Creating iOS apps with Xcode

Learn how to develop your own app

Aaron L Bratcher

www.bpbonline.com

First Edition 2024

Copyright © BPB Publications, India

ISBN: 978-93-55516-695

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Dedicated to

My beloved wife
Pamela

About the Author

Aaron L Bratcher has over 20 years of development experience in a variety of programming languages and industries. After working on database applications, he spent a small amount of time writing web applications with Microsoft's .Net. From there he found his niche in iOS programming where he has happily remained for over 10 years. From the day the Swift language was introduced, he strove to make it his primary development language. He has used it to create iOS apps for airlines, home automation, the hotel industry, banking, and more. He has also professionally used SwiftUI in multiple apps. In each organization he pushed for modernizing and simplifying the code. He is currently a Senior iOS Developer at JP Morgan Chase & Co.

About the Reviewers

❖ **Edgar Nzokwe** is a versatile polyglot programmer with expertise across various domains. His proficiency spans frontend technologies like JavaScript, React, and Angular, as well as backend development using Python and Java. Edgar is also well-versed in mobile development, working with frameworks like SwiftUI, UIKit, Java, and Kotlin.

  In the realm of DevOps, Edgar is adept at utilizing tools such as Bitbucket pipelines, Jenkins, Terraform, Docker, Helm, Kubernetes, ArgoCD, and AWS to streamline and automate processes, ensuring efficient application deployment and management.

  Outside of programming, Edgar indulges in his love for reading, delving into books on philosophy, politics, and fiction during his free time. This diverse range of interests enriches his knowledge and fuels his creativity and critical thinking, making him a well-rounded and insightful professional in the tech industry.

❖ **Roman Zakharov** is an experienced QA and Release Manager at Yousician, fostering collaboration with mobile app development teams to ensure smooth user experiences. His expertise lies in testing with Xcode and quality assurance within mobile application development. With an automation-based approach and a dedication to excellence, Roman crafts robust release processes and delivers high-quality apps using Xcode, Unity, JUCE, and other modern tools.

  Additionally, Roman has authored several open-source tools for iOS development, showcasing his commitment to the mobile development community. Beyond his role, he also participates as a speaker in game development and testing conferences across European countries.

  With a passion for ensuring flawless functionality across various platforms, Roman continues to make significant contributions to software engineering and quality assurance. His commitment to staying ahead of industry trends positions him as a trusted expert in the dynamic realm of mobile application testing and development.

Acknowledgement

This book could not have been written without the constant support of my wife, Pamela. It is her continued encouragement that keeps me learning and writing.

I also thank BPB Publications for their guidance and expertise in bringing this book to publication. The editors and reviewers helped make this the best book it could possibly be.

Finally, I thank all those who have taken an interest in this book. It is my genuine desire for anyone who reads this book to learn and grow in the capacity of an iOS developer.

Preface

There’s always room for improvement - this statement is a guiding light to everyone in the software business. It drives the industry forward as new features are added, bugs are fixed, or new ideas explored. It encourages hardware manufacturers to make more things possible than were available before. It sparks the imagination of those who wish to make their own software and make that glimmer of a vision come to reality.

The aim of this book is to help anyone, from the experienced professional moving to iOS development to the daring entrepreneur who has a bold idea on a new app, approach the task of iOS development with more confidence and an understanding of what it takes to make an app.

Starting with an overview of the deceptively easy-to-approach Swift language and moving on to the Apple provided way of displaying items on-screen and interacting with the user with SwiftUI. From there on, several concepts are introduced to give the reader the necessary tools to make an improvement to the world of iOS apps.

Chapter 1: Introduction to Swift – Explores the Swift programming language through the use of a playground, an environment that allows the quick entry of code with immediate feedback. We learn about variables, object types, flow control, and more.

Chapter 2: Learn SwiftUI Basics for Creating a User Interface – After reviewing the basics of the Xcode workspace, we learn how to create a user interface with SwiftUI utilizing several basic generic elements. We also take advantage of nearly instant previews, eliminating the old edit, compile, and run cycle for simple changes.

Chapter 3: Creating Reusable SwiftUI Views – We learn to small views that can be reused throughout the app, giving utility and consistency for both the developer and the end-user.

Chapter 4: Design the Household Chores App – In this chapter we understand the first of the three apps; the household chores app which is used to assign chores to family members. Review the concept and create a project that shows the basic interface and interactions along with the process of localizing text and making an app accessible to all possible users.

Chapter 5: Managing Data and Assets – Building on the previous chapter, we learn about data models and how to organize assets like colors and icons.

Chapter 6: Creating Units of Code that can be Shared – This chapter explores the modularity for the separation of concerns and ease of giving distinct areas of work to different developers is a common developmental goal. It also covers how to create a module that houses specific code that can be integrated into multiple apps.

Chapter 7: Saving Data – Leverage Apple's recently introduced SwiftData to save data to the local device. Also discover how to create user defaults that can be set in the iOS Settings app.

Chapter 8: Charting Your Progress – Learn about Apple's Swift Charts and the variety of ways they can be used. Then update the chores app to chart family member's progress on completing their chores.

Chapter 9: Create the New York City Schools App – The next app allows the user to see a list of schools in New York City along with their associated SAT scores. Create a new project with the appropriate interface and data structures based on the presented concept.

Chapter 10: Testing and Debugging – We learn how to ass unit and UI tests to ensure the app works as expected and continues to function properly as changes are made in the future.

Chapter 11: Networking – We learn how to download data from an internet resource asynchronously and parse it using Apple's built-in libraries.

Chapter 12: Make it Public – We get familiar with the easy way of creating test users and publishing the app to them, and the general public.

Chapter 13: Make a Generic App – We learn how to build an app that can be branded and differentiated across multiple companies. We also use a modular approach to separate the UI and business logic and utilize asset catalogs to manage the distinct personality of each company.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/szq1xtr

The code bundle for the book is also hosted on GitHub at
GitHub Link :- https://github.com/bpbpublications/Creating-iOS-apps-with-Xcode.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. Introduction to Swift 1

Introduction........ 1

Structure........ 1

Objectives 2

Download Xcode 2

Time to play........ 2

Variables........ 3

Collections 5

Optionals 9

Custom types 10

Structs 10

Enums 13

Classes 17

Protocols 22

Closures 24

Flow control 27

Order of operation........ 29

Conclusion........ 30

Exercises........ 30

2. Learn SwiftUI Basics for Creating a User Interface 33

Introduction........ 33

Structure........ 33

Objectives 34

Your first project 34

Interface basics........ 35

Navigators 36

Editing........ 36

Project structure........ 37

Canvas 37

Running your app 38

SwiftUI basics........ 38

The view ... 39

Layout ... 39

Modifiers ... 40

Controls ... 42

Slider ... 42

Toggle ... 43

Stepper ... 44

TextField ... 44

Picker ... 45

DatePicker ... 47

Extended content ... 48

ScrollView ... 49

ForEach ... 49

List ... 51

Drawing ... 55

Circle ... 56

Rectangle ... 56

Rounded rectangle ... 56

UnevenRoundedRectangle ... 57

Capsule ... 58

Ellipse ... 58

Custom shape ... 58

Effects ... 59

SF Symbols ... 62

Conclusion ... 66

Answers ... 66

3. Creating Reusable SwiftUI Views ... 71

Introduction ... 71

Structure ... 71

Objectives ... 71

Cards ... 72

Plain card ... 74

Disclosure card ... 76

Text styles ... 81

Buttons ... 85

Conclusion 90
Answers 90

4. Design the Household Chores App 91
Introduction 91
Structure 91
Objectives 91
User interface design 92
Localization of text 106
Accessibility 109
Conclusion 111

5. Managing Data and Assets 113
Introduction 113
Structure 113
Objectives 114
Data modeling 114
Hashable objects 118
Design patterns 119
Builder 119
Factory 121
Singleton 122
Delegation 122
Step 1: Create data structs 123
Step 2: Create a protocol 127
Step 3: Create a singleton 127
Memory management 128
Step 1: Adjust CommonUI to use the singleton 129
Step 2: Create the delegate class in Household Chores 133
Assets 137
Colors 138
Images 139
Conclusion 140

6. Creating Units of Code that can be Shared 141
Introduction 141
Structure 141

Objectives 141
Source Control 142
Local repository 144
Create a Git repository 144
Branches 144
Tags 147
GitHub 147
Create an SSH Key 148
Create a token 148
Add GitHub account to Xcode 148
Pushing CommonUI to GitHub 148
Swift Package Manager 149
Remote packages 150
Local packages 152
Conclusion 152

7. Saving Data 153
Introduction 153
Structure 153
Objectives 153
SwiftData 154
Deleting data 158
Assigning chores 159
Visual cleanup 161
Fix previews 163
Daily chores 164
User preferences 166
Integrating user preferences 168
Conclusion 171

8. Charting your Progress 173
Introduction 173
Structure 173
Objectives 173
Explore Swift Charts 174
Bar charts 174

Line graphs ... *176*

Charting different sets ... *179*

Proportional charts ... *181*

Integrating a chart into the Chores app ... 183

Conclusion ... 191

9. Create the New York City Schools App ... 193

Introduction ... 193

Structure ... 193

Objectives ... 193

Interface design ... 194

Model design ... 197

Scores ... *208*

Search bar ... *212*

Conclusion ... 215

10. Testing and Debugging ... 217

Introduction ... 217

Structure ... 217

Objectives ... 218

Tests ... 218

UI tests ... 218

Unit tests ... 223

Breakpoints ... 229

Debugger commands ... 232

Logging ... 232

Conclusion ... 233

11. Networking ... 235

Introduction ... 235

Structure ... 235

Objectives ... 236

Codable models ... 236

Decoding ... *236*

Downloading data ... 240

Generics ... 242

Async / await ... 242

Error handling.. 246
Cleanup.. 247
Testing 249
Conclusion.. 250

12. Make it Public.. 251
Introduction.. 251
Structure.. 251
Objectives 252
App icon 252
Using physical devices 252
No developer accounts.. 254
With a developer account 255
App Store Connect 255
Preparing the app for upload.. 255
Test users.. 258
Submitting to the App Store 260
Conclusion.. 260

13. Make a Generic App.. 261
Introduction.. 261
Structure.. 261
Objectives 262
Designing the app 262
App structure.. 263
Create target application 263
Defining customizable options.. 265
Card colors.. 265
Countdown timer 266
Toppings 267
Crust.. 268
Text 268
Bundles 270
Create the common interface elements 271
Card.. 272
Progress bars 275

Add button 277
Text style 278
Create the primary app 281
Data model 281
Order view 283
App utility 284
Order list 286
Foundation extensions 288
Add order sheet 288
Localization 290
Running the app 291
The other app 292
Conclusion 292

Index 293-296

Chapter 1
Introduction to Swift

Introduction

In 2014, at the Worldwide Developers Conference, Apple introduced the world to the new Swift programming language with great fanfare. Since then, the language has been adopted as the premier language for creating apps not only for Apple products, but for servers and other platforms too. The future and direction of this language is publicly presented and debated. See **https://swift.org** for more information.

Structure

This chapter contains the following topics:

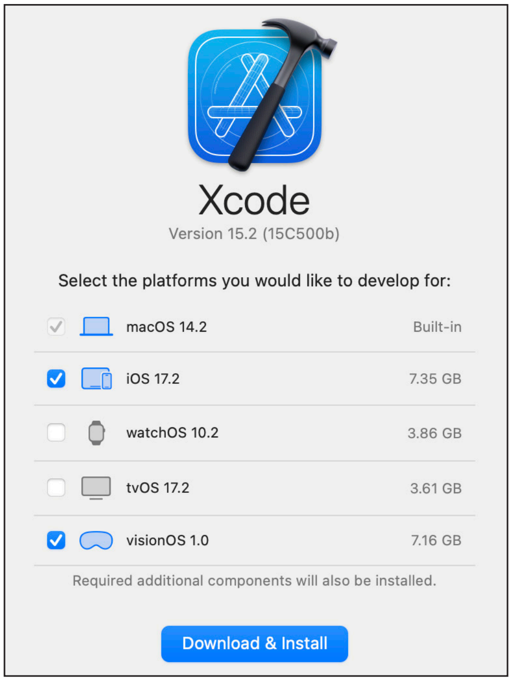
- Download Xcode
- Time to play
- Variables
- Custom types
- Closures
- Flow control
- Order of operation

Objectives

By the end of this chapter, the reader should know the basic syntax of the Swift language, some commonly used types, how to create custom types, and how to manage the flow of how code runs.

Download Xcode

The App Store is the best place to securely download and install applications to your computer. After installation, launch the app. Because Xcode is used to develop apps for all Apple platforms, an initial set of devices to develop for must be specified. Make sure iOS is selected and click the **Download & Install** button, as shown in *Figure 1.1* below:



Xcode
Version 15.2 (15C500b)

Select the platforms you would like to develop for:

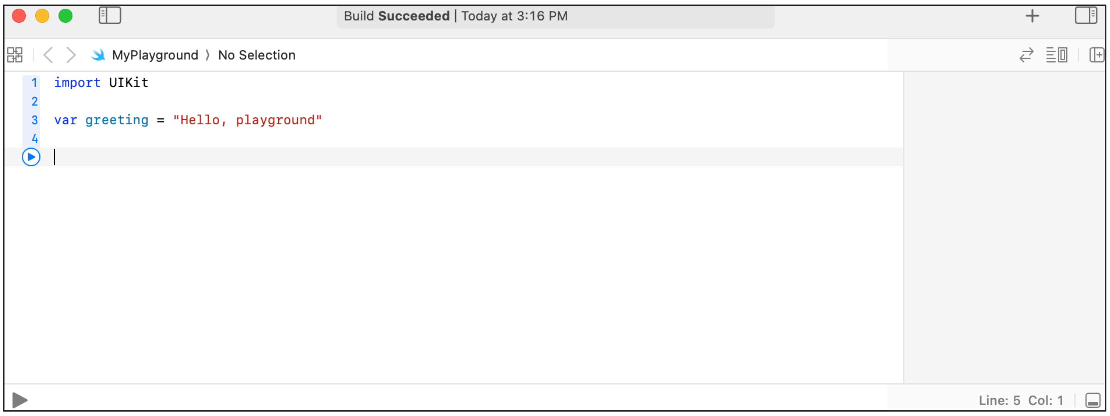
✓	macOS 14.2	Built-in
✓	iOS 17.2	7.35 GB
	watchOS 10.2	3.86 GB
	tvOS 17.2	3.61 GB
✓	visionOS 1.0	7.16 GB

Required additional components will also be installed.

Figure 1.1: *Platform selection dialogue*

Time to play

From the **File** menu, select **New** and then **Playground**. Select the **iOS** template at the top and the **Blank** Playground, then click the **Next** button on the bottom. Specify the location to save the playground and, if desired, change the name. Then click the **Create** button. In moments a screen that looks like *Figure 1.2* will be shown:


Build Succeeded | Today at 3:16 PM
MyPlayground 〉 No Selection
Line: 5 Col: 1

```
import UIKit

var greeting = "Hello, playground"

```

Figure 1.2: *New Blank Playground*

The playground is a great way to try some quick code without creating a full project. The bulk of the display is the editor. To the right, is the **Live View**. In the editor, two lines of code have been provided. For now, we will ignore the import statement at the top.

Below the import we see: **`var greeting = "Hello, playground"`**

Variables

A significant goal of programming is to manipulate the environment on the device or react to a change in the environment. To achieve this, the app needs to store and manipulate data using values and types. For instance, the value **"`Hello, playground`"** shown above is a string type. The **`var`** keyword declares a mutable variable called **`greeting`**. When an object is created and assigned to a variable, it is said to be instantiated.

On the bottom-left of the window is the execute button. ▶ Click and hold it to see a pop-up menu. Select **Automatically Run** and the screen will change. In the live view, to the right, the same string that was assigned, **"`Hello, playground`"** is displayed with a small square next to it.

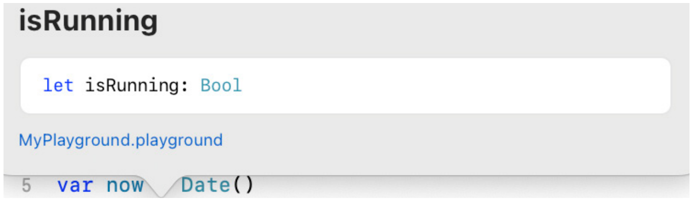
▣ - this square is the results toggle button. Clicking it will toggle the results display in the editor.

On a blank line at the bottom, enter the following lines of code:

```
greeting = "Swift is easy"

var now = Date()

let isRunning = true

let maxLength = 256

let avgNumberOfChildren = 2.2
```

It may be necessary to disable the automatic running of code while typing this in. After all the lines of code are entered, make sure to re-enable **Automatically Run** or click on the **Run** button to the left of the last line of code. ▶ When entering a lot of code, the second option may be best. In the live view, values that are assigned to the variables will be displayed. Notice how the greeting shows two values? The initial value and then the new one. If you hover the pointer over any of the values, the line of code responsible for it will be highlighted in the editor. On the right side of the value is a preview icon. Click the results toggle or the preview icon. For now, a repeat of the value is shown. As Swift is explored, this will give more information.

In the code entered, the value type was not specified. Swift will implicitly assign the type based on the value initially given to it. To see the type, hold down the option key (⌥) and click on the variable name to see a more formal declaration as shown in *Figure 1.3*. Explicitly specifying the type will be shown later in the chapter.



isRunning

```
let isRunning: Bool
```

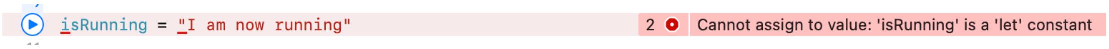
MyPlayground.playground

```
var now  Date()
```

Figure 1.3*: Variable definition popover*

The keyword let was used instead of `var` for three of the lines above. What does this mean? On a blank line, at the bottom of the variable declarations, try adding a line to change the value of `isRunning` to `I am now running`.

What happens? *Figure 1.4* illustrates the error:



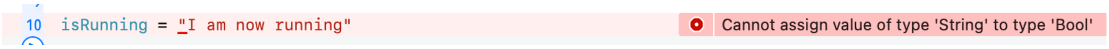
```
isRunning = "I am now running"
```

2 Cannot assign to value: 'isRunning' is a 'let' constant

Figure 1.4*: (Immutable variable error)*

A variable defined using the let syntax makes it immutable, or constant, after a value has been assigned. When a variable's value does not need to change, use let. To allow isRunning's value to change, go to the declaration and change it to `var`.

What happens? *Figure 1.5* shows an additional error:



```
isRunning = "I am now running"
```

Cannot assign value of type 'String' to type 'Bool'

Figure 1.5*: (Variable type error)*

Some languages allow the value type of a variable to be changed after it has been set. However, Swift is a strongly typed language, which means the value type cannot change. The variable `isRunning` is a Bool type so it can only hold the values of true or false. Remove the errant line of code.

When the first error was displayed, a debug area was shown to give additional information. To show or hide this area, click the **Debug** area toggle button below the Live View panel.

The names used for the variables are arbitrary. Swift convention says variable names start with a lowercase letter and an uppercase for concatenated words in the variable name. This is called camel case. In Swift, variable names can be any combination of letters, numbers, and even emojis. Names have these primary restrictions: They cannot begin with a number and must not contain mathematical symbols or white space characters, that is, spaces, tabs, newlines, and so on. Try it now. Change the names of the variables and introduce numbers in the middle or at the end of the names or try an emoji character. What works and what does not?

So far, we have been assigning values to variables at the time of declaration. While this may work in limited cases, the value is usually not known until a later time. On a blank link, enter these lines of code and run by either setting the playground to run automatically or clicking the **Run** button to the left of the last line of code:

```
let maxToppings: Int

var useBox: Bool

maxToppings = 10

useBox = false
```

When a value is not immediately assigned to the variable, the type must be specified at the time of declaration. But wait, there is more.

Collections

When deciding on the name of a variable, it is important to describe the type of value and what the value represents. For instance, if the name **`maxLength`** were changed to **`dogNames`** how would your expectations of the value change?

The first expectation is for the variable to hold String values. The second is to hold multiple values because of the plurality in the name. When a variable holds multiple values, it is called a collection. The three principal collection types in Swift are array, dictionary and set. Type in the following line of code on a blank line and run:

```
var dogNames = ["Daisy", "Caesar", "Luna"]

var dogAges = [2, 5,  7]
```

See how the Live View shows array of 3 elements?

Now, click on the results toggle to see the values. Try different values inside the square brackets. Notice how there is always a comma between each value and how the types must match. Use Boolean, string, integer, or floating-point numbers. Some languages would call these primitive types, however in Swift they are treated just like any other object type we will learn about later.

There are different ways of accessing the data in an array. The easiest is to use the ordinal position. On the line after the **`dogAges`** variable, type in: **`let firstDog = dogNames[0]`**. Notice that array index starts at zero.

Like any other variable, a collection can have values assigned after the declaration. Remove the values between the square brackets and watch what happens. An error is displayed. Swift does not know what type of values the variable will store. It needs to be specified either by the developer or by the compiler. Here, the compiler does not know what type to assign so you need to.

Change the line of code to look like this:

```
var dogNames: [String] = []
```

Now we have an empty array for strings. Let us explore different ways of assigning and removing values to this variable. Type in these lines of code and run:

```
dogNames = ["Daisy", "Caesar", "Luna"]

dogNames.append("Ralph")

dogNames.insert("Sparky", at: 0)

dogNames.remove(at: 1)

let sortedNames = dogNames.sorted()
```

Before looking at the results, what do you think each line of code does? The first and last lines are probably obvious, but what about the others? After giving it some thought, click on the results toggle button next to each entry in the Live View. Are the results what you expected?

Something that may be surprising is the value shown for the **`remove`** statement because it shows the value affected by the statement instead of the changed variable. The returned value is being ignored. It can be captured by changing the line to something like this: **`let removedDog = dogNames.remove(at: 1).`** You may have been surprised that **`Daisy`** was removed and not **`Ralph`**. Remember, arrays start at index zero.

In the code above for inserting, change the **`0`** to **`5`** and see what happens. We will explore debugging in a later chapter.

The next collection to discuss is the dictionary. It uses a unique identifier, or key, and associates it with a value. On a blank line, enter the following lines of code and run:

```
let stations = [2: "Shell", 16: "BP", 32: "Chevron"]

print(stations[2])
```

Like arrays, each entry is separated by a comma. Each entry consists of the key, a colon, then the value. The type for all provided values must match. If you are wondering why the **`print`** statement said **`Optional`**, then you are in for a treat. That will be discussed soon. To get a preview, try changing the **`2`** to **`5`**.

The type used for this dictionary's key is **Int**, however **AnyHashable** type can be used. In fact, when declaring the dictionary variable, **AnyHashable** can be specified as the type. Change the let statement to the following and run:

```
var stations: [AnyHashable: String]

stations = [2:"Shell", "16":"BP", 32:"Chevron"]
```

The declaration specifies the key type and the value type. Later, we will look at using other types for the values. Notice how the keys used on the second line are now **Int** and **String**. Print the **BP** value by changing the key used in the **print** statement to **16**. In Swift, **Int** and **String** are **hashable** types. As it is possible to make custom types **hashable**, they can also serve as keys, if desired. Most often **Int** and **String** are used as the key type.

Like arrays, dictionaries are not always populated at declaration and instantiation. Enter the following lines of code and run:

```
1. stations[55] = "Phillips 66"
2. stations[55] = "PetroChina"
3. stations.removeValue(forKey: 2)
4. let stationCount = stations.count
5. print(stations.keys)
6. print(stations.values)
7. stations = [:]
```

There is a lot happening here, so let us dissect it one line at a time. The first line assigns the value **Phillips 66** to the dictionary with the key **55**. The second line assigns the value **PetroChina** to the dictionary with the key **55**. However, we must wonder, how is that that possible? Keys are supposed to be unique.

The second line replaces the value associated with the key **55**. This is confirmed with the last **print** statement. Line three completely removes the key, **2**, and the value associated with it.

The fourth line assigns the number of dictionary entries to the variable **stationCount**. All collections have the `count` property. In Swift, properties are variables associated with an object. Other helpful properties for collections are: **isEmpty**, **first**, and **last**.

The **print** statements on lines five and six above print out the keys and values properties respectively. Note that they are arrays. The keys and values printed may or may not be in the order entered because, unlike an array, a dictionary is an unordered collection. The last statement instantiates an empty dictionary and assigns it to the variable.

The final collection type we will cover is the **Set**. With the dictionary, any object type that is **hashable** can be used as the key that is associated with some data. What if we do not have any associated data? A situation where the key is the data, and we need a unique collection of these objects. This is the perfect time to use a **Set**. Let us try it.

Type in the following lines and run:

```
var partyGuests: Set<String> = []
partyGuests.insert("Aaron L Bratcher")
partyGuests.insert("Anne McCaffrey")
partyGuests.insert("Douglas Adams")
partyGuests.insert("Richard Castle")
print(partyGuests)
partyGuests.remove("Douglas Adams")
print(partyGuests)
let insertResults = partyGuests.insert("Richard Castle")
print(insertResults)
```

To limit the output in the Debug area, remove all code entered before or create a new playground. Again, we see something new in the declaration and initialization of the variable. A **`Set`** can hold any **`hashable`** type. Like the dictionary, the type must be specified. The type to be stored, in this case a **`String`**, is specified in the angle brackets. Set utilizes generics, a feature covered later in this book. The square brackets initialize the variable with an empty set.

As a **`Set`** is not ordered, the concept of appending is not logical. Instead, an object is inserted into the set and conversely, a known object can be removed. When an object is inserted, the response can be ignored, as was shown the first several times or captured. One could initially think the returned value of an insert would be a **`Bool`** type to indicate the success of the insert, however upon inspection you will find something new. Option-click on the variable and you will see this definition:

```
let insertResults: (inserted: Bool, memberAfterInsert: String)
```

The parenthesis around two element definitions is a Tuple. Tuples can have one or multiple elements with, or without, a name. If a name is included, each element is like a variable definition. Here are two examples:

Example 1:

```
    let someData: (String, String, String)

    someData = (“first element”, “second object”, “third”)
```

Example 2:

```
    var moreData: (name: String, city: String)
    moreData = ("Thomas", "New York City")
    moreData = (name: "Roger", city: "Los Angeles")
```

It can be easy to forget what each value represents without a name, so names are recommended. However, when a name is not available, the ordinal index can be used. Like arrays, the index of tuple elements is zero based.

```
1. print(someData.1)
2. print(moreData.0)
3. print(moreData.city)
```

Optionals

An optional starts with a question. Does a value exist? Remember when stations **`[2]`** was printed earlier? The output was **`Optional("Shell")\n`** and when stations **`[5]`** was printed, the output was **`nil\n`**. Why? Because when an attempt to retrieve a value from a dictionary is performed, the key or the value associated with it may not exist. It is only appropriate that a question mark is used to define when a variable or property is optional. Here is a declaration of a variable/property that may or may not have a **`String`** value:

```
let crust: String?
```

An attempt to print the value of crust, will generate an error. An immutable optional needs to be initialized. In this case either with a string or nil. Nil indicates no value exists. If the let was to be changed to a **`var`**, then it would compile and run properly as it starts with nil assigned.

To properly retrieve the value, the optional must be unwrapped or coalesced to a non-optional value. Here are a few accepted ways of doing this:

```
let crust: String?

crust = "thin"

if let crust {

   print(crust)

}

if var crust {

   crust = "altered crust"

   print(crust)

}

let crustType = crust ?? ""
```

The first option is to use the **`if let`** syntax. This checks if a value exists and, if one does, assigns it to a variable of the same name. This new variable is limited to the scope of the if statement, that is, between the curly braces. The variable can now be used safely. Outside of the scope, the original optional variable still exists.

The second option uses the **`if var`** syntax. It is like the first option, but it gives a mutable variable that can be used within the scope of the if statement. Also, while the value of the new variable may be changed within the scope, it does not affect the original variable or property.

The last option shown here is to coalesce. When the original variable has no value, the option after the coalesce operator is used instead. Multiple coalesce operators can be used in the same expression to chain multiple **`optionals`** together. Add a print statement to see what the value of **`crustType`** is. There is also the use of a **`guard`** statement, but it will be shown in a later segment because of how it is used.

Something not shown, because it is frowned on, however sometimes it is needed, is the force-unwrap operator. This will attempt to unwrap the variable and assign the value. However, if it fails, the app will crash. Remove the assignment of **`thin`** to the variable and then run the statement below and an error will appear.

```
let crustType = crust!
```

A lot has been covered for how a variable is defined and used with some built-in types. We have also looked at collections and optionals. These basics are used in all aspects of Swift development and needs to be completely understood. Try to do the following on your own (answers are at the end of the chapter):

- Create a variable that holds an array of optional strings.
- Create a variable that holds a dictionary with an integer key and an optional tuple of two strings.

Custom types

When designing an app, it is a common task to structure the data into logical units for storage and manipulation. In this section, the most common ways of creating custom types will be explored.

Structs

A struct is a way to organize naturally related data elements together and, optionally, provide ways of manipulating them. They are very efficient for this task. For instance, a pizza representation may have a struct like this:

```
struct Pizza {
   let crust: String?
   Let sauce: String
   let toppings: [String]
}
```

The key word **`struct`** specifies what kind of custom type we are creating. Notice the name, **`Pizza`**, is capitalized. When creating a new type, Swift convention is to capitalize the name of the type. When creating an instance object of the type, the variable or property starts with a lowercase letter.

Some properties were created to describe a typical pizza. What kind of crust? Thin, hand-tossed, cheese filled, or, maybe, none. Some sauce and toppings.

Like a string or dictionary, **`Pizza`** is now a type that can be created and assigned to a variable or property. The struct elements are properties that must be initialized with values. Swift helps us out by automatically creating an initializer for the struct so we can simply do this to instantiate the object:

```
let pizza = Pizza(crust: nil, sauce: "tomato", toppings: ["onion", "green
pepper", "mushroom", "black olive"])
print(pizza.toppings)
```

This creates a **`Pizza`** object and assigns it to our variable, **`pizza`**. This could be a pizza bowl, where no crust exists. Like many programming languages, Swift is case-sensitive. It is very common to have a variable name that closely matches the type name.

There are times a coded initializer is needed. An initializer provides a point where parameters may be passed to make a valid object. It consists of the keyword, **`init`**, followed by an open parenthesis, maybe some parameters, then the closing parenthesis.

The first case, shown below, is a one-to-one mapping of the **`init`** parameters to the properties that need a value. Swift implicitly provides this case for simple structs. The parameters provided to the **`init`** can also have default values assigned. Add the **`init`** code so the struct looks like this:

```
struct Pizza {
   let crust: String?
   let sauce: String
   let toppings: [String]
   init(crust: String? = nil, sauce: String = "tomato",
toppings: [String]) {
      self.crust = crust
      self.sauce = sauce
      self.toppings = toppings
   }
}
```

What is the first thing you notice? That the parameter declarations look remarkably like the variable work done earlier. The first parameter, **`crust`**, is an optional string. A default value of nil is given. This would be helpful for pizza bowls where a crust does not exist. The default value for sauce is **`tomato`** and toppings does not have a default value so it must be specified when creating the object. Using this initializer, the creation of the **`pizza`** bowl can now look like this:

```
let pizza = Pizza(toppings: ["onion", "green pepper", "mushroom",
"black olive"])
```

The default values are for convenience but do not keep the developer from using the parameters. Any or all the parameters may be used for the instantiation of the **pizza**:

```
let pizza = Pizza(crust: "thin", sauce: "tomato",
toppings: ["onion", "green pepper", "mushroom", "black olive"])
```

The second thing to be noticed, may be the use of self. This is a reference to the one specific object and the items inside it. Remember when the dictionary keys were printed? The code referenced the keys property of the single dictionary object. The initializer is assigning values to the properties of self. When there is a need to reference something within the object itself, self is used when there is ambiguity. Otherwise, it is implied, (there is another use for it that will be looked at later in the book). For instance, what if the first parameter in **init** had been called base instead of crust? If that were done, then base could have been assigned to the **crust** property without use of the **self** designation.

Immutable properties on this struct were used. We can also use mutable properties that have a default value, so they are changeable from outside the struct. When all the properties have a default value, a second **init** without parameters can be created. Adjust your code like so:

```
struct Pizza {
   var crust: String?
   var sauce: String = "tomato"
   var toppings: [String] = []
   init(crust: String? = nil, sauce: String = "tomato",
toppings: [String]) {
      self.crust = crust
      self.sauce = sauce
      self.toppings = toppings
   }
   init() { }
}
let pizza = Pizza()
pizza.toppings = ["onion", "green pepper", "mushroom", "black olive"]
print(pizza.sauce)
```

When using a mutable, optional property, the default value is nil, so it does not need to be assigned. The second **init** function has an empty body. It is provided so an instance of **Pizza** can be created without using any parameters. Now a **pizza** can be instantiated using the **init** with parameters or the one without. Swift will determine, automatically, which one to use at the point of instantiation. When you make these changes and attempt to run, you will get an error. Do you know why? The **pizza** variable is an immutable struct. Changes cannot be made to it. Change the let to a **var** on line sixteen and the problem will be fixed. There is so much more to structs that we will look at later. For now, it is time to fix another problem.

Enums

The **Pizza** struct uses **String** properties. However, each one has a limited number of options. When the is a known, fixed, number of options, it is best to enumerate them. Clear your playground of all code, except the import at the top, and add this:

```
enum Crust {
   case thin
   case handTossed
   case deepDish
   case none
}
enum Sauce {
   case tomato, white, arrabbiata, none
}
let crust = Crust.handTossed
let sauce: Sauce
sauce = .tomato
```

Like a struct, you start with your keyword **enum**, then the name of your custom type. Again, the name of the type is capitalized. See how each name assigned is singular, not plural. When creating a variable, you can use implicit type allocation if you give a value immediately or assign the type and give a value later. Notice how the full enum name **Sauce.tomato** is not required when assigning the value to the sauce variable? Swift is contextually aware of the **Sauce** type so only the specific enumeration is necessary.

It was mentioned above that a pizza may not have a crust. While an optional **enum** is possible, it is best to have an enumeration when possible. Below the **Crust** and **Sauce** enums, enter these lines of code and run:

```
enum Size: Int {
   case six = 6
   case twelve = 12
   case eighteen = 18
}
```