# Swift Programming
## the big nerd ranch guide

# Table of Contents

# Introduction

## Learning Swift

Apple's Worldwide Developers Conference is an annual landmark event for its developer community. It is a big deal every year, but 2014 was particularly special because Apple introduced Swift, an entirely new language for the development of iOS and OS X (now called macOS) applications.

As a relatively new language, Swift represents a fairly dramatic shift for macOS and iOS developers. More experienced iOS developers have something new to learn, and new developers cannot rely on a venerable community for tried and true answers and patterns. Naturally, this shift creates some uncertainty.

But this is also an exciting time to be a macOS and iOS developer. There is a lot to learn in a new language, and this is especially true for Swift. The language has evolved quite a bit since its beta release in the summer of 2014, and it continues to evolve.

We are all at the forefront of this language's development. As new features are added to Swift, its users can collaboratively determine its best practices. You can directly contribute to this conversation, and your work with this book will start you on your way to becoming a contributing member of the Swift community.

## Why Swift?

If you have experience using Objective-C to develop for Apple platforms, you may be wondering why Apple released a new language. After all, developers had been producing high-quality apps for OS X and iOS for years. Apple had a few things in mind.

First, Objective-C is an older language. And while this is not always a problem, it leads to some difficulty in this case. The syntax of Objective-C was solidified prior to the rise of prominent scripting languages in the 1990s that popularized more streamlined and elegant syntax (e.g., JavaScript, Python, PHP, Ruby, and others). This makes Objective-C feel strange to most developers when they get started, so its syntax can be an impediment to developer productivity. Additionally, as an older language, Objective-C is missing many advancements developers in modern languages currently enjoy.

Also, Swift aims to be safe. Objective-C did not aim to be unsafe, but things have changed quite a bit since Objective-C was released in the 1980s. For example, the Swift compiler aims to minimize undefined behavior, which is intended to save the developer time debugging code that failed during the runtime of an application.

Another goal of Swift is to be a suitable replacement for the C family of languages (C, C++, and Objective-C). That means Swift has to be fast. Indeed, Swift's performance is comparable to these languages in most cases.

Swift gives you safety and performance all in a clean, modern syntax. The language is quite expressive; developers can write code that feels natural. This feature makes Swift a joy to write and easy to read, which makes it great for collaborating on larger projects.

Last, Apple wants Swift to be a general purpose programming language. This desire was reflected by the decision to open source Swift in December 2015. Open sourcing the language not only invites developer involvement to help the language progress, but also makes it easier for developers to port the language to other systems beyond macOS and iOS. Apple hopes that developers will use Swift to write apps for a variety of mobile and desktop platforms and to develop back-end web applications as well. Swift is intended to become a mainstream programming language that is the best solution for writing a variety of applications on a variety of systems.

## Whither Objective-C?

So, what about Objective-C, Apple's previous *lingua franca* for its platforms? Do you still need to know that language? For professional macOS and iOS developers, we think that answer is an unequivocal "yes." Apple's Cocoa and UIKit libraries, which you will use extensively, are written in Objective-C, so debugging will be easier if you understand that language. Indeed, Apple has made it easy – and sometimes preferable – to mix and match

Objective-C with Swift in the same project. As a macOS or iOS developer, you are bound to encounter Objective-C, so it makes sense to be familiar with the language.

But do you need to know Objective-C to learn Swift or to begin to develop macOS or iOS apps? Not at all. Swift coexists and interoperates with Objective-C, but it is its own language. If you do not know Objective-C, it will not hinder you in learning Swift. (We will only use Objective-C directly in one chapter toward the end of this book, and even then it will not be important for you to understand the language.)

## Prerequisites

We have written this book for all types of macOS and iOS developers, from platform experts to first-timers. For readers just starting software development, we will highlight and implement best practices for Swift and programming in general. Our strategy is to teach you the fundamentals of programming while learning Swift. For more experienced developers, we believe this book will serve as a helpful introduction to your platform's new language. So while having some development experience will be helpful, we do not believe that it is necessary to have a good experience with this book.

We have also written this book with numerous examples so that you can refer to it in the future. Instead of focusing on abstract concepts and theory, we have written in favor of the practical. Our approach favors using concrete examples to unpack the more difficult ideas and also to expose the best practices that make code more fun to write, more readable, and easier to maintain.

## How This Book Is Organized

This book is organized in six parts. Each is designed to accomplish a specific set of goals that build on each other. By the end of the book, you will have built your knowledge of Swift from that of a beginner to a more advanced developer.

| | |
|---|---|
| *Getting Started* | This part of the book focuses on the tools that you will need to write Swift code and introduces Swift's syntax. |
| *The Basics* | *The Basics* introduces the fundamental data types that you will use every day as a Swift developer. This part of the book also covers Swift's *control flow* features that will help you to control the order in which your code executes. |
| *Collections and Functions* | You will often want to gather related data in your application. Once you do, you will want to operate on that data. This part of the book covers the *collections* and *functions* Swift offers to help with these tasks. |
| *Enumerations, Structures, and Classes* | This part of the book covers how you will model your data in your own development. We discuss the differences between Swift's *enumerations*, *structures*, and *classes* and make some recommendations on when to use each. |
| *Advanced Swift* | As a modern language, Swift provides a number of more advanced features that enable you to write elegant, readable, and effective code. This part of the book discusses how to use these elements of Swift to write idiomatic code that will set you apart from more casual Swift developers. |
| *Event-Driven Applications* | This part of the book walks you through writing your first macOS and iOS applications. For readers working with older OS X or iOS applications, we conclude this part of the book by discussing how to interoperate between Objective-C and Swift. |

## How to Use This Book

Programming can be tough, and this book is here to make it easier. To get the most out of it, we recommend you follow these steps:

- Read the book. Really! Do not just browse it nightly before going to bed.

- Type out the examples as you read along. Part of learning is muscle memory. If your fingers know where to go and what to type without too much thought on your part, then you are on your way to becoming a more effective developer.

- Make mistakes! In our experience, the best way to learn how things work is to first figure out what makes them not work. Break our code examples and then make them work again.

- Experiment as your imagination sees fit. Whether that means tinkering with the code you find in the book or going off in your own direction, the sooner you start solving your own problems with Swift, the sooner you will become a better developer.

- Do the challenges we have provided. As we mentioned, it is important to begin solving problems with Swift as soon as possible. Doing so will help you to start thinking like a developer.

More experienced developers may not need to go through some of the earlier parts of the book. *Getting Started* and *The Basics* may be very familiar to some developers.

One caveat: In *The Basics*, do not skip the chapter on optionals as they are at the heart of Swift, and in many ways they define what is unique about the language.

Subsequent chapters like Arrays, Dictionaries, Functions, Enumerations, and Structs and Classes may seem like they will not present anything new to the practiced developer, but we feel that Swift's approach to these topics is unique enough that every reader should at least skim these chapters.

Last, remember that learning new things takes time. Dedicate some time to going through this book when you are able to avoid distractions. You will get more out of the text if you can give it your undivided attention.

## Challenges

Many of the chapters conclude with exercises for you to work through on your own. These are opportunities for you to challenge yourself. In our experience, truly deep learning is accomplished when you solve problems in your own way.

## For the More Curious

Relatedly, we include "For the More Curious" sections at the end of many chapters. These sections address questions that may have occurred to the curious reader working through the chapter. Sometimes, we discuss how a given language feature's underlying mechanics work, or we may explore a programming concept not quite related to the heart of the chapter.

## Typographical Conventions

You will be writing a lot of code as you work through this book. To make things easier, we use a couple of conventions to identify what code is old, what should be added, and what should be removed. For example, in the function implementation below, you are deleting `print("Hello")` and adding `print("Goodbye")`. The line reading `func talkToMe() {` and the final brace `}` were already in the code. They are shown to help you locate the changes.

```
func talkToMe() {
    print("Hello")
    print("Goodbye")
}
```

## Necessary Hardware and Software

To build and run the applications in this book, you will need a Mac running macOS El Capitan (10.11.4) or newer. You will also need to install Xcode, Apple's *integrated development environment* (IDE), which is available on the App Store. Xcode includes the Swift compiler as well as other development tools you will use throughout the book.

Swift is still under rapid development. This book is written for Swift 3.0 and Xcode 8.0. Many of the examples will not work as written with older versions of Xcode. If you are using a newer version of Xcode, there may have been changes in the language that will cause some examples to fail.

As this book is moving into the printing process, Xcode 8.1 Beta is available. The code samples in the book work with the latest beta version we have been able to use. If future versions of Xcode do cause problems, take heart – the vast majority of what you learn will continue to be applicable to future versions of Swift even though there may be changes in syntax or names. You can also check out our forums at `forums.bignerdranch.com` for help.

## Before We Begin

We hope to show you how much fun it can be to make applications for the Apple ecosystem. While writing code can be extremely frustrating, it can also be gratifying. There is something magical and exhilarating about solving a problem, not to mention the special joy that comes from making an app that helps people and brings them happiness.

The best way to improve at anything is with practice. If you want to be a developer, then let's get started! If you find that you do not think you are very good at it, who cares? Keep at it and we are sure that you will surprise yourself. Your next steps lie ahead. Onward!

# Part I
## Getting Started

This part of the book introduces the toolchain for writing Swift code. It introduces Xcode as the Swift developer's primary development tool and uses playgrounds to provide a lightweight environment for trying out code. These initial chapters will also help you become familiar with some of Swift's most basic concepts, like constants and variables, which will set the stage for the rest of the book and a deeper understanding of the language.

# 1
# Getting Started

In this chapter, you will set up your environment and take a small tour of some of the tools you will use every day as an iOS and macOS developer. Additionally, you will get your hands dirty with some code to help you get better acquainted with Swift and Xcode.

## Getting Started with Xcode

If you have not already done so, download and install Xcode, available on the App Store. Make sure to download Xcode 8 or higher.

When you have Xcode installed, launch it. The welcome screen gives you several options, including Get started with a playground and Create a new Xcode project (Figure 1.1).

Figure 1.1  Welcome to Xcode
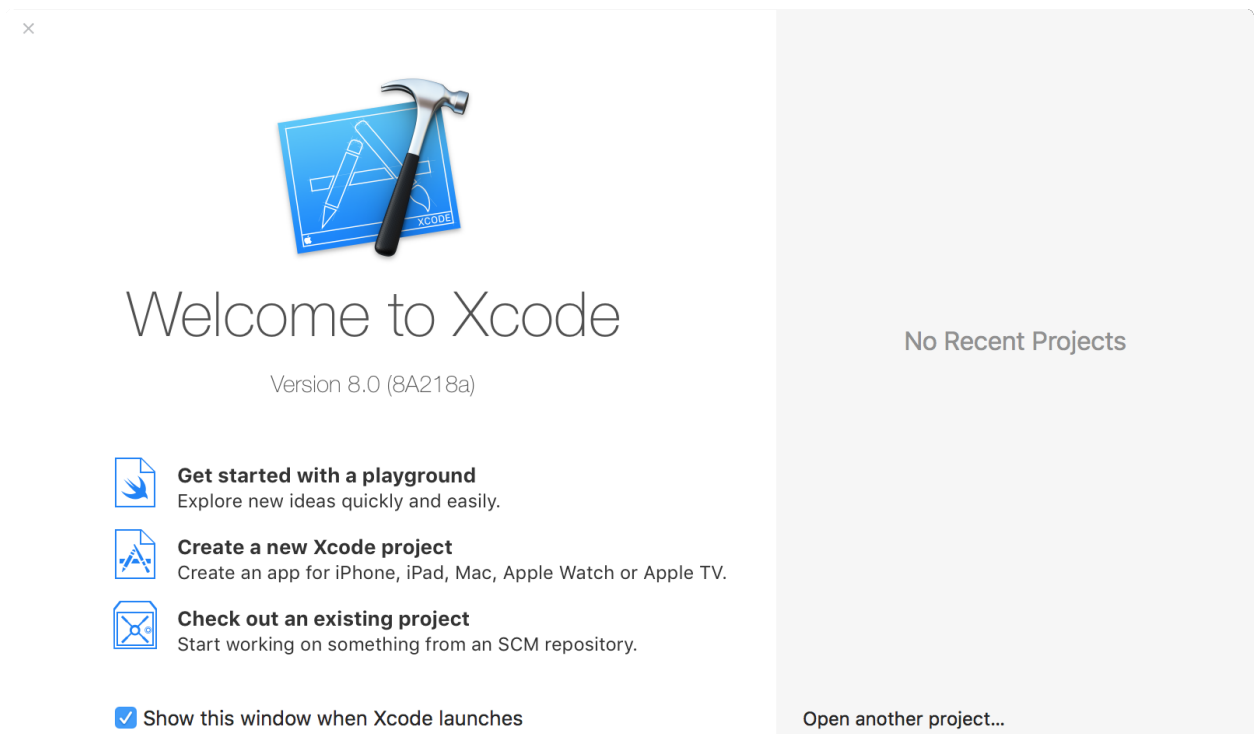


Playgrounds were released in Xcode 6. They provide an interactive environment for rapidly developing and evaluating Swift code and have become a useful prototyping tool. A playground does not require that you compile and run a complete project. Instead, playgrounds evaluate your Swift code on the fly, so they are ideal for testing and experimenting with the Swift language in a lightweight environment. You will be using playgrounds frequently throughout this book to get quick feedback on your Swift code.

In addition to playgrounds, you will create native command-line tools in later chapters. Why not just use playgrounds? You would miss out on a lot of Xcode's features and would not get as much exposure to the IDE. You will be spending a lot of time in Xcode, and it is good to get comfortable with it as soon as possible.

From the welcome screen, select Get started with a playground.

Next, name your playground MyPlayground. For the platform (iOS, macOS, or tvOS), select macOS, even if you are an iOS developer (Figure 1.2). The Swift features you will be covering are common to both platforms. Click Next.

Figure 1.2  Naming a playground
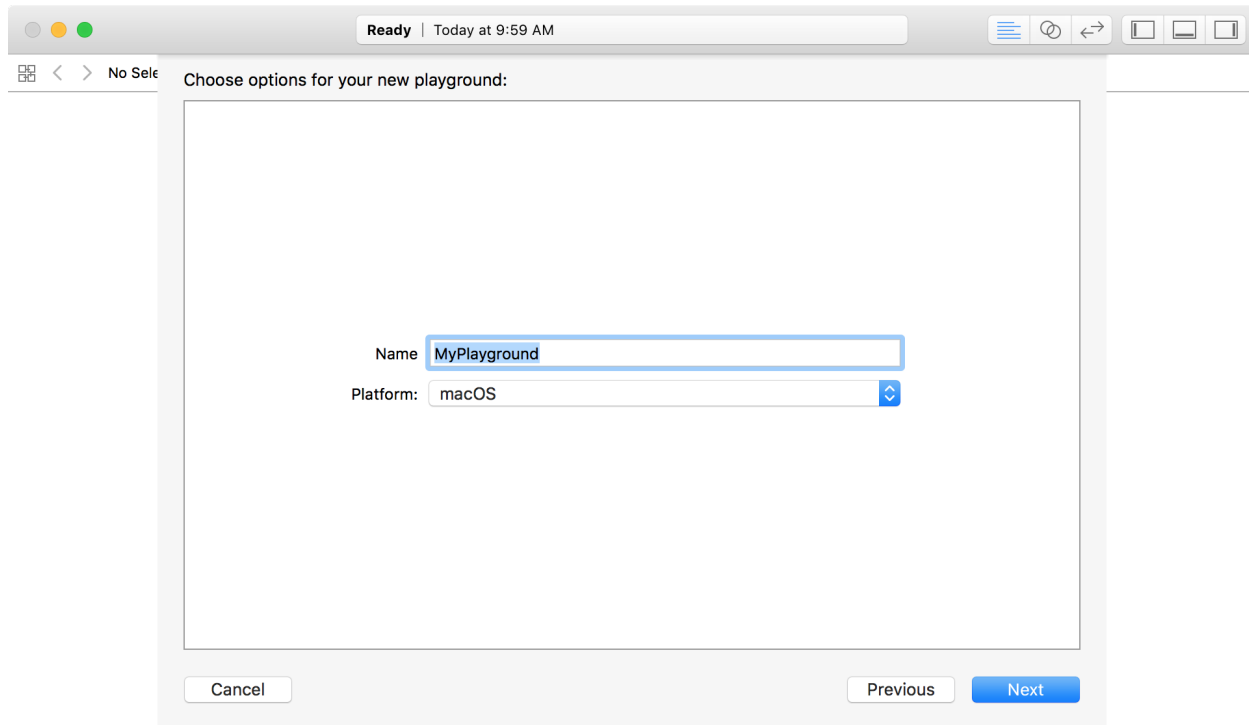


Finally, you are prompted to save your playground. As you work through this book, it is a good idea to put all of your work in one folder. Choose a location that works for you and click Create (Figure 1.3).

4

Figure 1.3  Saving a playground
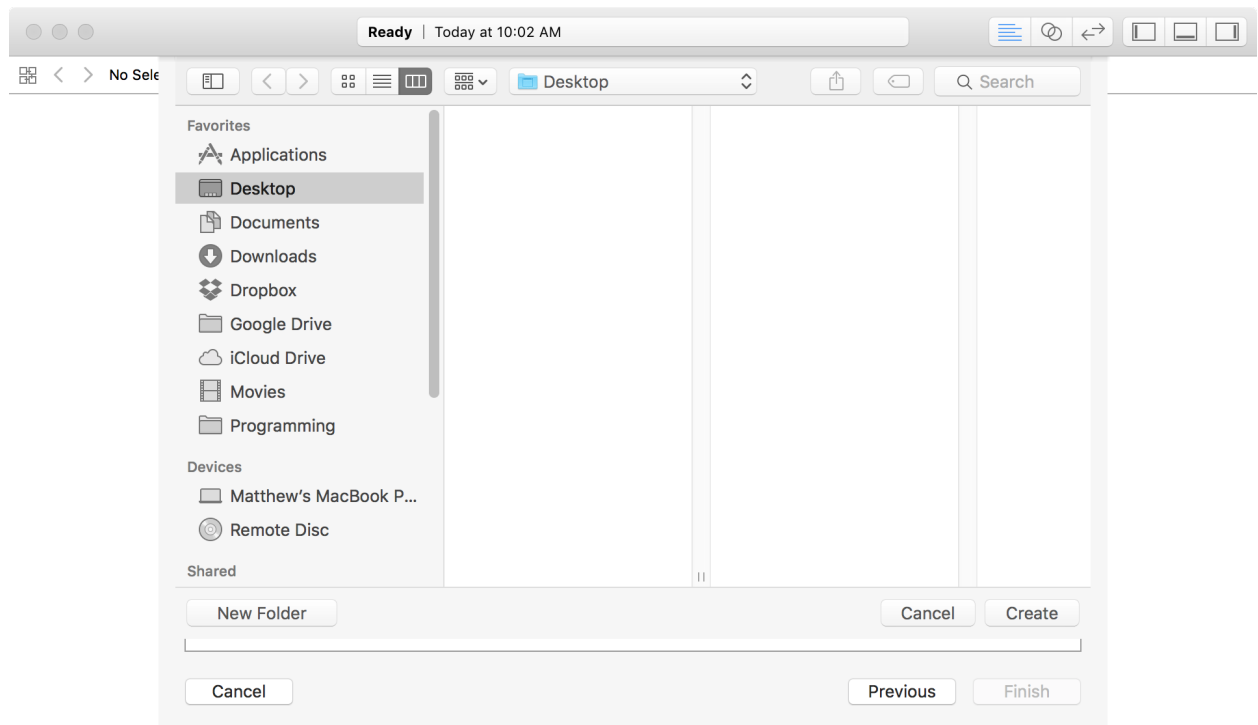


# Playing in a Playground

As you can see in Figure 1.4, a Swift playground opens with two sections. On the left, you have the Swift code editor. On the right, you have the results sidebar. The code in the editor is evaluated and run top to bottom, if possible, every time the source changes. The results of the code are displayed in the results sidebar.
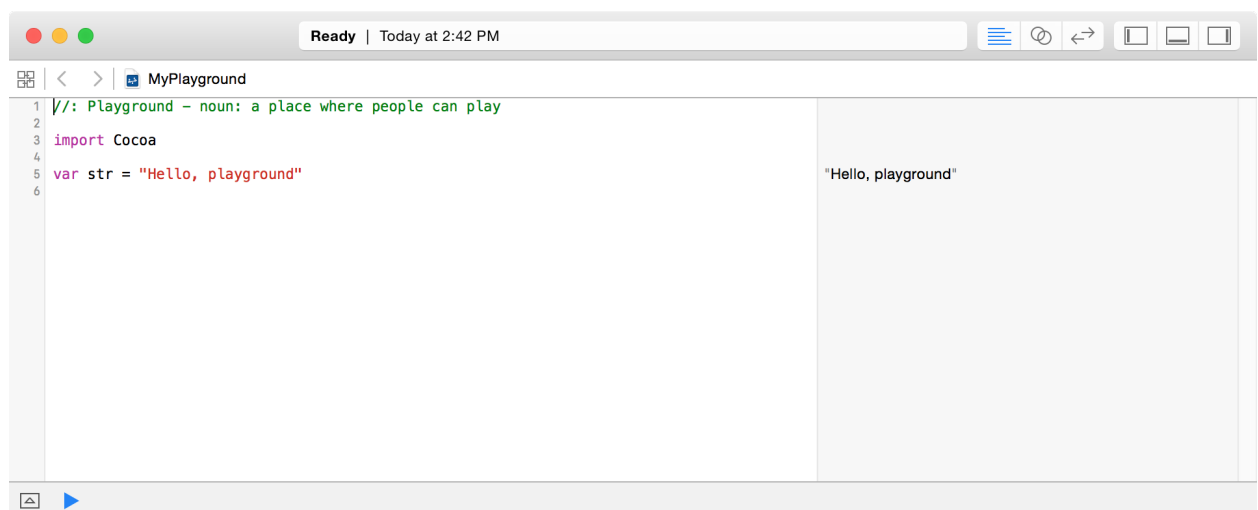
Figure 1.4  Your new playground

Let's take a look at your new playground. Notice that the first line of text is green and that it begins with two forward slashes: `//`. The slashes signify to the compiler that the line is a *comment*, and Xcode shows comments in green.

Developers use comments as inline documentation or for notes to help keep track of what happens where. The forward slashes tell the compiler that it should not treat that line as code. Delete the forward slashes. The compiler will issue an error complaining that it cannot parse the expression. Add the slashes back using the handy keyboard shortcut Command-/. (If you just installed Xcode and Command-/ does not work, restart your computer and try again.)

Just below the comment, the playground imports the Cocoa framework. This import statement means that your playground has complete access to all of the application programming interfaces (APIs) in the Cocoa framework. (An API is similar to a prescription – or set of definitions – for how a program can be written.)

Below the import statement is a line that reads `var str = "Hello, playground"`. The text in quotes is copied on the right, in the results sidebar: `"Hello, playground"`. Let's take a closer look at that line of code.

First, notice the equals sign, which is called the *assignment operator*. The assignment operator assigns the result of code on its righthand side to a constant or variable on its lefthand side. For example, on the lefthand side of the equals sign, you have the text `var str`. Swift's keyword `var` is used to declare a variable. This is an important concept that you will see in greater detail in the next chapter. For now, a variable represents some value that you expect to change or vary.

On the righthand side of the equality, you have `"Hello, playground"`. In Swift, the quotation marks indicate a **String**, an ordered collection of characters. The template named this new variable `str`, but variables can be named almost anything. (There are limitations, of course. Try to change the name `str` to be `var`. What happens? Why do you think you cannot name your variable `var`? Be sure to change the name back to `str` before moving on.)

Now you can understand the text printed on the right in the results sidebar: It is the string value assigned to the variable `str`.

## Varying Variables and Printing to the Console

**String** is a *type*, and we say that the `str` variable is "an instance of the **String** type." Types describe a particular structure for representing data. Swift has many types, which you will meet throughout this book. Each type has specific abilities (what the type can do with data) and limitations (what it cannot do with data). For example, the **String** type is designed to work with an ordered collection of characters and defines a number of functions to work with that ordered collection of characters.

Recall that `str` is a variable. That means you can change `str`'s value. Let's append an exclamation point to the end of the string to make it a well-punctuated sentence. (Whenever new code is added in this book, it will be shown in bold. Deletions will be struck through.)

Listing 1.1  Proper punctuation

```
import Cocoa

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

To add the exclamation point, you are using the += *addition assignment operator*. The addition assignment operator combines the addition (+) and assignment (=) operations in a single operator. (You will learn more about operators in Chapter 3.)

Did you notice anything in the results sidebar on the right? You should see a new line of results representing `str`'s new value, complete with exclamation point (Figure 1.5).

## Figure 1.5  Varying `str`
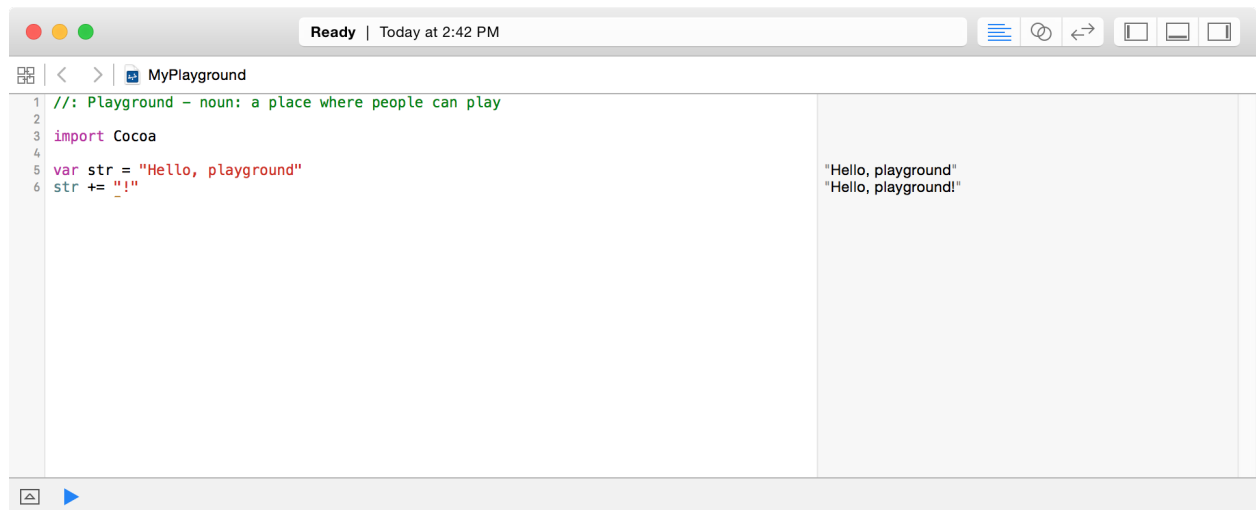


Next, add some code to print the value held by the variable `str` to the *console*. In Xcode, the console displays text messages that you create and want to log as things occur in your program. Xcode also uses the console to display warnings and errors as they occur.

To print to the console, you will use the function **print()**. *Functions* are groupings of related code that send instructions to the computer to complete a specific task. **print()** is a function used to print a value to the console, followed by a line break. Unlike playgrounds, Xcode projects do not have a results sidebar, so you will use the **print()** function frequently when you are writing fully featured apps. The console is useful for checking the current value of some variable of interest.

## Listing 1.2  Printing to the console

```
import Cocoa

var str = "Hello, playground"
str += "!"
print(str)
```

After you enter this line and the playground executes the code, the console will open automatically at the bottom of the Xcode screen. (If it does not, you can open the *debug area* to see it. Click on View → Debug Area → Show Debug Area, as shown in Figure 1.6. Notice the keyboard shortcut next to this last step? You can also type Shift-Command-Y on your keyboard to open the debug area.)

Figure 1.6  Showing the debug area
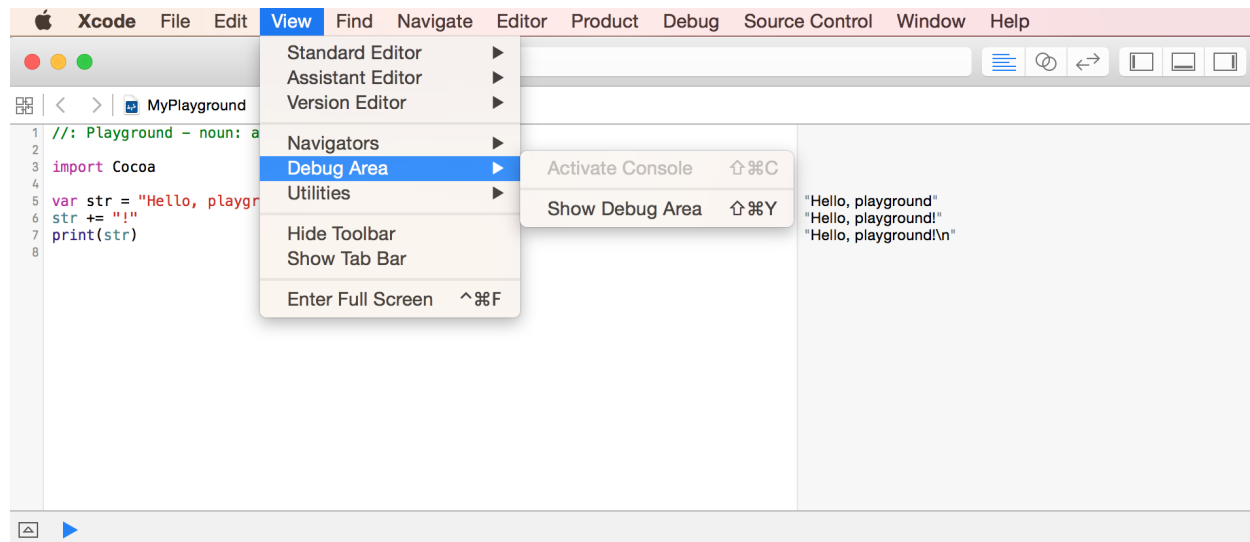


Now that you have your debug area open, you should see something like Figure 1.7.

Figure 1.7  Your first Swift code
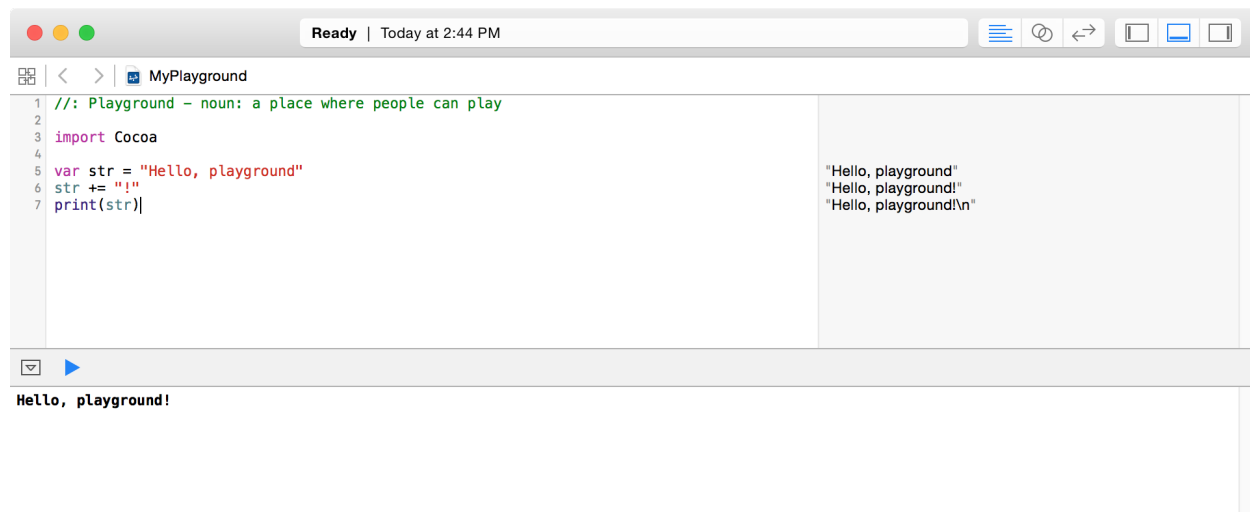


# You Are on Your Way!

Let's review what you have accomplished so far. You have:

- installed Xcode

- created and gotten acquainted with a playground

- used a variable and modified it

- learned about the **String** type

- used a function to print to the console

That is good! You will be making your own apps in no time. Until then, stick with it. As you continue, you will see that most everything in this book is merely a variation on the themes you have covered thus far.

# Bronze Challenge

Many of the chapters in this book end with one or more challenges. The challenges are for you to work through on your own to deepen your understanding of Swift and get a little extra experience. Your first challenge is below. Before you get started, create a new playground.

You learned about the `String` type and printing to the console using `print()`. Use your new playground to create a new instance of the `String` type. Set the value of this instance to be equal to your last name. Print its value to the console.

# Types, Constants, and Variables

This chapter will introduce you to Swift's basic data types, constants, and variables. These elements are the fundamental building blocks of any program. You will use constants and variables to store values and to pass data around in your applications. Types describe the nature of the data held by the constant or variable. There are important differences between constants and variables, as well as each of the data types, that shape their uses.

## Types

Variables and constants have a data type. The type describes the nature of the data and provides information to the compiler on how to handle the data. Based on the type of a constant or variable, the compiler knows how much memory to reserve and will also be able to help with *type checking*, a feature of Swift that helps prevent you from assigning the wrong kind of data to a variable.

Let's see this in action. Create a new macOS playground. (From the welcome screen, choose Get started with a playground. From within Xcode, choose File → New → Playground... .) Name the playground `Variables`.

Suppose you want to model a small town in your code. You might want a variable for the number of stoplights in the town. Remove the code that came with the template, create a variable called `numberOfStoplights`, and give it a value. (Remember that code you are to delete is shown struck through.)

Listing 2.1  Assigning a string to a variable

```
import Cocoa

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

Here, you have assigned an instance of the **String** type to the variable called `numberOfStoplights`. Let's go piece by piece to see why this is so. The assignment operator (=) assigns the value on its right side to whatever is on its left side. Swift uses *type inference* to determine the data type of your variable. In this case, the compiler knows the variable `numberOfStoplights` is of the **String** type because the value on the right side of the assignment operator is an instance of **String**. Why is "Four" an instance of the **String** type? Because the quotation marks indicate that it is a **String** literal.

Now add the integer 2 to your variable, using += as you did in the last chapter.

Listing 2.2  Adding **"Four"** and 2

```
import Cocoa

var numberOfStoplights = "Four"
numberOfStoplights += 2
```

The compiler gives you an error telling you that this operation does not make sense. You get this error because you are trying to add a number to a variable that is an instance of a different type: **String**. What does it mean to add the number 2 to a string? Does it double the string and give you "FourFour"? Does it put "2" on the end and give you "Four2"? Nobody knows. It just does not make sense to add a number to an instance of **String**.

If you are thinking that it does not make sense to have `numberOfStoplights` be of type **String** in the first place, you are right. Because this variable represents the *number* of stoplights in your theoretical town, it makes sense to use a *numerical* type. Swift provides an **Int** type to represent whole integers that is perfect for your variable. Change your code to use **Int** instead.

### Listing 2.3  Using a numerical type

```
import Cocoa

var numberOfStoplights = "Four"
var numberOfStoplights: Int = 4
numberOfStoplights += 2
```

Let's take a look at the changes here. Before, the compiler relied on type inference to determine the data type of the variable `numberOfStoplights`. Now, you are explicitly declaring the variable to be of the **Int** type using Swift's *type annotation* syntax. The colon in the code above represents the phrase "*of type*." This code could be read as: "Declare a variable called `numberOfStoplights` of type **Int** that starts out with a value of 4."

Note that type annotation does not mean that the compiler is no longer paying attention to what is on each side of the equality. If, for example, you tried to reassign your previous **String** instance of `"Four"` to be an integer using type annotation, the compiler would give you a warning telling you that it cannot convert a string to an integer.

Swift has a host of frequently used data types. You will learn more about strings, which contain textual data, in Chapter 7 and numbers in Chapter 4. Other commonly used types are the various *collection types*, which you will see later in the book.

Notice something else about the new code you entered: Your error has disappeared. It is fine to add 2 to the integer variable representing your town's number of stoplights. In fact, because you have declared this instance to be a variable, this operation is perfectly natural.

## Constants vs Variables

We said that types describe the nature of the data held by a constant or variable. What, then, are constants and variables? Up to now, you have only seen variables. Variables' values can vary, which means that you can assign them a new value. You varied `numberOfStoplights`'s value in this code: `numberOfStoplights += 2`.

Often, however, you will want to create instances with values that do not change. Use *constants* for these cases. As the name indicates, the value of a constant cannot be changed.

You made `numberOfStoplights` a variable, and you changed its value. But what if you did not want to vary the value of `numberOfStoplights`? In that case, making `numberOfStoplights` a constant would be better. A good rule of thumb is to use variables for instances that must vary and constants for instances that will not.

Swift has different syntax for declaring constants and variables. As you have seen, you declare a variable with the keyword `var`. You use the `let` keyword to declare that an instance is a constant.

Declare a constant in the current playground to fix the number of stoplights in your small town.

### Listing 2.4  Declaring a constant

```
import Cocoa

var numberOfStoplights: Int = 4
let numberOfStoplights: Int = 4
numberOfStoplights += 2
```

You declare `numberOfStoplights` to be a constant via the `let` keyword. Unfortunately, this change causes the compiler to issue an error. Why?

You have just changed `numberOfStoplights` to be a constant, but you still have code that attempts to change its value: `numberOfStoplights += 2`. Because constants cannot change, the compiler gives you an error when you try to change it. Fix the problem by removing the addition and assignment code.

### Listing 2.5  Constants do not vary

```
import Cocoa

let numberOfStoplights: Int = 4
numberOfStoplights += 2
```

Now, add an **Int** to represent the town's population. (Do you think it should be a variable or a constant?)

### Listing 2.6  Declaring population

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
```

Your town's population is likely to vary over time. Thus, you declared `population` with the `var` keyword to make this instance a variable. You also declared `population` to be an instance of type **Int**. You did so because a town's population is counted in terms of whole persons. But you did not *initialize* `population` with any value. It is therefore an *uninitialized* **Int**.

(Initialization, which you will learn more about in Chapter 17, is the operation of setting up an instance of a type so that it is prepared and available to use.)

Use the assignment operator to give `population` its starting value.

### Listing 2.7  Giving population a value

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
```

# String Interpolation

Every town needs a name. Your town is fairly stable, so it will not be changing its name any time soon. Make the town name a constant of type **String**.

### Listing 2.8  Giving the town a name

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
let townName: String = "Knowhere"
```

It would be nice to have a short description of the town that the Tourism Council could use. The description is going to be a constant **String**, but you will be creating it a bit differently than the constants and variables you have created so far. The description will include all the data you have entered, and you are going to create it using a Swift feature called *string interpolation*.

String interpolation lets you combine constant and variable values into a new string. You can then assign the string to a new variable or constant or just print it to the console. You are going to print the town description to the console.

Listing 2.9  Crafting the town description

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
let townName: String = "Knowhere"
let townDescription =
"\(townName) has a population of \(population) and \(numberOfStoplights) stoplights."
print(townDescription)
```

The \() syntax represents a placeholder in the **String** literal that accesses an instance's value and places it (or "interpolates" it) within the new **String**. For example, \(townName) accesses the constant townName's value and places it within the new **String** instance.

The result of the new code is shown in Figure 2.1.

Figure 2.1  Knowhere's short description
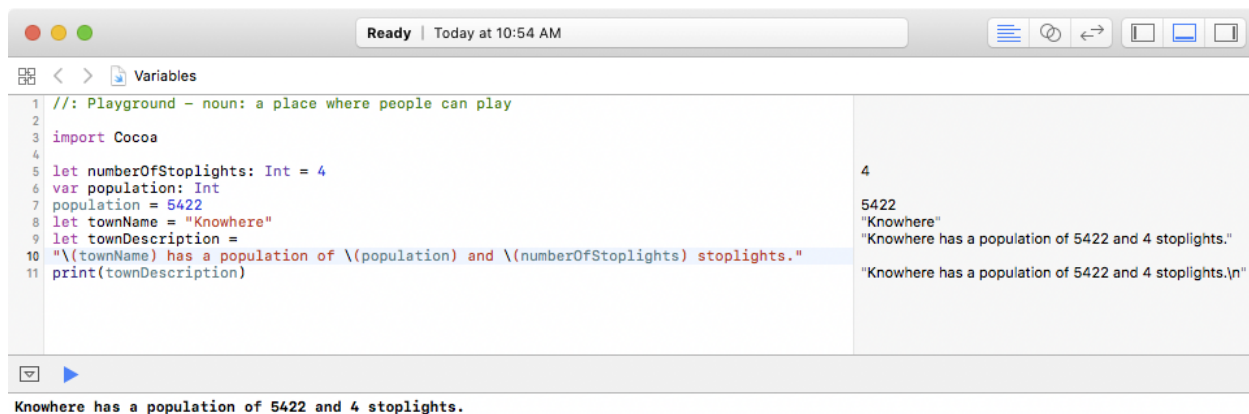


# Bronze Challenge

Add a new variable to your playground representing Knowhere's level of unemployment. Which data type should you use? Give this variable a value and update townDescription to use this new information.

# Part II
## The Basics

Programs execute code in a specific order. Writing software means having control over the order in which code executes. Programming languages provide *control flow statements* to help developers organize the execution of their code. This part of the book introduces the concepts of conditionals and loops to accomplish this task.

The chapters in this section of the book will also show you how Swift represents numbers and text – often called strings – in code. These types of data are the building blocks of many applications. By the end of these chapters, you will have a good understanding of how numbers and strings work in Swift.

Last, this part of the book introduces the concept of *optionals* in Swift. Optionals play an important role in the language and provide a mechanism for the language to represent the concept of *nothing* in a safe way. As you will see, how Swift deals with optionals highlights the language's approach to writing safe and reliable code.

# 3

# Conditionals

In previous chapters your code led a relatively simple life: You declared some constants and variables and then assigned them values. But of course, an application really comes to life – and programming becomes a bit more challenging – when the application makes decisions based on the contents of its variables. For example, a game may let players leap a tall building *if* they have eaten a power-up. You use conditional statements to help applications make these kind of decisions.

## if/else

`if/else` statements execute code based on a specific logical condition. You have a relatively simple either/or situation, and depending on the result one branch of code or another (but not both) runs.

Consider Knowhere, your small town from the previous chapter, and imagine that you need to buy stamps. Either Knowhere has a post office or it does not. If it has a post office, you will buy stamps there. If it does not have a post office, you will need to drive to the next town to buy stamps. Whether there is a post office is your logical condition. The different behaviors are "get stamps in town" and "get stamps out of town."

Some situations are more complex than a binary yes/no. You will see a more flexible mechanism called `switch` in Chapter 5. But for now, let's keep it simple.

Create a new macOS playground and name it `Conditionals`. Enter the code below, which shows the basic syntax for an `if/else` statement:

Listing 3.1  Big or small?

```
import Cocoa

var str = "Hello, playground"
var population: Int = 5422
var message: String

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
print(message)
```
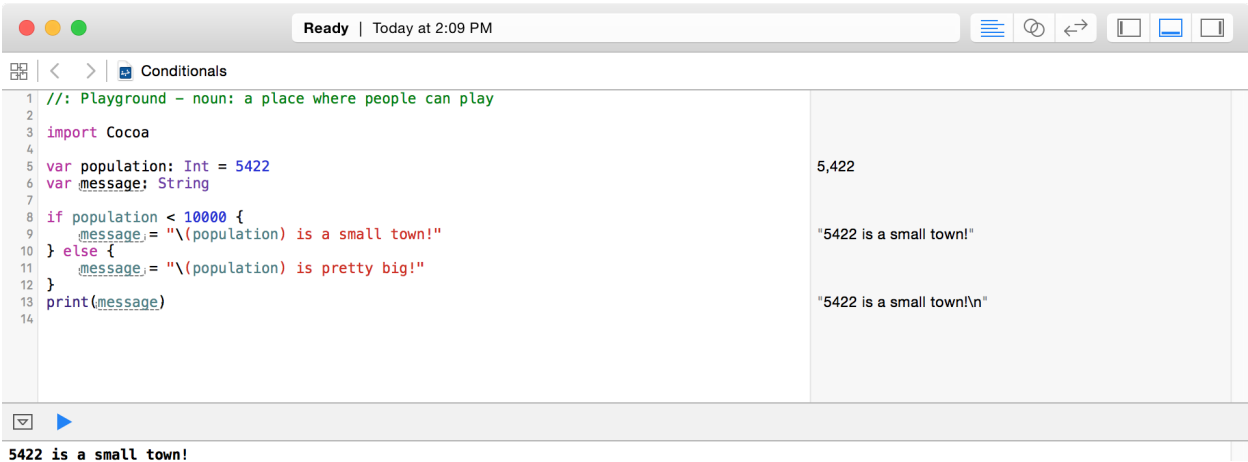
You first declare `population` as an instance of the **Int** type and then assign it a value of 5,422. You also declare a variable called `message` that is of the **String** type. You leave this variable uninitialized at first, meaning that you do not assign it a value.

Next comes the conditional `if/else` statement. This is where `message` is assigned a value based on whether the "if" statement evaluates to true. (Notice that you use *string interpolation* to put the population into the `message` string.)

Figure 3.1 shows what your playground should look like. The console and the results sidebar show that `message` has been set to be equal to the string literal assigned when the conditional evaluates to true. How did this happen?

## Figure 3.1  Conditionally describing a town's population



```
 1  //: Playground - noun: a place where people can play
 2
 3  import Cocoa
 4
 5  var population: Int = 5422                                    5,422
 6  var message: String
 7
 8  if population < 10000 {
 9      message = "\(population) is a small town!"               "5422 is a small town!"
10  } else {
11      message = "\(population) is pretty big!"
12  }
13  print(message)                                               "5422 is a small town!\n"
14
```

```
5422 is a small town!
```

The condition in the `if`/`else` statement tests whether your town's population is less than 10,000 via the `<` *comparison operator*. If the condition evaluates to true, then `message` is set to be equal to the first string literal (`"X is a small town!"`). If the condition evaluates to false – if the population is 10,000 or greater – then `message` is set to be equal to the second string literal (`"X is pretty big!"`). In this case, the town's population is less than 10,000, so `message` is set to `"5422 is a small town!"`.

Table 3.1 lists Swift's comparison operators.

## Table 3.1  Comparison operators

| Operator | Description |
|---|---|
| < | Evaluates whether the value on the left is less than the value on the right. |
| <= | Evaluates whether the value on the left is less than or equal to the value on the right. |
| > | Evaluates whether the value on the left is greater than the value on the right. |
| >= | Evaluates whether the value on the left is greater than or equal to the value on the right. |
| == | Evaluates whether the value on the left is equal to the value on the right. |
| != | Evaluates whether the value on the left is not equal to the value on the right. |
| === | Evaluates whether the two instances point to the same reference. |
| !== | Evaluates whether the two instances do not point to the same reference. |

You do not need to understand all of the operators' descriptions right now. You will see many of them in action as you move through the book, and they will become clearer as you use them. Refer back to this table as a reference if you have questions.

Sometimes you only care about one aspect of the condition that is under evaluation. That is, you want to execute code if a certain condition is met and do nothing if it is not. Enter the code below. (Notice that new code, shown in bold, appears in two places.)

Listing 3.2  Is there a post office?

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

Here, you add a new variable called hasPostOffice. This variable has the type **Bool**, short for "Boolean." Boolean types can take one of two values: true or false. In this case, the Boolean hasPostOffice variable keeps track of whether the town has a post office. You set it to true, meaning that it does.

The ! is a *logical operator* known as *logical not*. It tests whether hasPostOffice is false. You can think of ! as inverting a Boolean value: True becomes false, and false becomes true.

The code above first sets hasPostOffice to true, then asks whether it is false. If hasPostOffice is false, you do not know where to buy stamps, so you ask. If hasPostOffice is true, you know where to buy stamps and do not have to ask, so nothing happens.

Because the town *does* have a post office (because hasPostOffice was initialized to true), the condition ! hasPostOffice is false. That is, it is *not* the case that hasPostOffice is false. Therefore, the **print()** function never gets called.

Table 3.2 lists Swift's logical operators.

Table 3.2  Logical operators

| Operator | Description |
|----------|-------------|
| && | Logical and: true if and only if both are true (false otherwise). |
| \|\| | Logical or: true if either is true (false only if both are false). |
| ! | Logical not: true becomes false, false becomes true. |

# Ternary Operator

The *ternary operator* is very similar to an if/else statement, but has the more concise syntax a ? b : c. In English, the ternary operator reads something like, "If a is true, then do b. Otherwise, do c."

Let's rewrite the town population check that used if/else using the ternary operator instead.

Listing 3.3  Using the ternary operator

```
...
if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}

message = population < 10000 ? "\(population) is a small town!" :
                    "\(population) is pretty big!"
...
```

The ternary operator can be a source of controversy: Some programmers love it; some programmers loathe it. We come down somewhere in the middle. This particular usage is not very elegant. Your assignment to `message` requires more than a simple `a ? b : c`. The ternary operator is great for concise statements, but if your statement starts wrapping to the next line, we think you should use `if/else` instead.

Hit Command-Z to undo, removing the ternary operator and restoring your `if/else` statement.

Listing 3.4  Restoring `if/else`

```
...
message = population < 10000 ? "\(population) is a small town!" :
                    "\(population) is pretty big!"
if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
...
```

# Nested ifs

You can nest `if` statements for scenarios with more than two possibilities. You do this by writing an `if/else` statement inside the curly braces of another `if/else` statement. To see this, nest an `if/else` statement within the `else` block of your existing `if/else` statement.

Listing 3.5  Nesting conditionals

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
        message = "\(population) is pretty big!"
    }
}
print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

Your nested `if` clause makes use of the `>=` *comparator* (comparison operator) and the `&&` logical operator to check whether `population` is within the range of 10,000 to 50,000. Because your town's `population` does not fall within that range, your `message` is set to `"5422 is a small town!"` as before.

Try bumping up the population to exercise the other branches.

Nested `if/else` statements are common in programming. You will find them out in the wild, and you will be writing them as well. There is no limit to how deeply you can nest these statements. However, the danger of nesting them too deeply is that it makes the code harder to read. One or two levels are fine, but beyond that your code becomes less readable and maintainable.

There are ways to avoid nested statements. Next, you are going to *refactor* the code that you have just written to make it a little easier to follow. Refactoring means changing code so that it does the same work but in a different way. It may be more efficient, be easier to understand, or just look prettier.

# else if

The `else if` conditional lets you chain multiple conditional statements together. `else if` allows you to check against multiple cases and conditionally executes code depending on which clause evaluates to true. You can have as many `else if` clauses as you want. Only one condition will match.

To make your code a little easier to read, extract the nested `if/else` statement to be a standalone clause that evaluates whether your town is of medium size.

Listing 3.6 Using `else if`

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else if population >= 10000 && population < 50000 {
    message = "\(population) is a medium town!"
}  else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
    message = "\(population) is pretty big!"
    }
}
print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

You are using one `else if` clause, but you could have chained many more. This block of code is an improvement over the nested `if/else` above. If you find yourself with lots of `if/else` statements, you may want to use another mechanism – such as `switch`, described in Chapter 5. Stay tuned.

# Bronze Challenge

Add an additional `else if` statement to the town-sizing code to see if your town's `population` is very large. Choose your own population thresholds. Set the `message` variable accordingly.

# Numbers

Numbers are the fundamental language of computers. They are also a staple of software development. Numbers are used to keep track of temperature, determine how many letters are in a sentence, and count the zombies infesting a town. Numbers come in two basic flavors: integers and floating-point numbers.

## Integers

You have worked with integers already, but we have not yet defined them. An integer is a number that does not have a decimal point or fractional component – a whole number. Integers are frequently used to represent a count of "things," such as the number of pages in a book. A difference between integers used by computers and numbers you use elsewhere is that an integer type on a computer takes up a fixed amount of memory. Therefore, they cannot represent all possible whole numbers – they have a minimum and maximum value.

We could tell you those minimum and maximum values, but we are going to let Swift tell you instead. Create a new playground, name it `Numbers`, and enter the following code.

Listing 4.1  Maximum and minimum values for **Int**

```
import Cocoa

var str = "Hello, playground"

print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
```

In the console, you should see the following output:

```
The maximum Int value is 9223372036854775807.
The minimum Int value is −9223372036854775808.
```

Why are those numbers the minimum and maximum **Int** values? Computers store integers in binary form with a fixed number of bits. A bit is a single 0 or 1. Each bit position represents a power of 2; to compute the value of a binary number, add up each of the powers of 2 whose bit is a 1. For example, the binary representations of 38 and -94 using an 8-bit signed integer are shown in Figure 4.1. (Note that the bit positions are read from right to left. *Signed* means that the integer can represent positive or negative values. More about signed integers in a moment.)

Figure 4.1  Binary numbers

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

$-2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$

$= 2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$

| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

$-2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$

$= 2^1 + 2^5 - 2^7 = 2 + 32 - 128 = -94$

In macOS, `Int` is a 64-bit integer, which means it has $2^{64}$ possible values. Imagine Figure 4.1, only 64 bits wide instead of 8. The power of 2 represented by the top (left-most) bit would be $-2^{63}$ = -9,223,372,036,854,775,808 – exactly the value you see for `Int.min` in your playground. And, if you were to add up $2^0$, $2^1$, …, $2^{62}$, you would arrive at 9,223,372,036,854,775,807 – the value you see for `Int.max`.

In iOS, `Int` is slightly more complicated. Apple introduced 64-bit devices starting with iPhone 5S, iPad Air, and iPad mini with Retina display. Earlier devices had a 32-bit architecture. If you write an iOS app for newer devices, which is called "targeting a 64-bit architecture," `Int` is a 64-bit integer just like in macOS. On the other hand, if you target a 32-bit architecture like iPhone 5 or iPad 2, `Int` is a 32-bit integer. The compiler determines the appropriate size for `Int` when it builds your program.

If you need to know the exact size of an integer, you can use one of Swift's explicitly sized integer types. For example, `Int32` is Swift's 32-bit signed integer type. Use `Int32` to see the minimum and maximum value for a 32-bit integer.

## Listing 4.2  Maximum and minimum values for **Int32**

```
...
print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
print("The maximum value for a 32-bit integer is \(Int32.max).")
print("The minimum value for a 32-bit integer is \(Int32.min).")
```

Also available are `Int8`, `Int16`, and `Int64`, for 8-bit, 16-bit, and 64-bit signed integer types. You use the sized integer types when you need to know the size of the underlying integer, such as for some algorithms (common in cryptography) or to exchange integers with another computer (such as sending data across the internet). You will not use these types much; good Swift style is to use an `Int` for most use cases.

All the integer types you have seen so far are signed, which means they can represent both positive and negative numbers. Swift also has unsigned integer types to represent whole numbers greater than or equal to 0. Every signed integer type (`Int`, `Int16`, etc.) has a corresponding unsigned integer type (`UInt`, `UInt16`, etc.). The difference between signed and unsigned integers at the binary level is that the power of 2 represented by the top-most bit ($2^7$ for 8-bit integers) is positive instead of negative. Test a couple of these.

## Listing 4.3  Maximum and minimum values for unsigned integers

```
...
print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
print("The maximum value for a 32-bit integer is \(Int32.max).")
print("The minimum value for a 32-bit integer is \(Int32.min).")

print("The maximum UInt value is \(UInt.max).")
print("The minimum UInt value is \(UInt.min).")
print("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
print("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")
```

Like `Int`, `UInt` is a 64-bit integer in macOS and may be 32-bit or 64-bit depending on the target for iOS. The minimum value for all unsigned types is 0. The maximum value for an N-bit unsigned type is $2^N$ - 1. For example, the maximum value for a 64-bit unsigned type is $2^{64}$ - 1, which equals 18,446,744,073,709,551,615.

There is a relationship between the minimum and maximum values of signed and unsigned types: The maximum value of `UInt64` is equal to the maximum value of `Int64` plus the absolute value of the minimum value of `Int64`. Both signed and unsigned types have $2^{64}$ possible values, but the signed version has to devote half of the possible values to negative numbers.

Some quantities seem like they would naturally be represented by an unsigned integer. For example, it does not make sense for the count of a number of objects to ever be negative. However, Swift style is to prefer `Int` for all integer uses (including counts) unless an unsigned integer is required by the algorithm or code you are writing. The explanation for this involves topics we are going to cover later in this chapter, so we will return to the reasons behind consistently preferring `Int` soon.

# Creating Integer Instances

You created instances of `Int` in Chapter 2, where you learned that you can declare a type explicitly or implicitly.

### Listing 4.4  Declaring **Int** explicitly and implicitly

```
...
print("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
print("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")

let numberOfPages: Int = 10 // Declares the type explicitly
let numberOfChapters = 3    // Also of type Int, but inferred by the compiler
```

The compiler always assumes that implicit declarations with integer values are of type `Int`. However, you can create instances of the other integer types using explicit type declarations.

### Listing 4.5  Declaring other integer types explicitly

```
...
let numberOfPages: Int = 10 // Declares the type explicitly
let numberOfChapters = 3    // Also of type Int, but inferred by the compiler

let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = -1000
```

What happens if you try to create an instance with an invalid value? What if, for example, you try to create a `UInt` with a negative value or an `Int8` with a value greater than 127? Try it and find out.
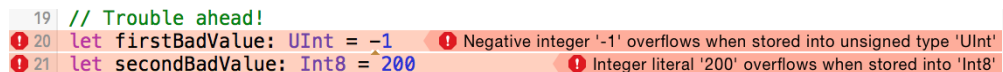
### Listing 4.6  Declaring integer types with invalid values

```
...
let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = -1000

// Trouble ahead!
let firstBadValue: UInt = -1
let secondBadValue: Int8 = 200
```

You should see red exclamation marks in the left column of the playground. Click on the exclamation marks to see the errors (Figure 4.2).

### Figure 4.2  Integer overflow error



The compiler reports that the two values you have typed in "overflow when stored into" constants of type `UInt` and `Int8`, respectively. "Overflows when stored into…" means that when the compiler tried to store your number into the type you specified, it did not fit in the type's allowed range of values. An `Int8` can hold values from -128 to 127; 200 is outside of that range, so trying to store 200 into an `Int8` overflows.

Remove the problematic code.

### Listing 4.7  No more bad values

```
...
// Trouble ahead!
let firstBadValue: UInt = -1
let secondBadValue: Int8 = 200
```

# Operations on Integers

Swift allows you to perform basic mathematical operations on integers using the familiar operators + (add), – (subtract), and ∗ (multiply). Try printing the results of some arithmetic.

Listing 4.8  Performing basic operations

```
...
let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = −1000

print(10 + 20)
print(30 − 5)
print(5 * 6)
```

The compiler respects the mathematical principles of precedence and associativity, which define the order of operations when there are multiple operators in a single expression. For example:

Listing 4.9  Order of operations

```
...
print(10 + 20)
print(30 − 5)
print(5 * 6)

print(10 + 2 * 5) // 20, because 2 * 5 is evaluated first
print(30 − 5 − 5) // 20, because 30 − 5 is evaluated first
```

You could memorize the rules governing precedence and associativity. However, we recommend taking the easy route and using parentheses to make your intentions explicit, because parentheses are always evaluated first.

Listing 4.10  Parentheses are your friends

```
...
print(10 + 2 * 5) // 20, because 2 * 5 is evaluated first
print(30 − 5 − 5) // 20, because 30 − 5 is evaluated first
print((10 + 2) * 5) // 60, because (10 + 2) is now evaluated first
print(30 − (5 − 5)) // 30, because (5 − 5) is now evaluated first
```

# Integer division

What is the value of the expression 11 / 3? You might (reasonably) expect 3.66666666667, but try it out.

Listing 4.11  Integer division can give unexpected results

```
...
print((10 + 2) * 5) // 60, because (10 + 2) is now evaluated first
print(30 − (5 − 5)) // 30, because (5 − 5) is now evaluated first

print(11 / 3) // Prints 3
```

The result of any operation between two integers is always another integer of the same type; 3.66666666667 is not a whole number and cannot be represented as an integer. Swift truncates the fractional part, leaving just 3. If the result is negative, such as −11 / 3, the fractional part is still truncated, giving a result of -3. Integer division, therefore, always rounds toward 0.

It is occasionally useful to get the remainder of a division operation. The remainder operator, %, returns exactly that. (If you are familiar with the modulo operator in math and some other programming languages, be warned: The remainder operator is not the same, and using it on a negative integer may not return what you expect.)

Listing 4.12  Remainders

```
...
print(11 / 3) // Prints 3
print(11 % 3) // Prints 2
print(-11 % 3) // Prints -2
```

## Operator shorthand

All the operators that you have seen so far return a new value. There are also versions of all of these operators that modify a variable in place. An extremely common operation in programming is to increase or decrease the value of an integer by another integer. You can use the += operator, which combines addition and assignment, or the −= operator, which combines subtraction and assignment.

Listing 4.13  Combining addition or subtraction and assignment

```
...
print(11 % 3) // Prints 2
print(-11 % 3) // Prints -2

var x = 10
x += 10 // Is equivalent to: x = x + 10
print("x has had 10 added to it and is now \(x)")
x -= 5 // Is equivalent to: x = x - 5
print("x has had 5 subtracted from it and is now \(x)")
```

There are also shorthand operation-and-assignment combination operators for the other basic math operations: *=, /=, and %=, each of which assigns the result of the operation to the value on the lefthand side of the operator.

## Overflow operators

What do you think the value of z will be in the following code? (Think about it for a minute before you type it in to find out for sure.)

Listing 4.14  Solve for z

```
...
let y: Int8 = 120
let z = y + 10
```

If you thought the value of z would be 130, you are not alone. But type it in, and you will find that instead Xcode is showing you an error. Click on it to see a more detailed message (Figure 4.3).

Figure 4.3  Execution interrupted when adding to an **Int8**
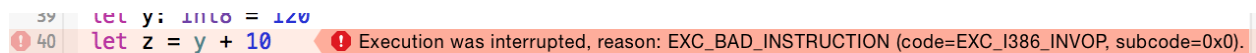


What does "Execution was interrupted" mean? Let's break down what is happening:

1. y is an **Int8**, so the compiler assumes y + 10 must be an **Int8**, too.

2. Therefore, the compiler infers the type of z to be **Int8**.

3. When your playground runs, Swift adds 10 to y, resulting in 130.

4. Before storing the result back into z, Swift checks that 130 is a valid value for an **Int8**.

But **Int8** can only hold values from -128 to 127; 130 is too big! Your playground therefore hits a *trap*, which stops the program from running. We will discuss traps in more detail in Chapter 20. For now, know that a trap results in your program stopping immediately and noisily, which indicates a serious problem you need to examine.

27

Swift provides *overflow operators* that have different behaviors when the value is too big (or too small). Instead of trapping the program, they "wrap around." To see what that means, try it now. The overflow addition operator is &+. Substitute it into your code.

Listing 4.15  Using an overflow operator

```
...
let y: Int8 = 120
let z = y + 10
let z = y &+ 10
print("120 &+ 10 is \(z)")
```

The result of overflow-adding 120 + 10 and storing the result into an **Int8** is -126. Was that what you expected?

Probably not. (And that is OK!) To understand the logic of this result, think about incrementing y one at a time. Because y is an **Int8**, once you get to 127 you cannot go any higher. Instead, incrementing one more time wraps around to -128. So 120 + 8 = -128, 120 + 9 = -127, and 120 + 10 = -126.

There are also overflow versions of the subtraction and multiplication operators: &– and &*. It should be apparent why there is an overflow version of the multiplication operator, but what about subtraction? Subtraction clearly cannot overflow, but it can *underflow*. For example, trying to subtract 10 from an **Int8** currently holding -120 would result in a value too negative to be stored in an **Int8**. Using &– would cause this underflow to wrap back around and give you positive 126.

Integer operations overflowing or underflowing unexpectedly can be a source of serious and hard-to-find bugs. Swift is designed to prioritize safety and minimize these errors. Swift's default behavior of trapping on overflow calculations may come as a surprise to you if you have programmed in another language. Most other languages default to the "wrap-around" behavior that Swift's overflow operators provide. The philosophy of the Swift language is that it is better to trap (even though this may result in a program crashing) than potentially have a security hole. There are some use cases for wrapping arithmetic, so these special operators are available if you need them.

# Converting Between Integer Types

So far, all the operations you have seen have been between two values with exactly the same type. What happens if you try to operate on numbers with different types?

Listing 4.16  Adding values of different types

```
...
let a: Int16 = 200
let b: Int8 = 50
let c = a + b // Uh–oh!
```

This is a compile-time error. You cannot add a and b because they are not of the same type. Some languages will automatically convert types for you to perform operations like this. Swift does not. Instead, you have to manually convert types to get them to match.

In this case, you could either convert a to an **Int8** or convert b to an **Int16**. Actually, though, only one of these will succeed. (Why? Reread the previous section!)

Listing 4.17  Converting type to allow addition

```
...
let a: Int16 = 200
let b: Int8 = 50
let c = a + b // Uh–oh!
let c = a + Int16(b)
```

We can now return to the recommendation to stick with **Int** for almost all integer needs in Swift, even for values that might naturally only make sense as positive values (like a count of "things"). Swift's default type inference for literals is **Int**, and you cannot typically perform operations between different integer types without converting one

of them. Using **Int** consistently throughout your code will greatly reduce the need for you to convert types, and it will allow you to use type inference for integers freely.

Requiring you, the programmer, to decide how to convert variables in order to do math between different types is another feature that distinguishes Swift from other languages. Again, this requirement is in favor of safety and correctness. The C programming language, for example, will convert numbers of different types in order to perform math between them, but the conversions it performs are sometimes "lossy" – you may lose information in the conversion. Swift code that requires math between numbers of different types will be more verbose, but it will be more clear about what conversions are taking place. The increase in verbosity will make it easier for you to reason about and maintain the code.

# Floating-Point Numbers

To represent a number that has a decimal point, like 3.2, you use a *floating-point number*. There are two things to bear in mind about floating-point numbers. First, in computers floating-point numbers are stored as a *mantissa* and an *exponent*, similar to how you write a number in scientific notation. For example, 123.45 could be stored as something like $1.2345 \times 10^2$ or $12.345 \times 10^1$ (although the computer will use base 2 instead of base 10). Additionally, floating-point numbers are often imprecise: There are many numbers that cannot be stored with perfect accuracy in a floating-point number. The computer will store a very close approximation to the number you expect. (More on that in a moment.)

Swift has two basic floating-point number types: **Float**, which is a 32-bit floating-point number, and **Double**, which is a 64-bit floating-point number. The different bit sizes of **Float** and **Double** do not determine a simple minimum and maximum value range as they do for integers. Instead, the bit sizes determine how much precision the numbers have. **Double** has more precision than **Float**, which means it is able to store more accurate approximations.

The default inferred type for floating-point numbers in Swift is **Double**. As with different types of integers, you can also declare **Float**s and **Double**s explicitly.

Listing 4.18  Declaring floating-point number types

```
...
let d1 = 1.1 // Implicitly Double
let d2: Double = 1.1
let f1: Float = 100.3
```

All the same numeric operators work on floating-point numbers (except for the remainder operator, which is typically only used on integers anyway).

Listing 4.19  Operations on floating-point numbers

```
...
let d1 = 1.1 // Implicitly Double
let d2: Double = 1.1
let f1: Float = 100.3

print(10.0 + 11.4)
print(11.0 / 3.0)
```

The fact that floating-point numbers are inherently imprecise is an important difference from integer numbers that you should keep in mind. Let's see an example. Recall the == operator from Chapter 3, which determines whether two values are equal to each other. As you might expect, you can also use it to compare floating-point numbers.

Listing 4.20  Comparing two floating-point numbers

```
...
print(10.0 + 11.4)
print(11.0 / 3.0)

if d1 == d2 {
    print("d1 and d2 are the same!")
}
```

d1 and d2 were both initialized with a value of `1.1`. So far, so good. Now, let's add 0.1 to d1. You would expect that to result in 1.2, so compare the result to that value.

## Listing 4.21  Unexpected results

```
if d1 == d2 {
    print("d1 and d2 are the same!")
}

print("d1 + 0.1 is \(d1 + 0.1)")
if d1 + 0.1 == 1.2 {
    print("d1 + 0.1 is equal to 1.2")
}
```

The results you get may be very surprising! You should see the output `d1 + 0.1 is 1.2` from your first **print()**, but the **print()** inside the if statement does not run. Why not? Isn't 1.2 equal to 1.2?

Well, sometimes it is and sometimes it is not.

As we said before, many numbers – including 1.2 – cannot be represented exactly in a floating-point number. Instead, the computer stores a very close approximation to 1.2. When you add 1.1 and 0.1, the result is really something like 1.2000000000000001. The value stored when you typed the literal 1.2 is really something like 1.1999999999999999. Swift will round both of those to 1.2 when you print them. But they are not technically equal, so the **print()** inside the if statement does not execute.

All the gory details behind floating-point arithmetic are outside the scope of this book. The moral of this story is to be aware that there are some potential pitfalls with floating-point numbers. One consequence is that you should never use floating-point numbers for values that must be exact (such as calculations dealing with money). There are other tools available for those purposes.

# Bronze Challenge

Set down your computer and grab a pencil and paper for this challenge. What is the binary representation of -1 using an 8-bit signed integer?

If you took that same bit pattern and interpreted it as an 8-bit unsigned integer, what would the value be?

# 5

# Switch

In an earlier chapter, you saw one sort of conditional statement: `if/else`. Along the way, we mentioned that `if/else` can be somewhat inadequate in scenarios that have more than a few conditions. This chapter looks at the `switch` statement. Unlike `if/else`, `switch` is ideal for handling multiple conditions. As you will see, Swift's `switch` statement is an incredibly flexible and powerful feature of the language.

## What Is a Switch?

`if/else` statements execute code based on whether the condition under consideration evaluates to true. In contrast, `switch` statements consider a particular value and attempt to match it against a number of *cases*. If there is a match, the `switch` executes the code associated with that case. Here is the basic syntax of a `switch` statement:

```
switch aValue {
case someValueToCompare:
    // Do something to respond

case anotherValueToCompare:
    // Do something to respond

default:
    // Do something when there are no matches
}
```

In the example above, the `switch` only compares against two cases, but a `switch` statement can include any number of cases. If `aValue` matches any of the comparison cases, then the body of that case will be executed.

Notice the use of the `default` case. It is executed when the comparison value does not match any of the cases. The `default` case is not mandatory. However, it *is* mandatory for `switch` statements to have a case for every value of the type being checked. So it is often efficient to use the `default` case rather than providing a specific case for every value in the type.

As you might guess, in order for the comparisons to be possible the type in each of the cases must match the type being compared against. In other words, `aValue`'s type must match the types of `someValueToCompare` and `anotherValueToCompare`.

This code shows the basic syntax of a `switch` statement, but it is not completely well formed. In fact, this `switch` statement would cause a compile-time error. Why? If you are curious, type it into a playground and see. Give `aValue` and all of the cases values. You should see an error for each of the cases, telling you `"'case' label in a 'switch' should have at least one executable statement."`

The problem is that every case must have at least one executable line of code associated with it. This is the purpose of a `switch` statement: Each case should represent a separate branch of execution. In the example, the cases only have comments under them. Because comments are not executable, the `switch` statement does not meet this requirement.

## Switch It Up

Create a new playground called `Switch` and set up a `switch`.

## Listing 5.1  Your first `switch`

```
import Cocoa

var str = "Hello, playground"

var statusCode: Int = 404
var errorString: String
switch statusCode {
case 400:
    errorString = "Bad request"

case 401:
    errorString = "Unauthorized"

case 403:
    errorString = "Forbidden"

case 404:
    errorString = "Not found"

default:
    errorString = "None"
}
```

The `switch` statement above compares an HTTP status code against four cases in order to match a **String** instance describing the error. Because `case 404` matches `statusCode`, `errorString` is assigned to be equal to `"Not found"`, as you can see in the sidebar (Figure 5.1). Try changing the value of `statusCode` to see the other results. When you are done, set it back to `404`.

## Figure 5.1  Matching an error string to a status code
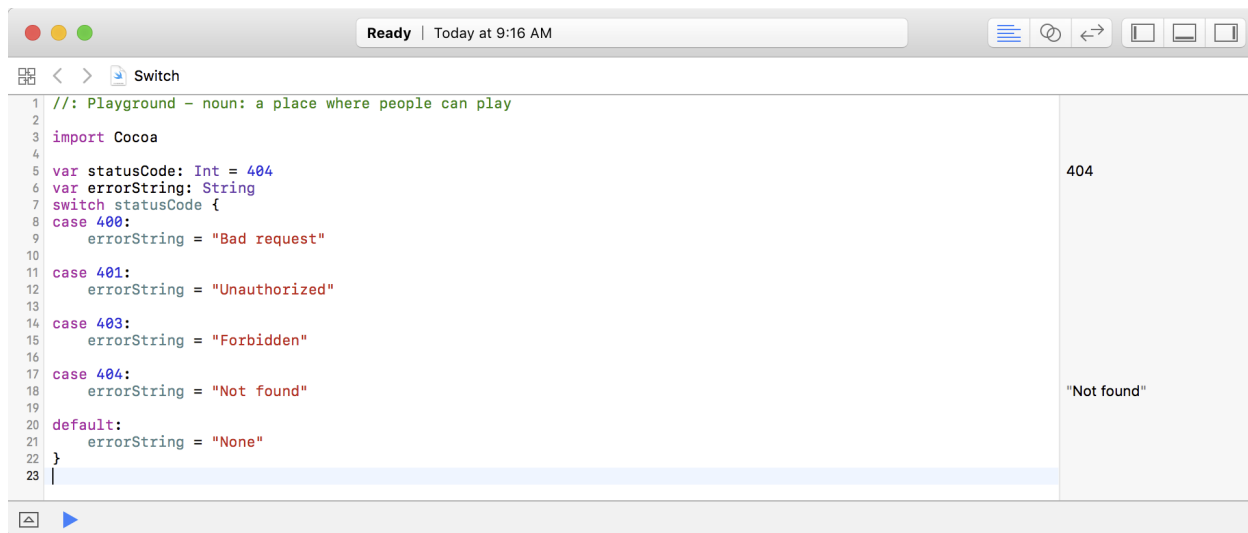


Suppose you want to use a `switch` statement to build up a meaningful error description. Update your code as shown.

## Listing 5.2  Switch cases can have multiple values

```
import Cocoa

var statusCode: Int = 404
var errorString: String = "The request failed:"
switch statusCode {
case 400:
    errorString = "Bad request"

case 401:
    errorString = "Unauthorized"

case 403:
    errorString = "Forbidden"

case 404:
    errorString = "Not found"

default:
    errorString = "None"
case 400, 401, 403, 404:
    errorString = "There was something wrong with the request."
    fallthrough
default:
    errorString += " Please review the request and try again."
}
```

There is now only one case for all of the error status codes (which are listed and separated by commas). If the `statusCode` matches any of the values in the case, the text `"There was something wrong with the request."` is given to the `errorString`.

You have also added a *control transfer statement* called `fallthrough`. Control transfer statements allow you to modify the order of execution in some control flow. These statements *transfer* control from one chunk of code to another. You will see another way to use control transfer statements in Chapter 6 on looping.

Here, `fallthrough` tells the `switch` statement to "fall through" the bottom of a case to the next one. If a matching case has a `fallthrough` control transfer statement at the end of it, it will first execute its code, and then transfer control to the case immediately below. That case will execute its code – whether or not it matches the value being checked against. If it also has a `fallthrough` statement at the end, it will hand off control to the case below, and so on. `fallthrough` statements allow you to enter a case and execute its code without having to match against it.

In this example, because of the `fallthrough` statement, the `switch` statement does not stop, even though the first case matches. Instead, it proceeds to the `default` case. Without the `fallthrough` keyword, the `switch` statement would have ended execution after the first match. The use of `fallthrough` in this example allows you to build up `errorString` without having to use strange logic that would guarantee that the comparison value matched all of the cases of interest.

The `default` case uses a compound-assignment operator (+=) to add a recommendation to review the request to the `errorString`. The end result of this `switch` statement is that `errorString` is set to `"There was something wrong with the request. Please review the request and try again."` If the status code provided had not matched the values in the case, the end result would have been `errorString` being set to `"The request failed: Please review the request and try again."`

If you are familiar with other languages like C or Objective-C, you will see that Swift's `switch` statement works differently. `switch` statements in those languages automatically fall through their cases. Those languages require a `break` control transfer statement at the end of the case's code to break out of the `switch`. Swift's `switch` works in the opposite manner. If you match on a case, the case executes its code and the `switch` stops running.

## Ranges

You have seen `switch` statements in which the cases have a single value to check against the comparison value and others in which the cases have multiple values. `switch` statements can also compare to a range of values using the syntax `valueX...valueY`. Update your code to see this in action.

Listing 5.3  Switch cases can have single values, multiple values, or ranges of values

```
import Cocoa

var statusCode: Int = 404
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 400, 401, 403, 404:
        errorString += " There was something wrong with the request."
        fallthrough
    default:
        errorString += " Please review the request and try again."
}

switch statusCode {
case 100, 101:
    errorString += " Informational, 1xx."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, 3xx."

case 400...417:
    errorString += " Client error, 4xx."

case 500...505:
    errorString += " Server error, 5xx."

default:
    errorString = "Unknown. Please review the request and try again."
}
```

The `switch` statement above takes advantage of the `...` syntax of *range matching* to create an inclusive range for categories of HTTP status codes. That is, `300...307` is a range that includes 300, 307, and everything in between.

You also have cases with a single HTTP status code (the second case) and with two codes explicitly listed and separated by a comma (the first case), as well as a default case. These are formed in the same way as the cases you saw before. All of the case syntax options can be combined in a `switch` statement.

The result of this `switch` statement is that `errorString` is set to equal `"The request failed with the error: Client error, 4xx."` Again, try changing the value of `statusCode` to see the other results. Be sure to set it back to 404 before continuing.

## Value binding

Suppose you want to include the actual numerical status codes in your `errorString`, whether the status code is recognized or not. You can build on your previous `switch` statement to include this information using Swift's *value binding* feature.

Value binding allows you to *bind* the matching value in a certain case to a local constant or variable. The constant or variable is available to use only within the matching case's body.

## Listing 5.4  Using value binding

```
...
switch statusCode {
case 100, 101:
    errorString += " Informational, 1xx."
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, 3xx."
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, 4xx."
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, 5xx."
    errorString += " Server error, \(statusCode)."

default:
    errorString = "Unknown. Please review the request and try again."

case let unknownCode:
    errorString = "\(unknownCode) is not a known error code."
}
```

Here you use string interpolation to pass statusCode into the errorString in each case.

Take a closer look at the last case. When the statusCode does not match any of the values provided in the cases above, you create a temporary constant, called unknownCode, binding it to the value of statusCode. For example, if the statusCode were set to be equal to 200, then your switch would set errorString to be equal to "200 is not a known error code." Because unknownCode takes on the value of any status code that does not match the earlier cases, you no longer need an explicit default case.

Note that by using a constant, you fix the value of unknownCode. If you needed to do work on unknownCode, for whatever reason, you could have declared it with var instead of let. Doing so would mean, for example, that you could then modify unknownCode's value within the final case's body.

This example shows you the syntax of value binding, but it does not really add much. The standard default case can produce the same result. Replace the final case with a default case.

Listing 5.5  Reverting to the default case

```
...
switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "\(statusCode) is not a known error code."
}
```

In the final case in Listing 5.4, you declared a constant with a value that was bound to the status code. This meant that the final case by definition matched everything that had not already matched a case in the switch statement. The switch statement was, therefore, exhaustive.

When you deleted the final case, the switch was no longer exhaustive. This means that you had to add a default case to the switch.

## where clauses

The code above is fine, but it is not great. After all, a status code of 200 is not *really* an error – 200 represents success! Therefore, it would be nice if your switch statement did not catch these cases.

To fix this, use a where clause to make sure unknownCode is not a 2xx indicating success. where allows you to check for additional conditions that must be met for the case to match and the value to be bound. This feature creates a sort of dynamic filter within the switch.

Listing 5.6  Using `where` to create a filter

```
import Cocoa

var statusCode: Int = 404
var statusCode: Int = 204
var errorString: String = "The request failed with the error:"

switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode where (unknownCode >= 200 && unknownCode < 300)
                                      || unknownCode > 505:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "\(statusCode) is not a known error code."
    errorString = "Unexpected error encountered."
}
```

Your case for `unknownCode` now specifies a range of status codes, which means that it is not exhaustive. This is not a problem because you already have a `default` case in the `switch`.

Without Swift's `fallthrough` feature, the `switch` statement will finish execution as soon as it finds a matching case and executes its body. When `statusCode` is equal to `204`, the `switch` will match at the second case and the `errorString` will be set accordingly. So, even though 204 is within the range specified in the `where` clause, the `switch` statement never gets to that clause.

Change `statusCode` to exercise the `where` clause and confirm that it works as expected.

## Tuples and pattern matching

Now that you have your `statusCode` and `errorString`, it would be helpful to pair those two pieces. Although they are logically related, they are currently stored in independent variables. A *tuple* can be used to group the two.

A tuple is a finite grouping of two or more values that are deemed by the developer to be logically related. The different values are grouped as a single, compound value. The result of this grouping is an ordered list of elements.

Create your first Swift tuple that groups the `statusCode` and `errorString`.

## Listing 5.7  Creating a tuple

```
import Cocoa

var statusCode: Int = 204
var statusCode: Int = 418
var errorString: String = "The request failed with the error:"

switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode where (unknownCode >= 200 && unknownCode < 300)
                                || unknownCode > 505:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "Unexpected error encountered."
}

let error = (statusCode, errorString)
```

You made a tuple by grouping `statusCode` and `errorString` within parentheses. The result was assigned to the constant `error`.

The elements of a tuple can be accessed by their index. You might have noticed `.0` and `.1` in the value of your tuple shown in the results sidebar – these are the elements' indices. Type in the following to access each element stored inside of the tuple.

## Listing 5.8  Accessing the elements of a tuple

```
...
let error = (statusCode, errorString)
error.0
error.1
```

You should see `418` and `"Unexpected error encountered."` displayed in the results sidebar for `error.0` (that is, the first element stored in the tuple) and `error.1` (the second element stored in the tuple), respectively.

Swift's tuples can also have named elements. Naming a tuple's elements makes for more readable code. It is not very easy to keep track of what values are represented by `error.0` and `error.1`. Named elements allow you to use easier-to-parse code like `error.code` and `error.error`.

Give your tuple's elements these more informative names.

## Listing 5.9  Naming the tuple's elements

```
...
let error = (statusCode, errorString)
error.0
error.1
let error = (code: statusCode, error: errorString)
error.code
error.error
```

Now you can access your tuple's elements by using their related names: code for statusCode and error for errorString. Your results sidebar should have the same information displayed.

You have already seen an example of pattern matching when you used ranges in the switch statement's cases. This form of pattern matching is called *interval matching* because each case attempts to match a given interval against the comparison value. Tuples are also helpful in matching patterns.

Imagine, for example, that you have an application that is making multiple web requests. You save the HTTP status code that comes back with the server's response each time. Later, you would like to see which requests, if any, failed with the status code 404 (the "requested resource not found" error). Using a tuple in the switch statement's cases enables you to match against very specific patterns.

Add the following code to create and switch on a new tuple.

Listing 5.10  Pattern matching in tuples

```
...
let error = (code: statusCode, error: errorString)
error.code
error.error

let firstErrorCode = 404
let secondErrorCode = 200
let errorCodes = (firstErrorCode, secondErrorCode)

switch errorCodes {
case (404, 404):
    print("No items found.")
case (404, _):
    print("First item not found.")
case (_, 404):
    print("Second item not found.")
default:
    print("All items found.")
}
```

You first add a few new constants. firstErrorCode and secondErrorCode represent the HTTP status codes associated with two different web requests. errorCodes is a tuple that groups these codes.

The new switch statement matches against several cases to determine what combination of 404s the requests might have yielded. The underscore (_) in the second and third cases is a wildcard that matches anything, which allows these cases to focus on a specific request's error code. The first case will match only if both of the requests failed with error code 404. The second case will match only if the first request failed with 404. The third case will match only if the second request failed with 404. Finally, if the switch has not found a match, that means none of the requests failed with the status code 404.

Because firstErrorCode did have the status code 404, you should see "First item not found." in the results sidebar.

## switch vs if/else

switch statements are primarily useful for comparing a value against a number of potentially matching cases. if/else statements, on the other hand, are better used for checking against a single condition. switches also offer a number of powerful features that allow you to match against ranges, bind values to local constants or variables, and match patterns in tuples – to name just a few features covered in this chapter.

Sometimes you might be tempted to use a switch statement on a value that could potentially match against any number of cases, but you really only care about one of them. For example, imagine checking an age constant of type Int looking for a specific demographic: ages 18-35. You might think writing a switch statement with a single case is your best option:

## Listing 5.11  Single-case `switch`

```
...
let age = 25
switch age {
case 18...35:
    print("Cool demographic")
default:
    break
}
```

age is a constant set to be equal to 25. It is possible that age could take on any value between 0 and 100 or so, but you are only interested in a particular range. The switch checks to see whether age is in the range from 18 to 35. If it is, then age is in the desired demographic and some code is executed. Otherwise, age is not in the target demographic and the default case matches, which simply transfers the flow of execution outside of the switch with the break control transfer statement.

Notice that you had to include a default case; switch statements have to be exhaustive. If this does not feel quite right to you, we agree. You do not really want to do anything here, which is why you used a break. It would be better to not have to write any code when you do not want anything to happen!

Swift provides a better way. In Chapter 3 you learned about if/else statements. Swift also has an if-case statement that provides pattern matching similar to what a switch statement offers.

## Listing 5.12  `if-case`

```
...
let age = 25
switch age {
case 18...35:
    print("Cool demographic")
default:
    break
}

if case 18...35 = age {
    print("Cool demographic")
}
```

This syntax is much more elegant. It simply checks to see whether age is in the given range. You did not have to write a default case that you did not care about. Instead, the syntax of the if-case allows you to focus on the single case of interest: whether age is in the range of 18 to 35.

if-cases can also include more complicated pattern matching, just as with switch statements. Say, for example, you wanted to know if age was greater than or equal to 21.

## Listing 5.13  `if-cases` with multiple conditions

```
...
let age = 25

if case 18...35 = age {
    print("Cool demographic")
}
if case 18...35 = age, age >= 21 {
    print("In cool demographic and of drinking age")
}
```

The new code above does the same as before, but adds something new. After the comma, it also checks to see whether age is 21 or greater. In the United States, this means that the person in question is also old enough to drink.

if-cases provide an elegant substitute for switch statements with only one condition. They also enjoy all of the pattern-matching power that make switch statements so wonderful. Use an if-case when you have only one case

in mind for a `switch` and you do not care about the `default` case. Because `if-case` statements are just `if/else` statements with improved pattern matching, you can also write the usual `else` block – but doing so would mean that you are effectively writing the `default` case and would detract from some of the `if-case`'s allure.

## Bronze Challenge

Review the `switch` statement below. What will be logged to the console? After you have decided, enter the code in a playground to see whether you were right.

```
let point = (x: 1, y: 4)

switch point {
case let q1 where (point.x > 0) && (point.y > 0):
    print("\(q1) is in quadrant 1")

case let q2 where (point.x < 0) && point.y > 0:
    print("\(q2) is in quadrant 2")

case let q3 where (point.x < 0) && point.y < 0:
    print("\(q3) is in quadrant 3")

case let q4 where (point.x > 0) && point.y < 0:
    print("\(q4) is in quadrant 4")

case (_, 0):
    print("\(point) sits on the x-axis")

case (0, _):
    print("\(point) sits on the y-axis")

default:
    print("Case not covered.")
}
```

## Silver Challenge

You can add more conditions to the `if-case` by supplying a comma-separated list. For example, you could check to see whether the person is: a) in the cool demographic, b) of drinking age in the United States, and c) not in their thirties. Add another condition to Listing 5.13 to check whether `age` meets all of these conditions.

# 6

# Loops

Loops help with repetitive tasks. They execute a set of code repeatedly, either for a given number of iterations or as long as a defined condition is met. Loops can save you from writing tedious and repetitive code, so take note! You will be using them a lot in your development.

In this chapter, you will use two sorts of loops:

- `for` loops

- `while` loops

`for` loops are ideal for iterating over the specific elements of an instance or collection of instances if the number of iterations to perform is either known or easy to derive. `while` loops, on the other hand, are well suited for tasks that execute repeatedly as long as a certain condition is met. Each type of loop has variations. Let's start with a `for-in` loop, which performs a set of code for each item in a specific range, sequence, or collection.

## for-in Loops

Create a new playground called `Loops`. Create a loop as shown.

Listing 6.1 A `for-in` loop

```
import Cocoa

var str = "Hello, playground"

var myFirstInt: Int = 0

for i in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
}
```

First, you declare a variable called `myFirstInt` that is an instance of **Int** and is initialized to be equal to `0`. Next, you create a for-in loop. Let's look at the components of the loop.

The `for` keyword signals that you are writing a loop. You next declare an *iterator* called `i` that represents the current iteration of the loop. The iterator is constant within the body of the loop and only exists here; it is also managed for you by the compiler.

In the first iteration of the loop, its value is the first value in the range of the loop. Because you used `...` to create an inclusive range of 1 through 5, the first value of `i` is 1. In the second iteration, the value of `i` is 2, and so on. You can think of `i` as being replaced with a new constant set to the next value in the range at the beginning of each iteration.

The code inside the braces (`{}`) is executed at each iteration of the loop. For each iteration, you increment `myFirstInt` by 1. After incrementing `myFirstInt`, you type the variable's name in the next line to expose its value

to the results sidebar. You then log this value to the console. These two steps – incrementing and logging – continue until i reaches the end of the range: 5. This loop is represented in Figure 6.1.

Figure 6.1  Looping over a range
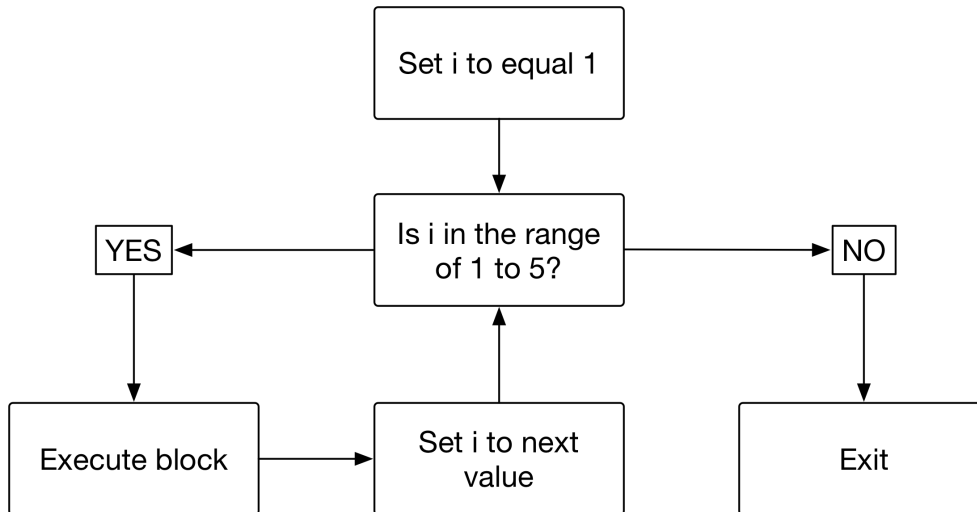


To see the results of your loop, find and click on the *results* button on the right edge of the results sidebar on the line with the code myFirstInt (Figure 6.2).

Figure 6.2  The results button
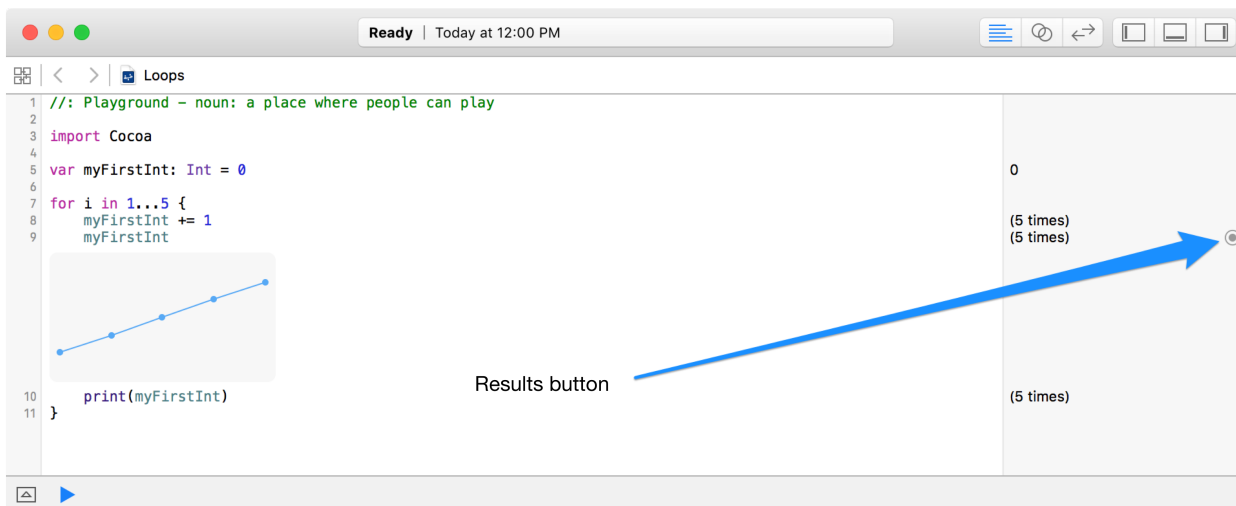


This opens a *results view* that displays the instance's value history inline with the code in the playground. You can grow or shrink the window of the graph by clicking and dragging its edges.

Move your mouse pointer into this new window and you will see that you can select individual points on this plot. For example, if you click the middle point, the playground will tell you that the value of this point is 3 (Figure 6.3).

Figure 6.3  Selecting a value on the plot
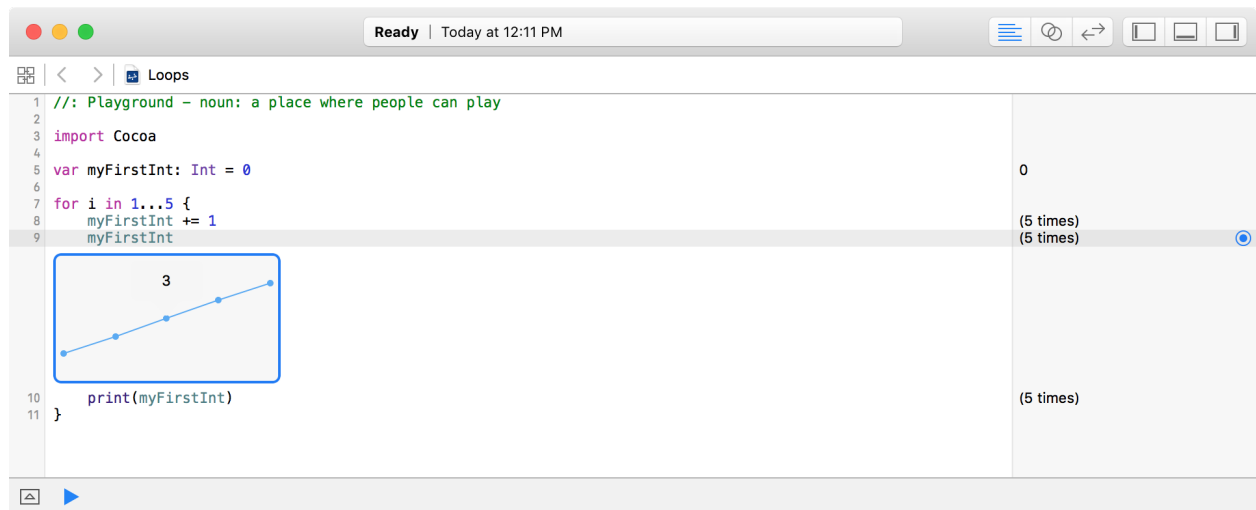


Because you declared i to be the iterator in the for-in loop, you can access i inside each iteration of the loop. Change your output to show the value of i at each iteration.

Listing 6.2  Printing the changing value of i to the console

```
...
for i in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
    print("myFirstInt equals \(myFirstInt) at iteration \(i)")
}
```

Instead of using an explicitly declared iterator, you can ignore it by using _. Replace your named constant with this wildcard and return your **print()** statement to its earlier implementation.

Listing 6.3  Replacing i with _

```
for i in 1...5 {
for _ in 1...5 {
    myFirstInt += 1
    myFirstInt
    print("myFirstInt equals \(myFirstInt) at iteration \(i)")
    print(myFirstInt)
}
```

This implementation of the for-in loop ensures that a specific operation occurs a set number of times. It does not check and report the value of the iterator in each pass of the loop over its range. You would typically use the explicit iterator i if you wanted to refer to that iterator within your loop's code block.

## where

Swift's for-in loop supports the use of where clauses similar to the ones you saw in Chapter 5. Using where allows for finer control over when the loop executes its code. The where clause provides a logical test that must be met in order to execute the loop's code. If the condition established by the where clause is not met, then the loop's code is not run.

For example, imagine that you want to write a loop that iterates over a range, but only executes its code when the loop's iterator is a multiple of 3.

Listing 6.4  A `for-in` loop with a `where` clause

```
...
for _ in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
}

for i in 1...100 where i % 3 == 0 {
    print(i)
}
```

As before, you create a local constant `i` that you can now use in the `where` clause's condition. Each integer in the range of 1 to 100 is bound to `i`. The `where` clause then checks to see whether `i` is divisible by 3. If the remainder is 0, the loop will execute its code. The result is that the loop will print out every multiple of 3 from 1 to 100.

Figure 6.4 demonstrates the flow of execution for this loop.

Figure 6.4  `where` clause loop diagram
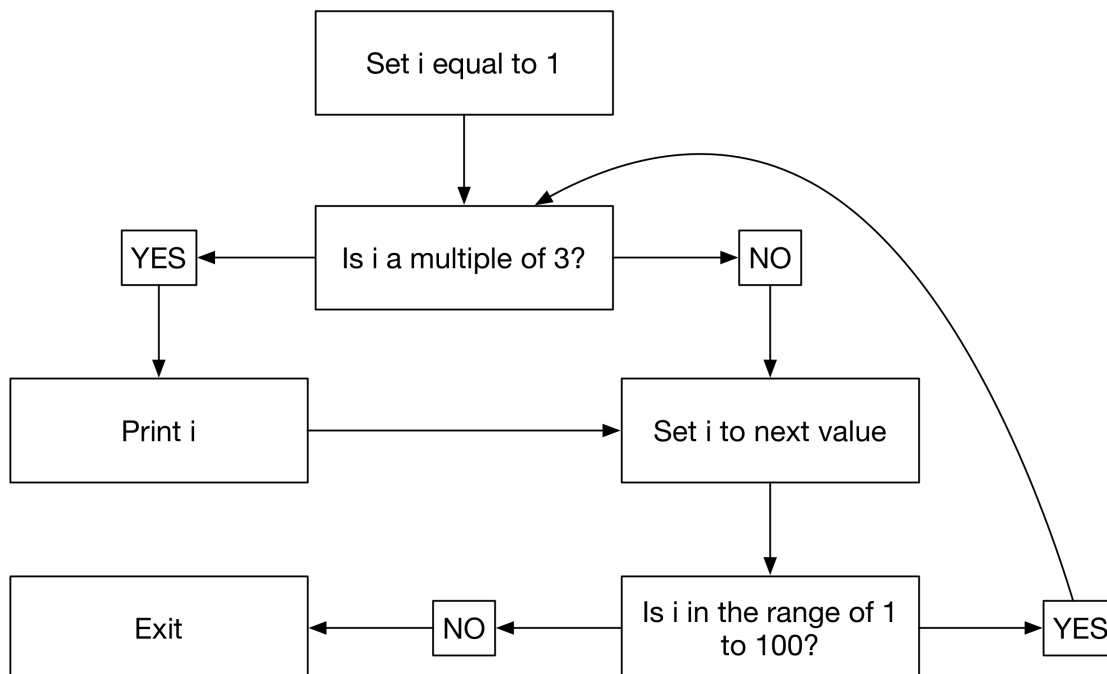


Imagine how you might accomplish this same result without the help of a `where` clause.

```
for i in 1...100 {
    if i % 3 == 0 {
        print(i)
    }
}
```

The above code does the same work as the loop with the `where` clause, but it is less elegant. There are more lines of code and there is a nested conditional within the loop. Generally speaking, we prefer fewer lines of code, so long as

the code is not overly complex to read. Swift's `where` clauses are very readable, so we typically choose this more concise solution.

# A Quick Note on Type Inference

Take a look at this code that you entered earlier:

```
for i in 1...5 {
    myFirstInt += 1
    print("myFirstInt equals \(myFirstInt) at iteration \(i)")
}
```

Notice that `i` is not declared to be of the **Int** type. It could be, as in `for i: Int in 1...5`, but an explicit type declaration is not necessary. The type of `i` is inferred from its context (as is the `let`). In this example, `i` is inferred to be of type **Int** because the specified range contains integers.

Type inference is handy: Because you will type less, you will make fewer typos. However, there are a few cases in which you need to specifically declare the type. We will highlight those when they come up. In general, however, we recommend that you take advantage of type inference whenever possible, and you will see many examples of it in this book.

# while Loops

A `while` loop executes the code inside its body so long as a condition is true. You can write `while` loops to do many of the same things you have seen in `for` loops above. For example, a `while` loop that replicates the `for` loop in Listing 6.1 can be expressed like so:

Listing 6.5  A `while` loop

```
...
var i = 1
while i < 6 {
    myFirstInt += 1
    print(myFirstInt)
    i += 1
}
```

Figure 6.5 shows the flow of execution in this code.

Figure 6.5  `while` loop diagram
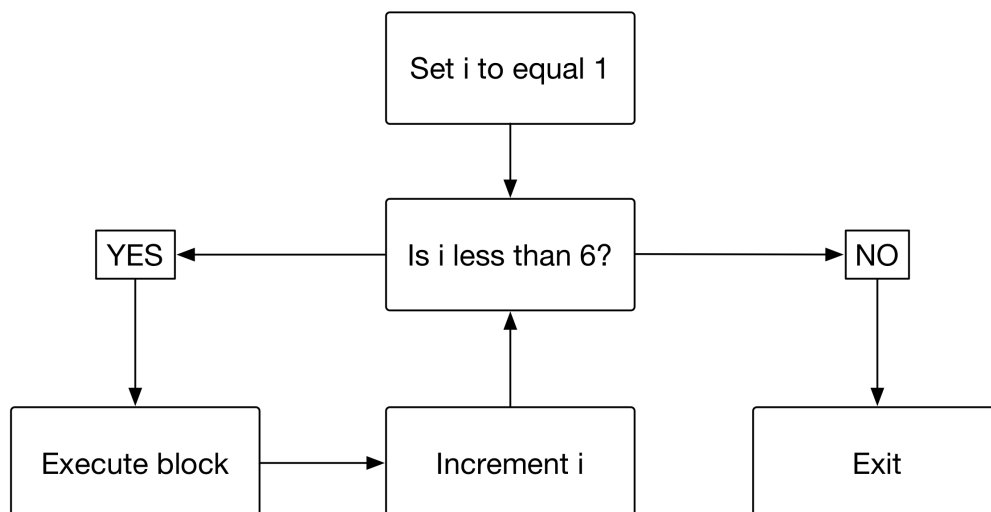
This `while` loop initializes an incrementer (`var i = 1`), evaluates a condition (`i < 6`), executes code if the condition is valid (`myFirstInt += 1`, `print(myFirstInt)`, increment `i`), and then returns to the top of the `while` loop to determine whether the loop should continue iterating.

`i` is declared as a variable because the condition you evaluate (`i < 6`) must be able to change. Remember, the `while` loop will run as long as the condition it checks is true. Thus, the condition for a `while` loop often checks some kind of state that will change at some point. Otherwise, if the state the condition examines never changes (or is always true), then the `while` loop will execute forever. Loops that never end are called *infinite loops*, and they are usually bugs.

`while` loops are best for circumstances in which the number of iterations the loop will pass through is unknown. For example, imagine a space shooter game with a spaceship that continuously fires its blasters as long as the spaceship has shields. Various external factors may lower or increase the ship's shields, so the exact number of iterations cannot be known. But if the shields have a value greater than 0, the blasters will keep shooting. The code snippet below illustrates a simplified implementation of this idea.

```
while shields > 0 {
    // Fire blasters!
    print("Fire blasters!")
}
```

## repeat-while Loops

Swift also supports a type of `while` loop called the `repeat-while` loop. The `repeat-while` loop is called a `do-while` loop in other languages. The difference between `while` and `repeat-while` loops is *when* they evaluate their condition. The `while` loop evaluates its condition before stepping into the loop. This means that the `while` loop may not ever execute, because its condition could be `false` when it is first evaluated. The `repeat-while` loop, on the other hand, executes its loop at least once and *then* evaluates its condition. The syntax for the `repeat-while` loop demonstrates this difference.

```
repeat {
    // Fire blasters!
    print("Fire blasters!")
} while shields > 0
```

In this `repeat-while` version of the space shooter game, the code block that contains the line `print("Fire blasters!")` is executed first. Then the `repeat-while` loop's condition is evaluated to determine whether the loop should continue iterating. Thus, the `repeat-while` loop ensures that the spaceship fires its blasters at least one time.

The `repeat-while` loop avoids a somewhat depressing scenario: What if the spaceship is created and, by some freak accident, immediately loses all of its shields? Perhaps it spawns in front of an oncoming asteroid. It would not even get to fire a shot. That would be a pretty poor user experience. A `repeat-while` loop ensures that the blasters fire at least once to avoid this anticlimactic scenario.

## Control Transfer Statements, Redux

Let's revisit control transfer statements in the context of loops. Recall from Chapter 5 (where you used `fallthrough` and `break`) that control transfer statements change the typical order of execution. In the context of a loop, you can control whether execution iterates to the top of the loop or leaves the loop altogether.

Let's elaborate on the space shooter game to see how this works. You are going to use the `continue` control transfer statement to stop the loop where it is and begin again from the top.

Listing 6.6  Using `continue`

```
...
var shields = 5
var blastersOverheating = false
var blasterFireCount = 0
while shields > 0 {

    if blastersOverheating {
        print("Blasters are overheated! Cooldown initiated.")
        sleep(5)
        print("Blasters ready to fire")
        sleep(1)
        blastersOverheating = false
        blasterFireCount = 0
    }

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    // Fire blasters!
    print("Fire blasters!")

    blasterFireCount += 1
}
```

You are adding a good bit of code, so let's break it down. First, you added some variables to keep track of certain information about the spaceship:

- `shields` is of type **Int**, keeps track of the shield strength, and is initialized to be equal to 5.

- `blastersOverheating` is a **Bool** initialized to `false` that keeps track of whether the blasters need time to cool down.

- `blasterFireCount` is of type **Int** and keeps track of the number of shots the spaceship has fired (which determines whether the blasters are overheating).

After creating your variables, you wrote two `if` statements, both contained in a `while` loop with a condition of `shields > 0`. The first `if` statement checks whether the blasters are overheating, and the second checks the fire count. For the first, if the blasters are overheating, a number of code steps execute. You log information to the console and the **sleep()** function tells the system to wait for 5 seconds, which models the blasters' cooldown phase. You next log that the blasters are ready to fire again, wait for 1 more second (simply because it makes it easier to see what logs to the console next), set `blastersOverheating` to be equal to `false`, and also reset `blasterFireCount` to `0`.

With shields intact and blasters cooled down, the spaceship is ready to fire away.

The second `if` statement checks whether `blasterFireCount` is greater than 100. If this conditional evaluates to `true`, you set the Boolean for `blastersOverheating` to be `true`. At this point, the blasters are overheated, so you need a way to jump back up to the top of the loop so that the spaceship does not fire. You use `continue` to do this. Because the spaceship's blasters have overheated, the conditional in the first `if` statement will evaluate to `true`, and the blasters will shut down to cool off.

If the second conditional is evaluated to be `false`, you log to the console as before. Next, you increment the `blasterFireCount` by one. After you increment this variable, the loop will jump back up to the top, evaluate the condition, and either iterate again or hand off the flow of execution to the line immediately after the closing brace of the loop. Figure 6.6 shows this flow of execution.

Figure 6.6 `while` loop diagram

Are shields > 0?

YES

NO

Are blasters overheating?

Exit/Do not execute loop

YES

NO

- Enter cooldown phase
- Reset fire count

Is fire count > 100?

YES

NO

Set blasters to overheating

- Fire blasters
- Increment blaster fire count
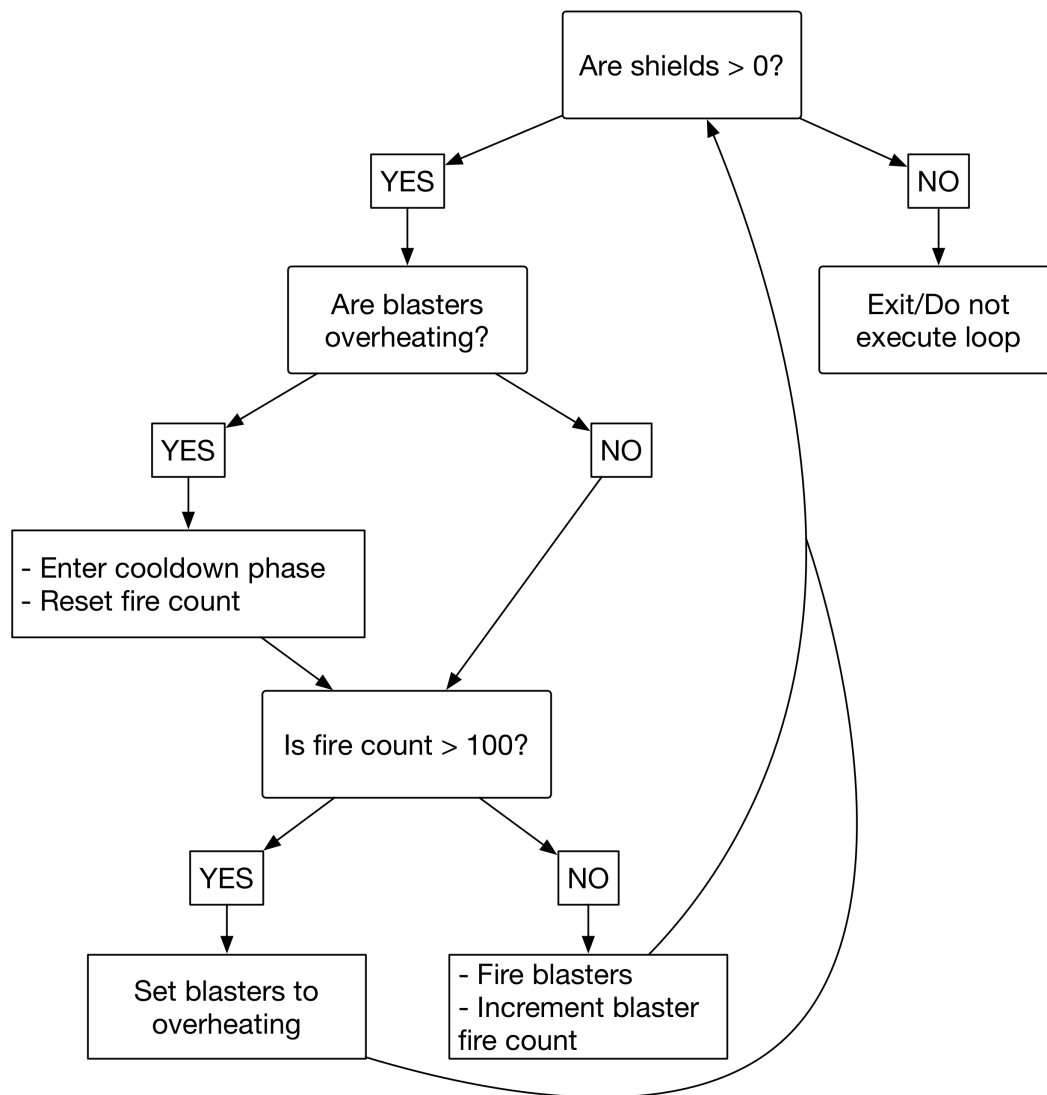
Note that this code will execute indefinitely. There is nothing to change the value of `shields`, so `while shields > 0` is always satisfied. If nothing changes, and your computer has enough power to run forever, this loop will continue to execute. This is what we call an *infinite loop*.

But all games must come to an end. Let's say that the game is over when the user has destroyed 500 space demons. To exit the loop, you will use the `break` control transfer statement.

## Listing 6.7  Using `break`

```
...
var shields = 5
var blastersOverheating = false
var blasterFireCount = 0
var spaceDemonsDestroyed = 0
while shields > 0 {

    if spaceDemonsDestroyed == 500 {
        print("You beat the game!")
        break
    }

    if blastersOverheating {
        print("Blasters are overheated! Cooldown initiated.")
        sleep(5)
        print("Blasters ready to fire")
        sleep(1)
        blastersOverheating = false
        blasterFireCount = 0
        continue
    }

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    // Fire blasters!
    print("Fire blasters!")

    blasterFireCount += 1
    spaceDemonsDestroyed += 1
}
```

Here, you add a new variable called `spaceDemonsDestroyed`, which is incremented each time the blasters fire. (You are a pretty good shot, apparently.) Next, you add a new `if` statement that checks whether the `spaceDemonsDestroyed` variable is equal to 500. If it is, you log victory to the console.

Note the use of `break`. The `break` control transfer statement will exit the `while` loop, and execution will pick up on the line immediately after the closing brace of the loop. This makes sense: If the user has destroyed 500 space demons and the game is won, the blasters do not need to fire anymore.

# Silver Challenge

Fizz Buzz is a game used to teach division. Create a version of the game that works like this: For every value in a given range, print out "FIZZ" if the current number is evenly divisible by 3. If the number is evenly divisible by 5, print out "BUZZ." If the number is evenly divisible by both 3 and 5, then print out "FIZZ BUZZ." If the number is not evenly divisible by 3 or 5, then simply print out the number.

For example, over the range of 1 through 10, playing Fizz Buzz should yield this: "1, 2, FIZZ, 4, BUZZ, FIZZ, 7, 8, FIZZ, BUZZ."

Computers love to play Fizz Buzz. The game is perfect for loops and conditionals. Loop over the range from 0 through 100 and print "FIZZ," "BUZZ," or "FIZZ BUZZ" appropriately for each number in the range.

For bonus points, solve Fizz Buzz with both an `if/else` conditional and a `switch` statement. When using the `switch`, make sure to match against a tuple in its various cases.

# 7
# Strings

In programming, textual content is represented by strings. You have seen and used strings already. `"Hello, playground"`, for example, is a string that appears at the top of every newly created playground. Like all strings, it can be thought of as an ordered collection of characters. In reality, Swift strings are not themselves collections, but they do provide a variety of views into their underlying contents that are collections. In this chapter, you will see more of what strings can do.

## Working with Strings

In Swift, you create strings with the **String** type. Create a new playground called Strings and add the following new instance of the **String** type.

Listing 7.1  Hello, playground

```
import Cocoa

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

You have created a **String** instance named `playground` using the string literal syntax, which encloses a sequence of text with quotation marks.

This instance was created with the `let` keyword, making it a constant. Recall that being a constant means that the instance cannot be changed. If you do try to change it, the compiler will give you an error.

Create a new string, but make this instance mutable.

Listing 7.2  Creating a mutable string

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
```

`mutablePlayground` is a mutable instance of the **String** type. In other words, you can change the contents of this string. Use the addition and assignment operator to add some final punctuation.

Listing 7.3  Adding to a mutable string

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"
```

Take a look at the results sidebar on the righthand side of the playground. You should see that the instance has changed to `"Hello, mutable playground!"`

The characters that comprise Swift's strings are of the **Character** type. You use Swift's **Character** type to represent Unicode characters, and in combination **Character**s form a **String** instance.

Loop through the `mutablePlayground` string to see the **Character** type in action.

### Listing 7.4 mutablePlayground's **Character**s

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}
```

This loop iterates through every **Character** c in `mutablePlayground`. In it, you access the `characters` property of the **String** `mutablePlayground`. Do not worry about what a property is right now; this topic will be covered in detail in Chapter 16. For now, all you need to know is that a property is a way a type holds on to data. In Swift, you access properties via *dot syntax*, as in `mutablePlayground.characters`.

The `characters` property represents the collection of characters that make up the instance. Each iteration of the loop logs one of the **String**'s characters to the console. Every character is logged to the console on its own line because `print()` prints a line break after logging its content.

Your output should look like Figure 7.1.

### Figure 7.1  Logging characters in a string
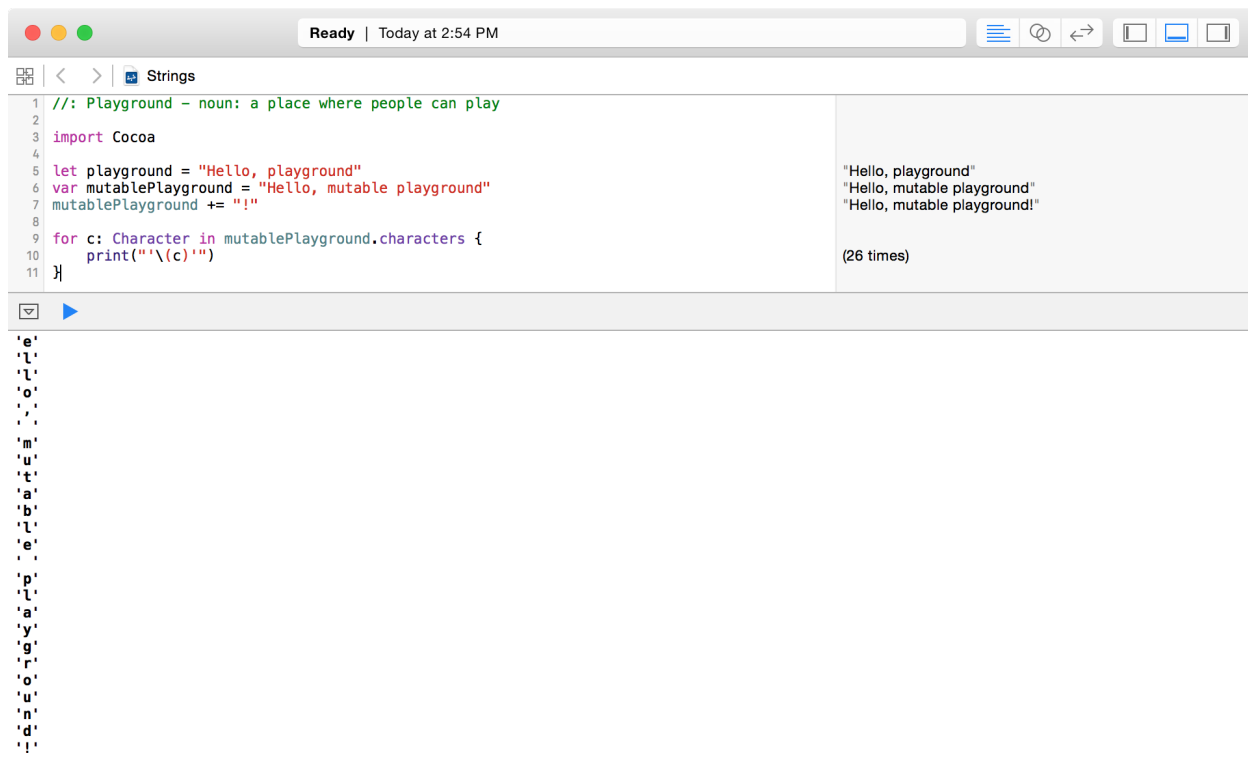


# Unicode

Unicode is an international standard that encodes characters so they can be seamlessly processed and represented regardless of the platform. Unicode represents human language (and other forms of communication like emoji) on computers. Every character in the standard is assigned a unique number.

Swift's **String** and **Character** types are built on top of Unicode and they do the majority of the heavy lifting. Nonetheless, it is good to have an understanding of how these types work with Unicode. Having this knowledge will likely save you some time and frustration in the future.

## Unicode scalars

At their heart, strings in Swift are composed of *Unicode scalars*. Unicode scalars are 21-bit numbers that represent a specific character in the Unicode standard. For example, U+0061 represents the Latin small letter "a." U+1F60E represents the smiley-faced emoji with sunglasses. The text U+1F60E is the standard way of writing a Unicode character. The 1F60E portion is a number written in hexadecimal, or base 16.

Create a constant to see how to use specific Unicode scalars in Swift and the playground.

Listing 7.5  Using a Unicode scalar

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
```

This time, you used a new syntax to create a string. The quotation marks are familiar, but what is inside them is not a string literal, as you have seen before. It does not match the results in the sidebar (Figure 7.2).

Figure 7.2  Emoji in the sidebar
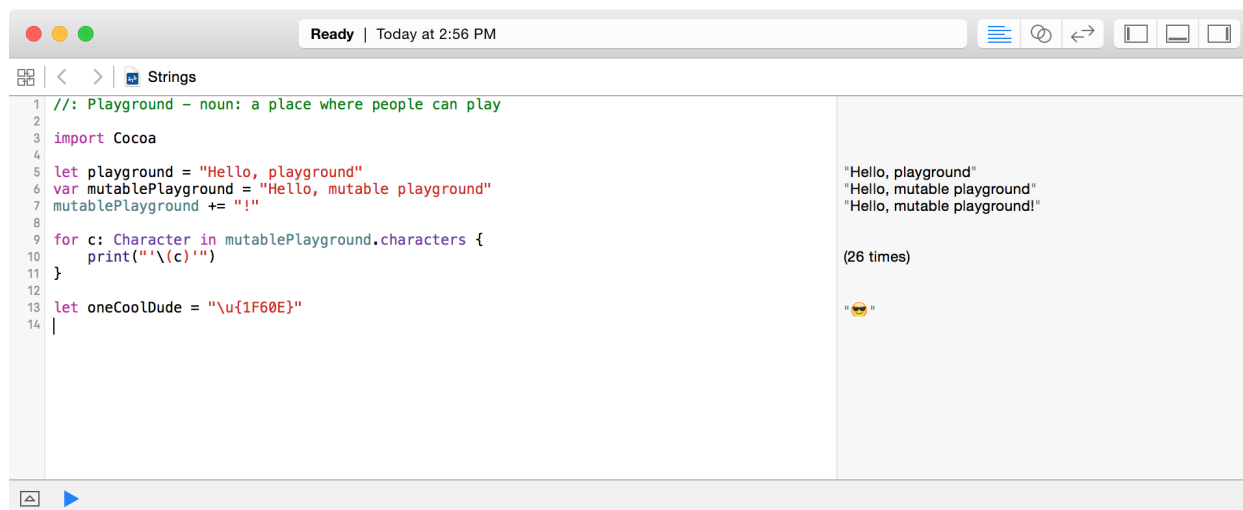


The \u{} syntax represents the Unicode scalar whose hexadecimal number appears between the braces. In this case, oneCoolDude is assigned to be equal to the character representing the sunglasses-wearing emoji.

How does this relate to more familiar strings? It turns out that every string in Swift is composed of Unicode scalars. So why do they look unfamiliar? To explain, we need to discuss a few more concepts.

Every character in Swift is built up from one or more Unicode scalars. One Unicode scalar maps onto one fundamental character in a given language. But we say that characters are built from "one or more" Unicode scalars

because there are also *combining scalars*. For example, `U+0301` represents the Unicode scalar for the combining acute accent: ´. This scalar is placed on top of – that is, combined with – the character that precedes it. You can use this scalar with the Latin small letter "a" to create the character á.

## Listing 7.6  Using a combining scalar

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
let aAcute = "\u{0061}\u{0301}"
```

You should see á, the combination of the letter "a" and the acute accent, in the results sidebar.

Every character in Swift is an *extended grapheme cluster*. Extended grapheme clusters are sequences of one or more Unicode scalars that combine to produce a single human-readable character. Just now, you decomposed the character á into its two constituent Unicode scalars: the letter and the accent. Making characters extended grapheme clusters gives Swift flexibility in dealing with complex script characters.

Swift also provides a mechanism to see all of the Unicode scalars in a string. For example, you can see all of the Unicode scalars that Swift uses to create the instance of **String** named `playground` that you created earlier using the `unicodeScalars` property, which holds all of the scalars that Swift uses to make the string. Add the following code to your playground to see `playground`'s Unicode scalars.

## Listing 7.7  Revealing the Unicode scalars behind a string

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}
```

You should see the following output in the console: `72 101 108 108 111 44 32 112 108 97 121 103 114 111 117 110 100`. What do all of these numbers mean?

Recall that the `unicodeScalars` property holds on to data representing all of the Unicode scalars used to create the string instance `playground`. Each number on the console corresponds to a Unicode scalar representing a single character in the string. But they are not the hexadecimal Unicode numbers. Instead, each is represented as an unsigned 32-bit integer. The first, `72`, corresponds to the Unicode scalar value of `U+0048`, or an uppercase "H."

# Canonical equivalence

While there is a role for combining scalars, Unicode also provides already combined forms for some common characters. For example, there is a specific scalar for á. You do not actually need to decompose it into its two parts: the letter and the accent. The scalar is `U+00E1`. Create a new constant string that uses this Unicode scalar.

### Listing 7.8  Using a precomposed character

```
...
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"
```

As you can see, `aAcutePrecomposed` appears to have the same value as `aAcute`. Indeed, if you check to see if these two characters are the same, you will find that Swift answers "yes."

### Listing 7.9  Checking equivalence

```
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

let b = (aAcute == aAcutePrecomposed) // true
```

`aAcute` was created using two Unicode scalars, and `aAcutePrecomposed` only used one. Why does Swift say that they are equivalent? The answer is *canonical equivalence*.

Canonical equivalence refers to whether two sequences of Unicode scalars are the same *linguistically*. Two characters, or two strings, are considered equal if they have the same linguistic meaning and appearance, regardless of whether they are built from the same Unicode scalars. `aAcute` and `aAcutePrecomposed` are equal strings because both represent the Latin small letter "a" with an acute accent. The fact that they were created with different Unicode scalars does not affect this.

## Counting elements

Canonical equivalence has implications for counting elements of a string. You might think that `aAcute` and `aAcutePrecomposed` would have different character counts. Write the following code to check.

### Listing 7.10  Counting characters

```
...
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

let b = (aAcute == aAcutePrecomposed) // true
print("aAcute: \(aAcute.characters.count);
       aAcutePrecomposed: \(aAcutePrecomposed.characters.count)")
```

You use the `count` property on `characters` to determine the character count of these two strings. `count` iterates over a string's Unicode scalars to determine its length. The results sidebar reveals that the character counts are the same: Both are 1 character long. Canonical equivalence means that whether you use a combining scalar or a precomposed scalar, the result is treated as a single character.

## Indices and ranges

Because strings can be thought of as ordered collections of characters, you might think that you can access a specific character on a string like so:

```
let index = playground[3] // 'l'???
```

The code `playground[3]` uses the subscript syntax. In general, the brackets (`[]`) indicate that you are using a subscript in Swift. Subscripts allow you to retrieve a specific value within a collection.

`3` is an *index* that is used to find a particular element within a collection. The code above suggests that you are trying to select the fourth character from the collection of characters making up the `playground` string (the first index is `0`). You will learn more about subscripts below and will also see them in action in Chapter 9 and Chapter 10 on arrays and dictionaries.

If you tried to use a subscript like this, you would get an error: `"'subscript' is unavailable: cannot subscript String with an Int."` The Swift compiler will not let you access a specific character on a string via a subscript index. This limitation has to do with the way Swift strings and characters are stored. You cannot index a string with an integer because Swift does not know which Unicode scalar corresponds to a given index without stepping through every preceding character. This operation can be expensive. Therefore, Swift forces you to be more explicit.

Swift uses a type called **`String.CharacterView.Index`** to keep track of indices. Do not worry about all of the periods (`.`) in **`String.CharacterView.Index`**; they just mean that **`Index`** is a type that is defined on **`CharacterView`**, which is a type defined on **`String`**. (You will read more about nested types in Chapter 16.) The type **`CharacterView`** is responsible for giving a view of a string's characters as an ordered collection.

As you have seen in this chapter, an individual character may be made up of multiple Unicode code points. It is the job of the **`CharacterView`** to represent each of these code points as a single **`Character`** instance and to combine these characters into the correct string.

It is convenient to have **`Index`** defined on **`CharacterView`**. This allows you to ask the character view to hand back indices that are meaningful for the string in question. For example, to find the character at a particular index, you begin with the **`String`** type's `startIndex` property. This property yields the starting index of a string as a **`String.CharacterIndex.Index`**. You can then use this starting point in conjunction with the **`index(_:offsetBy:)`** method to move forward until you arrive at the position of your choosing. (A *method* is like a function; you will learn more about them in Chapter 12.)

Say you want to know the fifth character of the `playground` string that you created at the beginning of this chapter.

## Listing 7.11  Finding the fifth character

```
...
let start = playground.startIndex
let end = playground.index(start, offsetBy: 4)
let fifthCharacter = playground[end] // "o"
```

You used the `startIndex` property on the string to get the first index of the string. This property yields an instance of type **`String.CharacterView.Index`**. Next, you used the **`index(_:offsetBy:)`** method to advance from the starting point to your desired position. The `offsetBy` parameter takes an argument of type **`Int`**, which the method then adds to the first index. You passed in 4 to represent the fifth character.

The result of calling **`index(_:offsetBy:)`** is a **`String.CharacterView.Index`** that you assigned to the constant end. Finally, you used end to subscript your `playground` string instance, which resulted in the character "o" being assigned to `fifthCharacter`. (`playground`, remember, is set to equal `"Hello, playground"`.)

Ranges, like indices, depend upon the **`String.CharacterView.Index`** type. Imagine that you wanted to grab the first five characters of `playground`. You can use the same start and end constants.

## Listing 7.12  Pulling out a range

```
...
let start = playground.startIndex
let end = playground.index(start, offsetBy: 4)
let fifthCharacter = playground[end] // "o"
let range = start...end
let firstFive = playground[range] // "Hello"
```

The result of the syntax `start...end` is called a *closed range* of type **String.CharacterView.Index**, which works similarly to the range you saw in Chapter 6 (`1...5`). You used this new range as a subscript on the `playground` string. The subscript grabbed the first five characters from `playground`, making `firstFive` a constant equal to `"Hello"`.

## Bronze Challenge

Create a new **String** instance called `empty` and give it an empty string: `let empty = ""`. It is useful to be able to tell if a string has any characters in it. Use the `startIndex` and `endIndex` properties on `empty` to determine whether this string is truly empty. Next, consult the documentation via the Help menu item in Xcode for a cleaner way to accomplish this check.

## Silver Challenge

Replace the `"Hello"` string with an instance created out of its corresponding Unicode scalars. You can find the appropriate codes on the internet.

# Index

## Symbols

# Index