString = string
```

The reverse is not true, however:

```
let swiftString: String = objcString              // Error!
```

Instead, you must explicitly cast it using `as`:

```
let swiftString: String = objcString as String    // Ok!
```

Another class that you may see is **`NSNumber`**. Because Foundation collections can only store objects, in order to store numbers they must be represented by an object. **`NSNumber`** is the class that Objective-C programmers use for this task. Swift numbers also bridge easily with **`NSNumber`**:

```
let objcNumber: NSNumber = 3
let swiftNumber = objcNumber as Int
```

Bridging with collections

Bridging with collections is similar, but a wrinkle emerges when casting from a Foundation array back to Swift:

```
let array = [1, 2, 4, 8]
let  objcArray: NSArray = array                 // So far so good...
let swiftArray: [Int] = objcArray as [Int]      // Error!
```

You may be surprised to learn that Foundation collections can hold any kind of object – that is, the collection's contents do not have to be of the same type! You will see the Swift type `AnyObject` used with these collections, like this: `[AnyObject]`. (If you are familiar with Objective-C, `AnyObject` has the same meaning as `id`.)

The solutions to this problem are similar to unwrapping optionals: there are safe and unsafe paths. The unsafe path is to use `as!`, the forced cast operator:

```
let swiftArray: [Int] = objcArray as! [Int]
```

As with forced unwrapping, if the type cannot be cast successfully your app will crash. If you are certain that the type is correct, such as when the value is coming from a known API, this is a reasonable assumption to make.

If you are not so certain, you should use the optional type casting operator `as?`, which will evaluate to `nil` if the values cannot be safely cast:

```
if let swiftArray: [Int] = objcArray as? [Int] {
    ...
}
```

This situation is most commonly seen with Cocoa APIs using **`NSDictionary`**: it is typical for the keys to all be **`NSString`**s, but the types of the values commonly differ depending on the key. We will further discuss how to handle these untyped collections safely in Chapter 28.

Suppose you were working with an Objective-C class that supplied a dictionary. When Swift imports the class, it does a basic level of conversion, but it does not know what type the method actually returns, so it is shown as `[NSObject : AnyObject]!`:

```
class NSProcessInfo: NSObject {
    ...
    var environment: [NSObject : AnyObject]! { get }
    ...
}
```

To work with this API you first need to know the actual types contained in the dictionary, which can usually be found in the documentation. You will then need to safely cast the result to Swift types:

```
let processInfo = NSProcessInfo()
if let environment = processInfo.environment as? [String : String] {
    if let path: String = environment["PATH"] {
        println("Path is: \(path)")
    }
}
```

It is important to remember that Swift strings and collections are value types and the Foundation types are all reference types. While Swift's compiler can enforce the constant-ness of an array, with **`NSArray`** the same array object may be referenced by many parts of an application.

The Foundation classes we have discussed so far are all *immutable*, meaning that they cannot be changed – equivalent to being defined with Swift's `let`. Each of them has a mutable subclass: **`NSMutableArray`**, **`NSMutableString`**, and so forth. This has less of an impact on Swift code, but it is important to watch out for if you are working with a significant body of Objective-C code. Because it is a reference type, an instance of **`NSMutableArray`** could be changed by any code that has a reference to it.

Runtime Errors

Despite your best efforts, things will sometimes go wrong while your app is running. In Cocoa, these errors fall into two categories.

Programmer errors are situations that should never happen, which means that they are the result of, well, a mistake you made. (We say they should never happen… but we have made plenty of these.) Examples include not meeting the precondition of a method (the index was not within the array's bounds), performing an illegal operation (such as force-casting incorrectly or force-unwrapping a `nil`), or sending a message to an Objective-C object that does not understand it.

Swift alerts you to programmer errors by trapping, which results in stopping the program. Cocoa APIs use Objective-C exceptions. A trap is typically accompanied by a `fatal error` line in the console, while exceptions have much longer output showing the full stack. Note that Swift does not presently support exceptions.

Recoverable errors, on the other hand, are errors that your application can check for, deal with, and move on from. Examples include being unable to contact a remote server, errors parsing data, or lacking hardware capabilities.

Recoverable errors will be communicated to your code through the return values of methods (such as a `nil` return). For more sophisticated APIs, especially those involving I/O, an **`NSError`** object will be used. You will learn about **`NSError`** in Chapter 12.

You can code defensively and check preconditions in your own code using **`assert()`** and **`fatalError()`**. For example:

```
let condition: Bool = ...
assert(condition, "Condition was not met")
```

`fatalError()` is useful in methods that are declared to return a value. The Swift compiler requires that all code paths return a value – unless you call a *noreturn* function like **`fatalError()`**:

```
func openFortuneCookie() -> String {
    if let cookie = cookie {
        return cookie.fortune
    }
    else {
        fatalError("Must have cookie!")
        // No return statement
    }
}
```

More Exploring of Apple’s Swift Documentation

Your homework for this chapter is to browse through the Classes and Structures, Properties, Methods, and Initialization sections of Apple's *The Swift Programming Language* guide. You should also tackle the challenge exercises given below.

Challenge: Safe Landing

Your investors have pointed out that rockets are expensive and letting them plummet to the ground does not enhance reusability. Enhance the **`Rocket`** class to deploy a parachute in order to slow its descent once it is descending (i.e., shows negative velocity on the Y axis) and reaches a certain altitude.

Challenge: Vector Angle

Your **`Vector`** structure should be able to report its angle in radians. Add a read-only, computed property to it called `angle`.

The angle of a vector can be expressed in Swift as:

```
atan2(y, x)
```

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Z