Table of Contents

Part I

Getting Started

1
Introduction

Apple's World Wide Developer's Conference (WWDC) is an annual landmark event for its developer community, but 2014 was a particularly special year. This was the year that Apple introduced an entirely new language called Swift for the development of iOS and OS X applications. This book aims to introduce the Swift programming language to individuals interested in developing applications for Apple hardware as well as serve as a resource for later reference.

So, what about Objective-C, Apple's previous *Lingua Franca* for its platforms? Do you still need to know that language? For the time being, we think that answer is an unequivocal "Yes." For example, Apple's `Cocoa` library, one that you will use extensively, is written in Objective-C, and so debugging will be made easier if you understand that language. Moreover, the bulk of learning materials and extant Mac and iOS apps are written in Objective-C. As an iOS or Mac developer, you'll be likely to encounter Objective-C, and so it will make sense to be familiar with the language. Indeed, Apple has made it easy, and sometimes preferable, to mix and match Objective-C with Swift in the same project.

But do you need to know Objective-C to learn Swift? No, but it won't hurt you! Indeed, Swift looks, feels, and behaves very similarly to Objective-C. If you know Objective-C, then Swift will feel familiar. If you don't know Objective-C, then some of Swift's new features and patterns won't feel as transcendentally awesome to you. Either way, you're in good position to learn the language.

Goals of the Book

We've written this book for all types of iOS and Mac OS X developers, whether you're a platform expert, or if this is your first time encountering Apple technologies. For those of you just starting software development, we will consciously 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 the more experienced developers, we believe this book will serve as a helpful introduction to your platform's new language.

We have also written this book with numerous examples such that you can use this book as a touchpoint to refer to in the future during your times of need. So, instead of focusing solely on abstract concepts and issues particular to theory, we've written in favor of the practical. That doesn't mean we won't be discussing some more complicated topics. On the contrary, our approach favors using concrete examples to unpack the more difficult ideas and to also expose the best practices that make code more readable, easier to maintain, and fun to write.

We hope to show you how fun it can be to make applications for the Apple ecosystem. While writing code can be extremely frustrating, it can also be even more gratifying. There is something magical and exhilirating about solving a problem, not to mention the special joy that comes out of making an app that helps people and brings them happiness. In the end, it's our wish to teach you how to write good code and enjoy yourself while doing it.

How to Use this Book

Programming can be tough, and this book is here to make it easier. How can we help you with that? Follow these steps:

- Read the book! Really! Don't just browse it nightly before going to bed.
- Type out the examples as you read along. Part of what you're learning here is muscle memory. If your fingers know where to go and what to type without you thinking very much, then you are on your way to becoming a more effective developer.

- Make mistakes! In our experience, the fastest way to learn how to make things work is to first figure out what makes them not work. Break our code examples and then make them work again.
- Experiment to the extent that 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 better developer you will become faster.
- Do the challenges. As we mentioned above, it's important to begin solving problems with Swift as soon as possible. Doing so will help you to start thinking like a developer.

The best way to improve at anything is with practice. Sure, some people have a knack for one thing or the other, but that will never take the place of hard work. If you want to be a developer, then let's get started! If you find that you don't think you're very good at it, then who cares?! Keep at it and we're sure that you'll surprise yourself. Either way, your next steps lie ahead. Onward!

2
Getting Started

Let's get started learning Swift! In this chapter, you will get your environment set up and take a small tour of some of the tools you will use every day as an iOS and Mac developer. Additionally, you will get your hands dirty with a little bit of code to get better acquainted with Swift and Xcode. Doing so will involve the following items:

- Downloading the developer tools
- Getting started with Xcode
- Creating a Playground to try out some code

Downloading the Developer Tools

In order for you to work with Swift, you must download and install Apple's *Integrated Development Environment* (IDE) called Xcode. Xcode has all that you need to make iOS and Mac apps. Make sure to download Xcode 6 or higher. Xcode 6 requires Mac OS X 10.9.3 or higher. Accordingly, this means that you'll need a Mac to work through this book. If you don't have Xcode already, you can download the application through the Mac App Store for free.

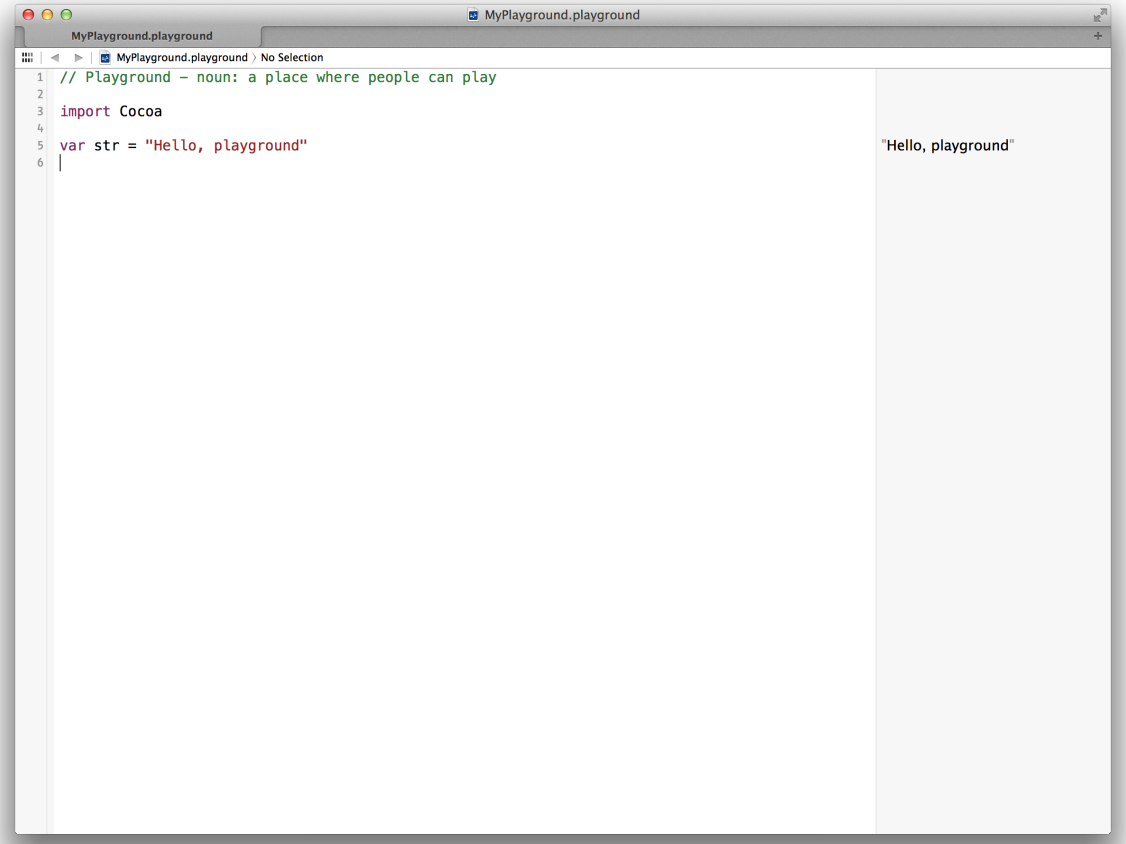
Getting Started with Xcode

Now that you have Xcode installed, launch it and you will see a window presenting you with a number of project templates to help you get started. Each template creates a project that is well-suited for a specific sort of program. For now, select "Get started with a playground".

Figure 2.1 Choosing the Playground Template

After you click on the Playground template, you will see a window asking you what you would like to name the project and where you would like to save it. Name and save the project what and where you would like. To help keep you organized, it is a good idea to make a folder in a convenient location and save all of your work there. You will also be asked to choose a platform (OS X or iOS). Select OS X, even if you're an iOS developer. The Swift features you'll be covering are common to both platforms.

Figure 2.2 Your New Playground



playgrounds are a new feature released in Xcode 6 that provide an interactive environment for rapidly developing and evaluating Swift code. A `playground` does not require that you compile and run a complete project. Furthermore, playgrounds evaluate your Swift code on the fly. Thus, they are ideal for testing and experimenting with the Swift language in a lightweight environment. You will be using playgrounds frequently throughout this book when you want quick feedback on your Swift code.

You will be using both playgrounds and native command-line tools to introduce you to Swift. Why not just use playgrounds? You'd miss out on a lot of Xcode's features and won't get as much exposure to the IDE. Trust us that you'll be spending a lot of time in Xcode, and it's good to get comfortable with it as soon as possible.

Playing in a Playground

As you can see in Figure 2.2, Swift playgrounds are primarily divided into two sections. On the left, you have the Swift code editor, and on the right, you have the results sidebar. The code in the editor is evaluated and run, if possible, every time the source changes. The results of the code is displayed on the right in the results sidebar.

On the very first line, you will likely notice that the color of the text is green, and is prefixed by two forward slashes: `//`. Lines that are prefixed by two forward slashes signify to the compiler that they are merely comments for the developer's edification. You typically use comments as inline documentation or developer's notes. If you remove those forward slashes, you'll see that the color of the text changes and the compiler will issue warnings to the effect that it cannot parse the expression. Incidentally, you can add and remove comments for a given selection by tapping "command-forward slash" on your keyboard. Go ahead, highlight the code in your playground. You should notice that the editor toggles comments in and out when you use the above keyboard combination.

You will also notice that the playground imports the Cocoa framework at very the top of the code editor. This import means that our playground has complete access to all of the Application Programming Interfaces (APIs)

made available to iOS and Mac developers by the Cocoa framework. If you're not familiar, an API is similar to prescription, or set of definitions, for how a program can be written.

In the code editor below the import, there is a line that reads: `var str = "Hello, playground"`. On the right in the results sidebar, there should read the text: “Hello, playground”. Let's break down that line of code to better understand what is going on. On the left hand side of the equals sign, you have the text `var str`. `var` is a keyword in Swift that means you are declaring a variable, which is an important concept that you will see in greater detail in the next chapter. For now, let's just say that a variable represents some value that you expect to change, or vary. On the right hand side of the equality, you have the text: `"Hello, playground"`.

In Swift, the quotations mean that you are making a `String`. `Strings` are used to represent an ordered collection of characters. The template named this new variable `str`, but you could have named this variable just about anything (There are limitations to this, of course. Try to change the name `str` to be `var`. What happens? Why do you think you cannot name your variable `var`?). Thus, you are now in a position to understand the text printed on the right in the results sidebar; it's the string value you just assigned to the variable `str`.

Varying Variables and Printing to the Console

In the section above, you created a variable named `str` and assigned to it some text representing an instance of the `String` type. Types describe a particular structure for representing data and define what the type can and cannot do with that 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.

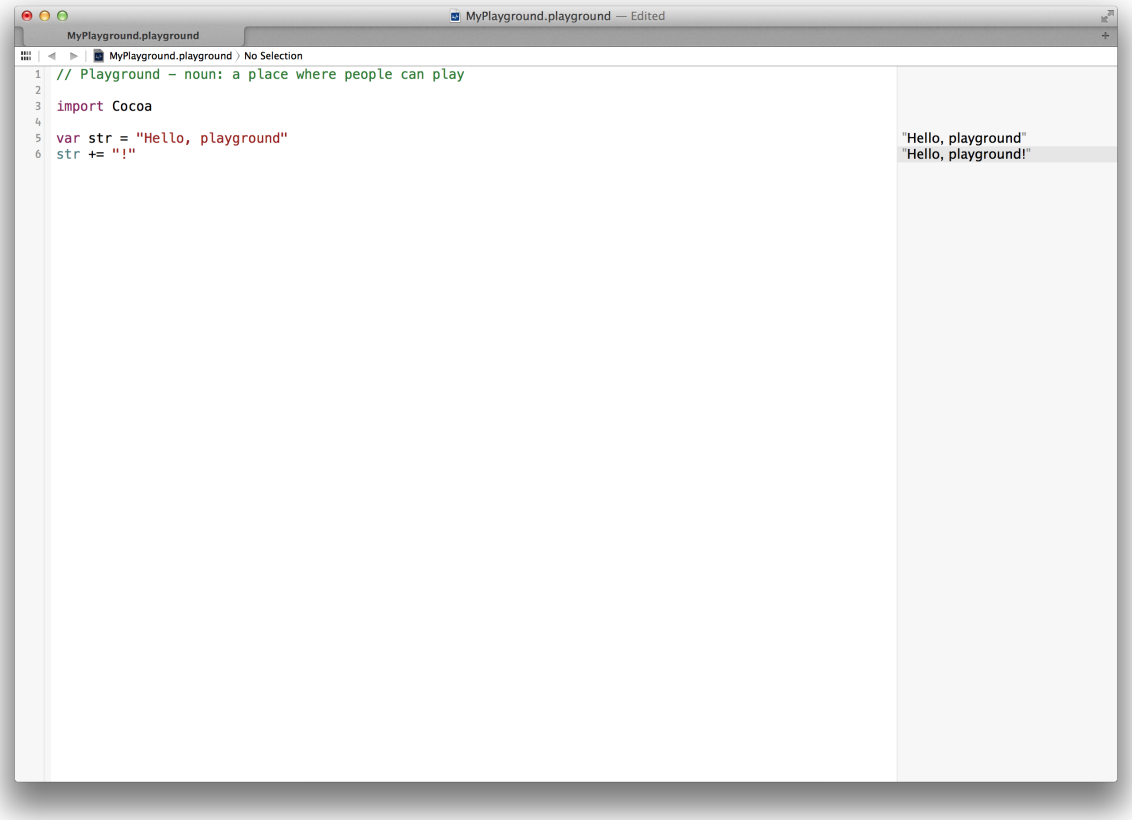
To make this more clear, recall that you created `str` above as a *variable*. This means that you can change the variable; i.e., you can vary the value for `str`. Append some punctuation to the end of the string that you have already made to make it more like a regular sentence. As a note, whenever we refer to a previous code listing and add new code, we will do so by listing the previous code along with the new code highlighted in bold font.

```
import Cocoa

var str = "Hello, playground"

str += "!"
```

Great! You have now appended an exclamation point to our string instance. To do so, you used the `+=` *compound assignment operator* for addition and assignment. The *addition assignment operator* allows us to combine the addition and assignment operations into one operator. (You will see more details on these operators coming up in chapter 4.) The result here is that you appended an exclamation point, which is a string instance, to the end of the variable `str`.

Did you see anything in the results sidebar on the right? If you typed everything as it is above, you should see a new line of results representing `str`'s new value. This new value for `str` will have the exclamation point appended to the end of your string. Refer to figure 2.3 to ensure that your editor looks the way ours does.

Figure 2.3 Varying `str`

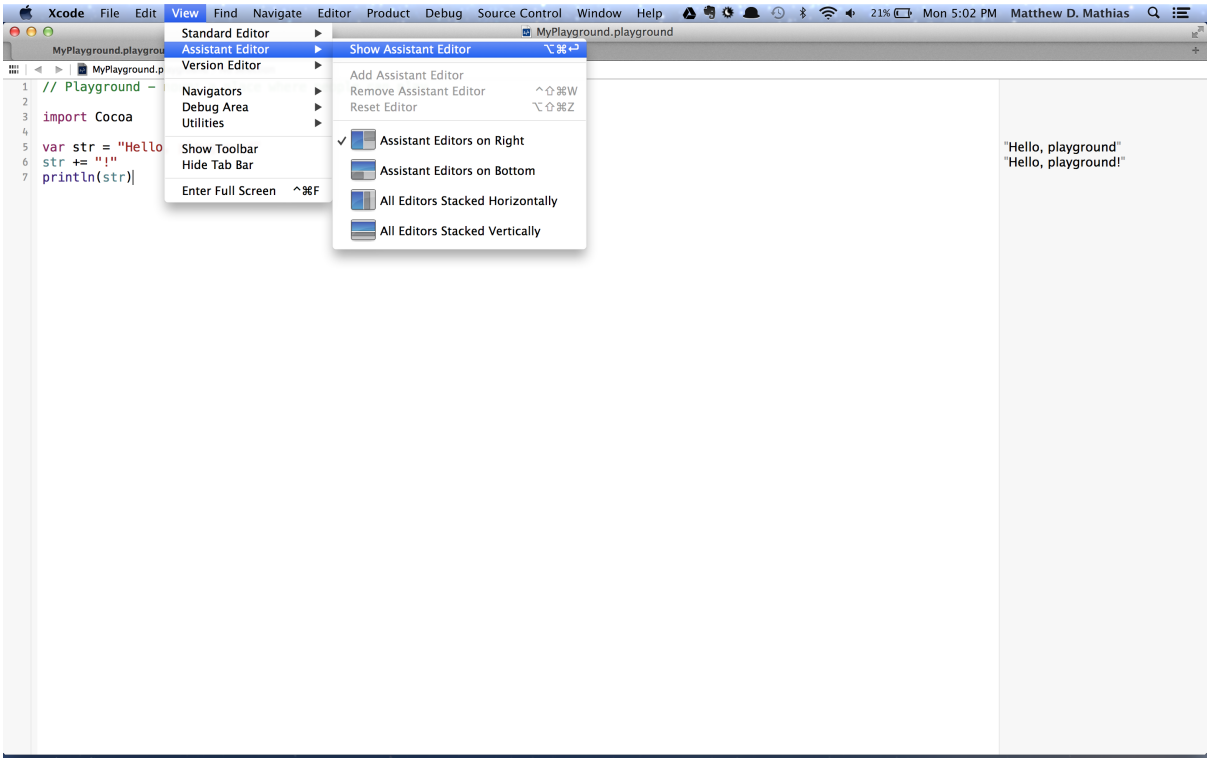


Next, add some code to print the value held by the variable `str` to the console. To do so, you will need to make use of a function called **`println()`**, which is pronounced “print line”. Functions are groupings of related code that send instructions to the computer to complete a specific task. **`println()`** is a function used to print a value to the console followed by a line break.

```
import Cocoa

var str = "Hello, playground"

str += "!"
println(str)
```

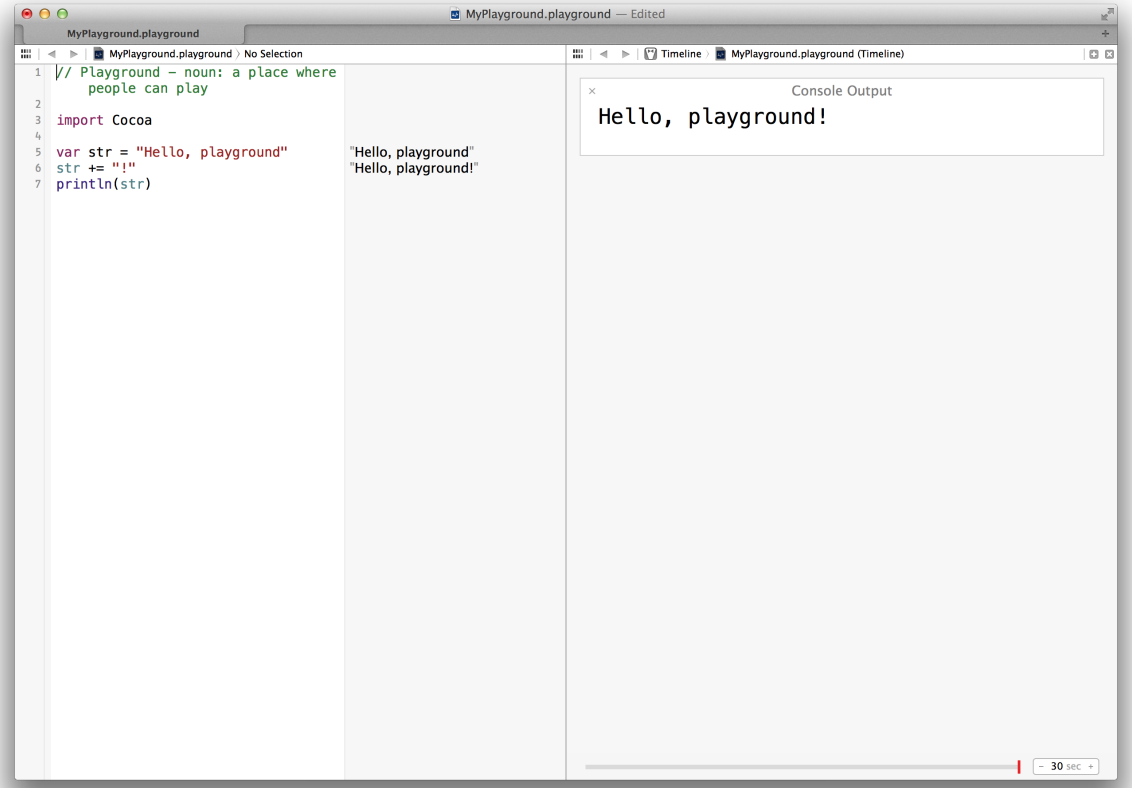
You usually log to the console to check the current value of some variable of interest. You will use the **`println()`** function frequently once your daily development workflow transitions from Swift playgrounds to more fully featured apps in `Xcode`. Currently, the playground's results sidebar is covering up your console. You need to open up the *Assistant Editor* to see it. As seen in figure 2.4, click on "View", "Assistant Editor", and then "Show Assistant Editor". Notice the keyboard shortcut next to this last step? You can optionally type "command-option-enter" all at the same time on your keyboard to expose the *Assistant Editor*.

Figure 2.4 Showing the Assistant Editor



Now that you have your *Assistant Editor* visible, you should see something similar to figure 2.5. In particular, note the new white box in the *Assitant Editor* called "Console Output". The text “Hello, playground!” should be visible on the console.

Figure 2.5 Your First Swift Code



Keep it Up!

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

- Installed `Xcode`
- Created and got acquainted with a playground
- Used a variable and modified it
- Learned about the `String` type
- Used a function to print to the console

That's a good bit of progress, and we didn't even mention all of the bugs, typos, and installation issues you may have encountered! You will be making your own apps in no time, but before then, be sure to stick with it. Learning something new is frustrating work, but it is also extremely rewarding. You have made it this far, and as you continue, you will see that most everything we will cover later on is merely a variation on the themes we've covered thus far.

Challenge

You will notice that many of the chapters in this book are ended with a section for some challenges. These sections are for you to work through on your own so you can deepen your understanding of Swift and get a little extra experience, too. This chapter is no different! Let's get started. But first, you'll probably want to create a new playground to work in. Go ahead and do that.

In addition to several other interesting details, you learned about the `String` type and printing to the console using **`println()`**. 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

3

Types, Constants, and Variables

This chapter will introduce you to constants, variables, and Swift's basic data types. 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 the constants and variables, as well as each of the data types, that shape their use cases.

Types

Variables and constants have a data type. Types describe the nature of the data, and provide information to the compiler on how to handle the data stored by the variable or constant. Based on the type of a constant or variable, the compiler will be able to know how much memory to reserve and will also be able to help out with *type checking*, a feature of Swift that helps to prevent you from assigning the wrong kind of data to a variable, for example.

For example, imagine that you are modeling a small town in your code. There might be a variable for the town's number of stop lights. Suppose that you write the following code to give this variable a value. Go ahead and create a new OS X Playground, and type in the code below.

```
import Cocoa

var numberOfStopLights = "Four"
```

The first thing to notice is that you have assigned an instance of the `String` type to the variable called `numberOfStopLights`. Swift uses *type inference* to determine the data type of your variable. This means that the compiler knows the variable `numberOfStopLights` is of the `String` type because the value on the right side of the equality, what you use to assign a value to what is on the left side, is an instance of `String`. In this case, the compiler knows that `"Four"` is of the `String` type because it is a `String` literal, as denoted by the quotation marks.

If you try to add 2 to this variable, the compiler will give you an error telling you that this operation does not make sense. The reason you get this error is because you are trying to add a number to your variable that is an instance of the `String` type. What does it mean to add a number to a string? Does it double the string and give you "FourFour"? Put "2" on the end and give you "Four2"? Nobody knows - this is what it doesn't make sense to add a number to an instance of `String`.

```
import Cocoa

var numberOfStopLights = "Four"
numberOfStopLights + 2
```

In fact, it doesn't really make sense to have `numberOfStopLights` be of type `String` in the first place. As this variable is explicitly representing the number of stop lights in our theoretical town, instead use a numerical type. Swift provides an `Int` type that is perfect for your variable, which will be a small integer. Change your code to match the following:

```
import Cocoa

~~var numberOfStopLights = "Four"~~
var numberOfStopLights: Int = 4
numberOfStopLights + 2
```

There are a few items to note here. First, notice that you are deleting your original assignment of `"Four"` to the variable `numberOfStopLights`. In this book, we will follow the convention of bolding and striking through old code that you should remove.

Second, notice how your code differs from our first attempt at giving `numberOfStopLights` a value. While before you relied upon Swift's capacity to infer a variable's type based on its value, you are now explicitly declaring that the variable `numberOfStopLights` to be of the `Int` type using Swift's *type annotation* syntax. The colon in the code above represents "*...of type...*". Thus, the code could be read as something like: “Declare a variable called `numberOfStopLights` of type `Int` that starts out with the value of 4.” If you were to later reassign your previous `String` instance of `"Four"`, then the compiler would give you a warning telling you that it cannot convert a string to an integer.

Finally, also note that your error has disappeared. It is perfectly fine to add 2 to your integer variable representing your town's number of stop lights. In fact, because you have declared this instance to be a variable, this operation is perfectly natural. You will return to this issue later on in the chapter.

Swift has a host of other frequently used data types, including numbers (the subject of Chapter 5) and strings, which contain textual data (the subject of Chapter 8). There are other commonly used types, such as *Collection Types*, which you will see shortly.

Constants vs. Variables

The Chapter title includes the terms "Constants" and "Variables". What exactly *are* those? Up to now, you have only seen variables. Variables' values can vary, which means that you can assign a new value to a variable. In this code, however, the values never varied, and things would have worked fine if they where declared as constants. You cannot change the value of a constant.

In fact, making `numberOfStopLights` constant would have been arguably better considering that you did not require its value to vary. Here's a good rule of thumb: use variables for instances that vary, and constants for instances that do not. Once you set a value for a constant, you cannot change its value. As you can imagine, this limitation is not the case for a variable; you can change a variable's value as much as you like.

Swift has a different syntax for declaring both constants and variables. Use the `let` keyword to declare that an instance is a constant and the `var` keyword to declare a variable. Add some constants and variables to the current playground to expand this small town.

```
import Cocoa

~~var numberOfStopLights: Int = 4~~
~~numberOfStopLights + 2~~
let numberOfStopLights: Int = 4
```

Notice that your variable `numberOfStopLights` is now declared to be a constant via the `let` keyword. This change makes some sense considering that the town you are modeling is small, and it is not likely to get a new stop light anytime soon. While we are on the topic of the town's size, add an `Int` to represent its population.

```
import Cocoa

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

Your town's population, though not likely to be large, is likely to vary upwards and downwards a bit. Thus, you declared `population` with the `var` keyword to make this instance a variable. Notice that you declared `population` to be an instance of type `Int`. You did so because a town's population is counted in terms of whole persons. It is also important to point out that `population` is not initialized with any value, and is therefore an empty `Int`. You can use the equals sign, otherwise known as the *assignment operator* to give `population` its starting value.

```
import Cocoa

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

String Interpolation

Every town needs a name. It'd be nice to have a short description of the town that the Tourism Council could use. Your town is fairly stable so it won't be changing its name any time soon. Make the town name a constant of type `String`.

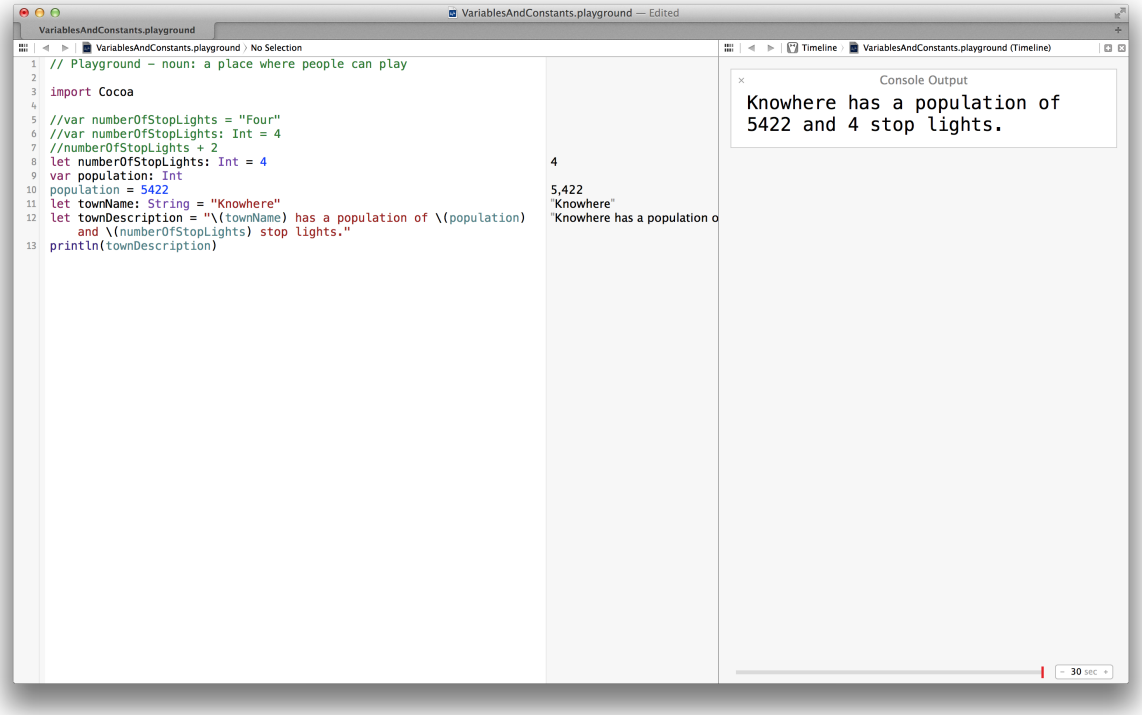
The description is going to be a `String` type constant, though you will be creating it a bit differently than the constants and variables you have already made. The description will include all the data you've seen so far, and you'll be creating it using a Swift feature *string interpolation* that lets you put constant and variable values right into a new string. After creating the description you will print it to the console:

```
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) stop lights."
println(townDescription)
```

String interpolation is a method you can use to create a new `String` value by combining variables and constants. You can then assign this string to a new variable or constant, or just print it to the console. The `\()` syntax represents a placeholder in the `String` literal that accesses an instance's value and places it within the new `String`. For example, `\(townName)` will access the constant `townName`'s value and place it within the new `String` instance.

The result of the new code included above is pictured below. As you can see, the text “Knowhere has a population of 5422 and 4 stop lights.” is printed to the console in the Assistant Editor on the right.

Figure 3.1 Knowhere's Short Description



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 your **`println()`** statement to log this new information to the console.

Part II

The Basics

4
Conditionals

In previous chapters your code led a relatively simple life: you declared some relatively simple 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 it makes decisions based upon the contents of its variables. For example, a game may let the player leap a tall building if they have eaten a particular power-up. You use conditional statements to make these kind of decisions.

if/else

You use `if/else` statements to conditionally execute code based upon a specific logical condition. So what does that mean? You have a relatively simple either-or decision to make, and depending on that decision, one branch of code or another gets run (but not both). Your small town from the previous chapter either has a Post Office or it does not. If your small town has a Post Office, you can buy stamps. If your small town does not have a Post Office, you will need to drive to the next town to buy stamps. There being a Post Office is your specific loglical condition. The different behaviors are "get stamps in town" or "get stamps out of town".

Some situations are more complex than a yes/no decision. You'll see a more flexible mechanism called `switch` in Chapter 6. But for now, let's keep it simple.

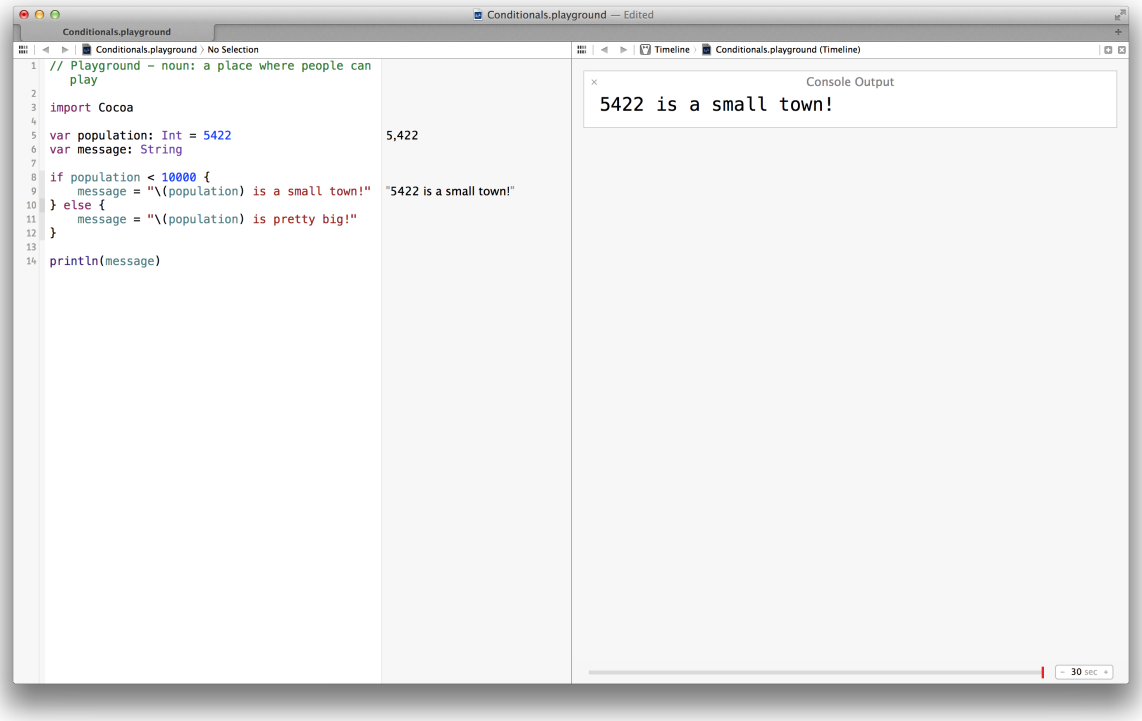
Go ahead and create a new OS X Playground. Name it Conditionals and save it where you like. Once you have a Playground ready, make it look like the code below. The code shows the basic syntax for an `if/else` statement:

```
import Cocoa

var population: Int = 5422
var message: String

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

println(message)
```

You first declared `population` as an instance of the `Int`, and then assigned it a value of 5,422. Next, you declared a variable called `message` that is of the `String` type. This time, however, you leave this variable uninitialized, meaning that `message` does not yet have a value associated with it. Instead, you elect to give `message` a value after you have evaluated your conditional `if/else` statement. Notice that you are using *string interpolation* to put the population into the string.

Figure 4.1 shows what your Playground should look like. Note that the console and the results sidebar show that `message` has been set to be equal to the string literal when the conditional evaluates to be true. How did this occur?

Figure 4.1 Conditionally Describing Town's Population



The condition in the `if/else` statement tests whether or not your town's population is less than 10000 via the < comparison operator. If it condition evaluates to true, then you should set `message` to be equal to the first string literal ("X is a small town!"). If the condition evaluates to false, the population is 10,000 or greater, so the `message` to be equal to the second string literal (X is pretty big!"). In this case, the town's population is indeed less than 10,000 and `message` was set to: “5422 is a small town!”. Table 4.1 below lists Swift's comparison operators.

Table 4.1 Comparison Operators

Operator	Description
<	Evaluates whether the number on the left is smaller than the number on the right.
<=	Evaluates whether the number on the left is smaller than or equal to the number on the right.
>	Evaluates whether the number on the left is greater than the number on the right.
>=	Evaluates whether the number on the left is greater than or equal to the number on the right.
==	Evaluates whether the number on the left is equal to the number on the right.
!=	Evaluates whether the number on the left is not equal to the number 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.

An aside: eventually you will need to know the difference between == and === in the table above. Right now it probably doesn't make much sense, but after you see structs and classes in Chapter 15, it should be clearer. Bookmark this description for later:

== returns a `Boolean` indicating whether or not two instances are equal. In other words, == checks for equality or *equivalence*. === returns a `Boolean` indicating whether or not the two objects point to the same location in memory, whether the two objects are *identical*. If they both live in the same place in memory, then === returns `true`. Similarly, `!==` checks whether or not two references *do not* point to the same location in memory. If they do not, then this operator returns `true`.

These last two operators check for *identity*, and are therefore referred to as *identity operators*. Equality and identity are distinct concepts. An object may be equal in value to another object, but it may not point to the exact same location in memory. Thus, developers have separate definitions for *equality* and *identity*. Instances may be equal (have the same value) but not identical (occupy the same location in memory). It is important to keep these two definitions separate in your head.

Sometimes you only care about one aspect of the condition that is under evaluation. That is, you only want to execute code if a certain condition is met. For example, consider the code below.

```
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!"
}

println(message)

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

The code above has a new kind of operation called a *logical operator*. The `!` tests whether or not `hasPostOffice` is false. You can think of `!` as inverting a `Boolean` value. true becomes false, and false becomes true. If `hasPostOffice` is false, you don't know where to buy stamps and have to ask. This operator is known as "logical not". Since the town does have a post office (that `hasPostOffice` got initialized to true), you know where to buy stamps and do not have to ask any questions. Thus, our condition (true, but flipped over to false) ultimately evaluates to false and the **`println()`** function never gets called. Refer to table Table 4.2 below for a listing and definition of Swift's logical operators.

Table 4.2 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 `if/else` statements, but is much more concise in terms of its syntax. The syntax looks likes this: `a ? b : c`. In English, the ternary operator reads something like, "If `a` is true, then do `b`. Otherwise, do `c`." The town populate check that used `if/else` could be rewritten 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 typing than a simple `a ? b : c`. They're great for concise statements, but if your ternary operator statement starts wrapping to the next line, then you'll want to stick with `if/else`.

Use undo to remove the ternary operator and restore your `if/else` statement:

```
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!"
```

Nested ifs

You can nest your `if` statements to conditionally execute code after an earlier condition evaluated to `true`. You do this by writing a new `if/else` statement inside of of the curly braces from another `if/else` statement. In the code below, nest an `if/else` statement within the `else` block:

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

println(message)

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

Your `nested if` clause makes use of the >= comparator and the && logical operator to check if `population` is within the range of 10,000 to 50,000. You are using this range to arbitrarily define a medium sized town. Since your town's `population` doesn't lie within the range, our `message` is set to “5422 is a small town!” as it was 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. The danger of nesting them too deeply is it makes the code harder to read, forcing whoever is reading your code to keep a lot of program state in their brains. One or two levels are fine, but beyond that your code becomes less readable and maintainable. There are ways of expressing code that don't require nested statements. Let's refactor the code that we've just written to make it a little easier to follow.

Actually, before refactoring the code, you might want to know what that word means. *Refactoring* is a term that means changing code so that it does the same work it did before, but does it in a different way. Maybe it's more efficient, or maybe it just looks prettier or is easier to understand. When you refactor something, you're not changing how it behaves, just how it looks.

else if

The `else if` conditional lets you chain multiple conditional statements together. To make your code a little easier to read, extract the nested `if/else` statement to be a standalone clause that evaluates whether or not your town is of medium size. `else if` allows you to check against multiple cases, and conditionally executes code depending upon which clause evaluates to true. You can have as many `if/else/else if` clauses as you want, but only one will be run.

```
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!"
    }
    message = "\(population) is pretty big!"
}

println(message)

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

The code above adds one `else if` clause, but you could have chained many more more. This block of code is an improvement over the nested `if/else` described above. If you find yourself with a whole bunch of `if/else` statements, you may want to use another mechanism, such as `switch` described in Chapter 6. Stay tuned.

Challenge

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

5
Numbers

At their lowest level, computers fundamentally work with numbers. They are also a staple of software development; you can use numbers to keep track of temperature, determine how many letters are in a sentence, and even how many zombies are infesting a town. There are two basic flavors of numbers: integers and floating-point numbers.

Integers

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 that you know is that an integer type on a computer takes up a fixed amount of memory. Therefore, they can't represent all possible whole numbers - they have a minimum and maximum value they can represent. To see this in action, create a new Playground, naming it `Numbers.playground`, and enter the following code:

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

Open up the Timeline view for your playground. You should see the following output:

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

What is the reasoning behind those values? Recall that integer types use a fixed amount of memory. The memory that an integer uses is ususally described in terms of the number of bits it requires; a bit is a single 0 or 1. On OS X, `Int` is a 64-bit integer, which means it has 2^{64} possible values. Additionally, `Int` is a *signed* integer, which means it can represent both positive and negative whole numbers. The minimum `Int` value is -2^{63} = -9,223,372,036,854,775,808, and the maximum `Int` value is $2^{63} - 1$ = 9,223,372,036,854,775,807 (the "- 1" leaves one representation for 0).

On iOS, `Int` is slightly more complicated. Apple introduced 64-bit devices starting with the iPhone 5S, iPad Air, and iPad mini with Retina display. If you write an iOS app for those devices, which is called targeting a 64-bit architecture, `Int` is a 64-bit integer just like on OS X. On the other hand, if you target a 32-bit architecture like the iPhone 5 or iPad 2, `Int` is a 32-bit integer instead. 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. Use `Int32` to see the minimum and maximum value for a 32-bit integer:

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

`Int32` is Swift's 32-bit signed integer type. Also available are `Int8`, `Int16`, and `Int64`, for 8-bit, 16-bit, and 64-bit signed integer types. The sized integer types exist when you truly need to know the size of the underlying integer, such as for some algorithms (common in cryptography) or if you need 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 can represent both positive and negative numbers. Swift also has unsigned integer types to represent whole numbers that are greater than or equal to 0. Try a couple of these out:

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

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

Every signed integer type (`Int8`, `Int16`, etc.) has a corresponding unsigned integer type; each of these gets its name by prepending a `U` in front of its integer type (`UInt8`, `UInt16`, etc.). The minimum value for all unsigned type 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 = 18,446,744,073,709,551,615. You can see the 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 have 2^{64} possible values, but the signed version has to devote half of them to negative numbers.

As you probably noticed from the code you typed into your Playground, `UInt` is the corresponding unsigned integer type to `Int`, and just like `Int`, `UInt` is a 64-bit integer on OS X and may be 32-bit or 64-bit depending on the target for iOS. Some quantities seem like they would naturally be represented by an unsigned integer. For example, it doesn't 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 3, where you learned that you can declare a type explicitly or implicitly:

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

let myInt: Int = 10 // declare type explicitly
let myOtherInt = 3  // also of type Int, inferred by the compiler
```

You can create instances of the other integer types using explicit type declarations:

```
let myInt: Int = 10 // declare type explicitly
let myOtherInt = 3  // also of type Int

let myUInt: UInt = 40
let myInt32: Int32 = -1000
```

What happens if you try to create an instance with an invalid value? For example, you might try to create a `UInt` with a negative value, or an `Int8` with a value greater than 127. Try it and find out:

```
let myUInt: UInt = 40
let myInt32: Int32 = -1000

// Trouble ahead!
let myBadUInt: UInt = -1
let myBadInt8: Int8 = 200
```

You should see a red exclamation mark in Xcode. Click on the exclamation mark to see the error. The compiler reports an "Integer literal overflow when stored into...". An "integer literal" is the whole number you typed in; `-1` and `200` are integer literals. "Overflow when stored into..." means when the compiler tries to store your number into the type you specified, it doesn't fit in the type's allowed range of values. An `Int8` can hold values from -128 to 127; 200 is outside of that range, trying to store 200 into an `Int8` overflows. Remove the problematic code and we will move on.

```
~~// Trouble ahead!~~
~~let myBadUInt: UInt = -1~~
~~let myBadInt8: Int8 = 200~~
```

Operations on Integers

Swift allows you to perform basic mathematical operations on integers. The arithmetic operations + (add), – (subtract), and * (multiply) exist and allow you to perform simple math between numbers. Try printing the result of some arithmetic:

```
let myUInt: UInt = 40
let myInt32: Int32 = -1000

println(10 + 20)
println(30 - 5)
println(5 * 6)
```

The operators also have a concept of *precedence* and *associativity*, which define the order of operations when there are multiple operators in a single expression. For example:

```
println(10 + 20)
println(30 - 5)
println(5 * 6)

println(10 + 2 * 5) // 20, because 2 * 5 is evaluated before 10 + 2
println(30 - 5 - 5) // 20, because 30 - 5 is evaluated before 5 - 5
```

You could memorize the rules governing precedence and associativity, but we would instead recommend using parentheses to make your intentions explicit:

```
println(10 + 2 * 5) // 20, because 2 * 5 is evaluated before 10 + 2
println(30 - 5 - 5) // 20, because 30 - 5 is evaluated before 5 - 5
println(10 + (2 * 5))
println((30 - 5) - 5)
```

Integer Division

What's the value of the expression `11 / 3`? You might (reasonably) expect 3.66666666667, but try it out:

```
println(10 + (2 * 5))
println((30 - 5) - 5)

println(11 / 3) // prints 3
```

The result of any operation between two integers is always another integer of the same type. 3.66666666667 isn't a whole number and can't be represented as an integer, so Swift truncates the fractional part, leaving just 3. If the result is negative, such as `-11 / 3`, the fractional part is still trucated, giving a result of -3. Integer division therefore always rounds towards 0.

It is also occasionally useful to get the remainder of a division operation. The remainder operator, %, returns exactly that. Be warned that the using the remainder operator on a negative integer may not return what you expect:

```
println(11 / 3) // prints 3
println(11 % 3) // prints 2
println(-11 % 3) // prints -2 (surprise!)
```

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 *increment* an integer (add 1 to it) or *decrement* an integer (subtract 1 from it). You can use the ++ operator and the -- operator to perform these operations:

```
println(-11 % 3) // prints -2 (surprise!)

var x = 10
x++
println("x has been incremented to \(x)")
x--
println("x has been decremented to \(x)")
```

What if you want to increase `x` by a number other than 1? You can use the `+=` operator, which combines addition and assignment:

```
x--
println("\(x) has been decremented")

x += 10 // equivalent: x = x + 10
println("x has had 10 added to it and is now \(x)")
```

There are also shorthand operation-and-assignment combination operators for the other basic math operations: `-=`, `*=`, `/=`, and `%=`.

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.)

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

If you thought the value `z` would be 130, you are not alone. Instead, you probably do not see a value at all. Open up the Timeline view of your playground. Below all the output of the previous parts of this chapter, you should see several lines of seemingly meaningless numbers and names, starting with something like "Playground execution failed: Execution was interrupted." Let's break down what's happening:

1. The compiler infers the type of `z` to be `Int8`, because `y` is an `Int8`, so `y + 10` must be too.
2. When your playground runs, Swift adds 10 to `y`, resulting in 130.
3. 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 14. For now, know that a trap results in your program stopping immediately and noisily, which indicates a serious problem you need to examine.

Swift provides *overflow operators* that have different behavior when the value is too big (or too small). Instead of trapping the program, they "wrap around". The overflow addition operator is `&+`; try it now:

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

The result of overflow-adding 120 + 10 and storing the result into an `Int8` is -126. Think about incrementing the number one at a time. Once you get to 127, incrementing one more time wraps around to -128. So 120 + 8 = -128, 120 + 9 = -127, and 120 + 10 = -126.

There are corresponding overflow versions of the other arithmetic operators: `&-`, `&*`, `&/`, and `&%`. It should be apparent why overflow versions of subtraction and multiplication exist, but what does it mean for division or remainder to "overflow"? If you try to divide by 0, or get the remainder of dividing by 0, that causes a trap. If you use the overflow divide operator to divide by 0 (or use the overflow remainder operator to get the remainder of dividing by 0), instead of trapping, you get 0 back.

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 languages; 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 far better to trap (which may result in a program "crashing", at least from a user's perspective) 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?

```
let a: Int = 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 `Int`. Only one of these will succeed (Why? Reread the previous section!), so do that now:

```
let a: Int = 200
let b: Int8 = 50
~~let c = a + b // Uh-oh!~~
let c = a + Int(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 perform operations between an `Int` and any other integer type 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 very freely.

Requiring you, the programmer, to decide how to convert variables in order to do math between different types is another factor that distinguishes Swift from other languages. Again, this requirement is in favor of safety and correctness. C, 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 when it does its 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 doing the math.

Floating-Point Numbers

To represent a number that has a decimal point, like 3.2, you use a floating-point number. In computers, floating-point numbers are stored as a *mantissa* and an *exponent*, similar to how you write a number in scientific notation: 123.45 might also be written as 1.2345×10^2 or 12.345×10^1. Additionally, floating-point numbers are often imprecise: there are many numbers that cannot be stored with perfect accuracy in a floating-point number, so the computer will often store a very close approximation to the number you expect.

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` don't determine a basic minimum and maximum value range like 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 literals in Swift is `Double`. You can use the same syntax as you did for integers to declare explicit 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 as well, including the remainder operator:

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

println(10.0 + 11.4)
println(11.0 / 3.0)
println(12.4 % 5.0)
```

The fact that floating-point numbers are inherently imprecise is an important difference from integer numbers that you should keep in mind. Recall the == operator from Chapter 4, which determines if two values are equal to each other. You can compare floating-point numbers:

```
println(10.0 + 11.4)
println(11.0 / 3.0)
println(12.4 % 5.0)

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

So far so good. The value of `d1` is 1.1, so you would expect that adding 0.1 to it would result in 1.2. However, try it out:

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

println("d1 + 0.1 is \(d1 + 0.1)")
if d1 + 0.1 == 1.2 {
    println("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 `println()`, but the `println()` inside the if statement doesn't run. Why not? It turns out that 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, and 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 aren't technically equal, which explains why the `println()` inside the if statement doesn't execute.

All the gory details behind floating-point arithmetic are well 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.

Conclusion

You have completed a tour of the basic number types provided in Swift. You now know about the many signed and unsigned integer types, and why you should try to stick with `Int` as much as you can. You learned about the basic arithmetic operators, including their shorthand forms and their overflow versions. You also have some basic knowledge about floating-point types, and know that they can be used to represent numbers that have decimal points, but you also know that computers cannot represent floating-point numbers exactly. Finally, you learned how to convert between numbers of different types, and that Swift requires you to do so.

6
Switch

Our previous chapter covered one sort of conditional statement: `if/else`. Along the way, we discussed that `if/else` can be somewhat inadequate in dealing with scenarios that have more than a few conditions. We now shift our focus to the `switch` statement. Unlike `if/else`, `switch` is ideally used for handling multiple cases. As you will see below, Swift's `switch` statement is an incredibly flexible and powerful feature of the language.

What is a Switch?

`if/else` statements execute code depending whether or not 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 a match occurs, the `switch` executes the code associated with the body of that case. Here is the general syntax of a `switch` statement.

```
switch aValue {

    case someValueToCompare:
        // Do something to respond

    case anotherValueToCompare:
        // Do something to respond

    default:
        // Do something to handle no matches

}
```

In the above, we only compare against two cases, but we could have included an arbitrary number of other cases to compare against. If `aValue` matches any of the comparison cases below, then the body of that case will be executed. Implicit in this point is the requirement that the type in each of the cases must match the type being compared against. More concretely, `aValue`'s type must matched by `someValueToCompare` and `anotherValueToCompare`'s type. Last, if `aValue` matches `someValueToCompare`, then the code below the case would be executed. In this case, there is only a comment: `// do something to respond`.

It is important to mention at this point that the `switch` statement above will cause a compile-time error. Why? If you're curious, go ahead and type the above into a `Playground` If `aValue` has a value, and so do all of the cases, you should see an error for each of the cases telling you the following: "'case' label in a 'switch' should at least one executable statement". This error means that every case should have at least one executable line of code associated with it. Since comments are intrinsically not executable, our `switch` statement does not meet this requirement. Thus, the error underlines the point that each case in `switch` statements represents a separate branch of execution.

Notice the use of the `default:` case. The `default` case is not mandatory, but it does help to fulfill an important requirement Swift has for its `switch` statements: they must be exhaustive for all cases. This requirement means that every value of the type checked against must have a case associated with it. Here is where the `default` case comes in: the `default` case is executed if no match occurs in the cases listed above it. We typically use a `default` case when it does not make sense to have a specific case for each value in the type to be matched. Thus, if `aValue` did not match `someValueToCompare` or `anotherValueToCompare`, then the code beneath the `default` case would be executed.

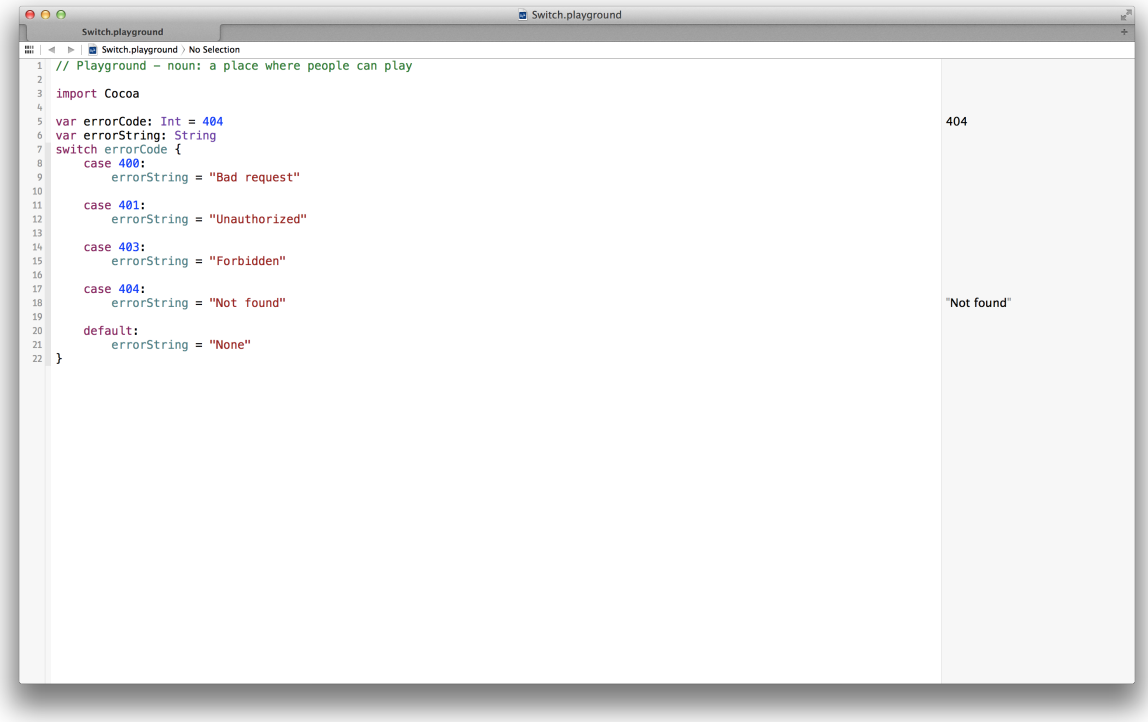
Switch It Up

Create a new `Playground` called `Switch` and save it to a location of your choosing. Type in the following to have your code match ours.

```
import Cocoa

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 uses a HTTP status code to match a `String` instance describing the error to the code if there is indeed an error. Notice that the results sidebar shows you which case matched the error. `errorString` is therefore assigned to be equal to `"Not found."` We can thereafter use `errorString` as needed.

Figure 6.1 Matching an Error String to an Error Code



Let's say that we want to use `switch` statement above to build up a meaningful error description. Change your code to match ours.

```
import Cocoa

var statusCode: Int = 404
~~var errorString: String~~
var errorString: String = "The request failed with the error:"
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."
}
```

This new `switch` statement differs somewhat from our previous example. We now initialize `errorString` with a string representing the first segment of our error's description. The colon at the end of `errorString` noticeably signals that this instance is not yet finished and ready to share. The rest of the `switch` statement has a few important differences. You now only have one case for all of the error status codes. You also used the compound assignment operator `+=` within the case's body to add the specific `statusCode` information to the `errorString`. If the `statusCode` matches any of the values in the case, then the `errorString` will have the `" There was something wrong with the request."` text added to it.

You also introduced a *control transfer statement* called `fallthrough`. The `fallthrough` keyword ensures that the `switch` statement "falls through" the bottom of each case to the next one. If there were multiple cases, the `switch` will cycle through each case, building up the `errorString`. The `fallthrough` control transfer statement causes the actual cases to be ignored, and ensures that the code within each case will be executed. In the above example, the `fallthrough` statement means that if the case matches, then the `switch` statement will not conclude, but will instead proceed to the `default` case. If we had not used the `fallthrough` keyword, then the `switch` statement would have ended execution after the first match. The result of the above `switch` statement is that `errorString` is set to: "The request failed with the error: There was something wrong with the request. Please review the request and try again."

Advanced Switching

Imagine that we don't really care to build a very specific `errorString`. One feature in Swift's `switch` statement that can help us is its ability to match ranges of cases to the value we would like to switch over. Let's rewrite our `switch` statement above to take advantage of this feature.

```
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 each category of HTTP status code. That is, `300...307` constitutes a range that includes 300, 307, and every integer that is in-between. You'll notice that our first range is separated by a comma, as there are only two codes to check against. The next case covers only 204 because the 200s generally represent "successful" status codes, but 204 is the code for "No content". The result of this `switch` statement is that `errorString` will be set to equal: "The request failed with the error: Client error 4xx." Note, if we simply do not recognize the code, then our `errorString` will reflect this information via our `default` case.

Imagine, however, that our application will be used by web development professionals well-acquainted with HTTP status codes. So, while we won't need to be extremely specific about the text of the status code, the users of our application would like to see the actual numerical status codes in our `errorString`. Also suppose that we need to guard against unknown error codes such that we don't erroneously create our `errorString`. We can build upon our previous `switch` statement and include this information using a feature of Swift's `switch` statements called *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."
}
```

We use *string interpolation* in the above example to pass our `statusCode` into the `errorString` in each case. The only case that is different is the last one. In this case, we created a temporary constant, called `unknownCode` that our `switch` will compare against when the `statusCode` does not match any of the values provided in the ranges above. For example, if the `statusCode` was for some reason set to be equal to 200, representing success, then our `switch` would set `errorString` to be equal to: "200 is not a known error code." If we needed to do work on `unknownCode`, for whatever reason, we could have declared it with `var` instead of `let`. Doing so would mean that we could have modified `unknownCode`'s value within the final case's body.

The above is fine, but actually may lead to trouble. After all, a status code of 200 is not *really* an error, and so it would be nice if our `switch` statement didn't catch these cases. Below, we use a `where` clause to make sure `unknownCode` is not a 2xx, success. `where` allows us to check for additional conditions within our `switch` statement. This feature has the behavior of creating a sort of dynamic filter within the `switch`.

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

    default:
        errorString = "Unexpected error encountered."
}
```

If you recall the previous example above, we simply declared a constant in that final case, which meant that this case matched on everything that eventually made its way down to the bottom of the `switch` statement. This functionally made our previous `switch` statement exhaustive. Since our case for `unknownCode` now specifies a specific range of status codes, it is no longer exhaustive. Thus, we added a `default` case that sets `errorString` to indicate that the error is unknown.

Since we are not utilizing Swift's `fallthrough` feature, the `switch` statement will finish execution as soon as it finds a matching case and executes its body. Accordingly, if `statusCode` is equal to 204, then it will match at that case and the `errorString` will be set accordingly. Thus, this value will never be matched in this case despite the fact that the range specified in the `where` clause includes it.

Tuples and Pattern Matching

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

A tuple is a grouping of values that are deemed by the developer to be logically related. These different values are grouped together as a compound value. The result of this grouping is an ordered list of elements.

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

```
var statusCode: Int = 420
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 100, 101:
        errorString += " Informational, \(statusCode)."

    case 204:
        errorString += " Successful but no content, \(statusCode)."

    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 = "Unknown error encountered."
}
println(errorString)

let error = (statusCode, errorString)
```

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

The elements of a tuple can be accessed by their index. Type in the following to access each element stored inside of the tuple.

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

You should see `420` and `"Unknown error encountered."` displayed in the results sidebar for `error.0` and `error.1` respectively.

Swift's tuples can also have named elements. Naming a tuple's elements makes for more readable code. For example, it is not very easy to keep track of what values are represented by `error.0` and `error.1`.

Give your tuple's elements an informative name.

```
...
~~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 a related name: `code` for `statusCode` and `error` for `errorString`. You should see that your results sidebar has the same information displayed.

Pattern Matching

Tuples are also very helpful in matching patterns. Imagine, for example, that you would like to write a `switch` that matches against a specific range of status codes in the tuple `error`, but you do not really care much about the error string. Switching over a tuple makes that quite easy to do in Swift.

Add the following code to switch on your new tuple to match against a specific range of status codes.

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

switch error {
    case (400...505, _):
        println("PANIC!")

    default:
        println("Not worth panicking.")
}
```

The new switch statement matches against only one case. Notice that the error string is not the focus of the pattern you are trying to match. The underscore (_) matches any error string, which allows you to ignore it. Thus, the focus here is on the specific status codes that are of interest.

Since the status code in the tuple is `420`, you should see `"PANIC!"` in the results sidebar.

Conclusion

We've covered a lot of what `switch` statements can offer. Let's briefly review what we've covered.

- `switch` statements' basic syntax and use
- range matching
- value binding
- control transfer mechanisms via `fallthrough`
- the `where` clause

Each of these items are powerful features, and allow `switch` statements to be an extremely effective means by which we control our applications' logic.

Challenge

Review the `switch` statement below. What will be logged to the console?

```
var statusCode: Int = 301
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 100, 101:
        errorString += " Informational, \(statusCode)."

    case 204:
        errorString += " Successful but no content, \(statusCode)."

    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 += " Unknown error."
}
println(errorString)
```

7
Loops

Loops help us to gain some traction over tasks that are repetitive in nature. They execute a set of code repeatedly depending upon either a given number of iterations or an established condition. You will see that loops can save us from writing tedious and repetitive code. So take note! You'll be using them a lot in your development.

In this chapter, you will use two sorts of loops: 1) the `for` loop, and 2) the `while` loop. The `for` loop is often used when you would like to iterate over the specific elements of an instance or collection of instances. `for` loops are often used when the number of iterations to perform is either known or easy to derive. In distinction, the `while` loop is well-suited for tasks that you would like to execute repeatedly so long as a certain condition is met. Let's get started with our first loop: the `for` loop.

for-in Loops

Create a new `Playground` called `Loops` and save it where you like. Add the following code:

```
import Cocoa

var int: Int = 0

for i in 1...5 {
    ++int
    println(int)
}
```

First, you declared a variable called `int` that is an instance of `Int` and is intitialized to be equal to 0. Second, you wrote what is called in Swift a `for-in` loop. The `for-in` loop performs a set of code for each item in a specific range, sequence, or collection. In the code above, you made use of `...` to create an inclusive range that began at 1 and continued through 5. Additionally, you declared a constant called `i` that represents the current value of the iterator as the loop traverses the range from 1 through 5. For each integer in this range, you incremented `int` by 1. You then logged this value to the console. If you had used `..<` instead, then the range would have still began at 1, but would have ended earlier at 4.

In the loop above, you declared a constant called `i` that only exists inside the body of loop. In the first iteration of the loop, its value is the first value in the range of the loop: 1. In the second iteration, the value of `i` is 2, and so on. The code inside the braces (`{}`) is executed at each iteration of the loop. This process is continued to until `i` reaches the end of the range. Hence, you can read the `for-in` loop above as: "for i in the range of 1 to 5".

To see the results of your loop, reveal the assistant editor by tapping `command-option-return` on your keyboard. As before, you should now see the console and the value of `int` that was logged at each iteration of the loop over its range. Let's add some more information by clicking on the *Value History* button on the right edge of the results sidebar on the line with the code: `++int`. Clicking here will reveal the *timeline* for that instance's value history. Incidentally, you can also tap the value history button to reveal both the assistant editory and the timeline at the same time. See figure 6.1 below for details.

Figure 7.1 The Value History Button

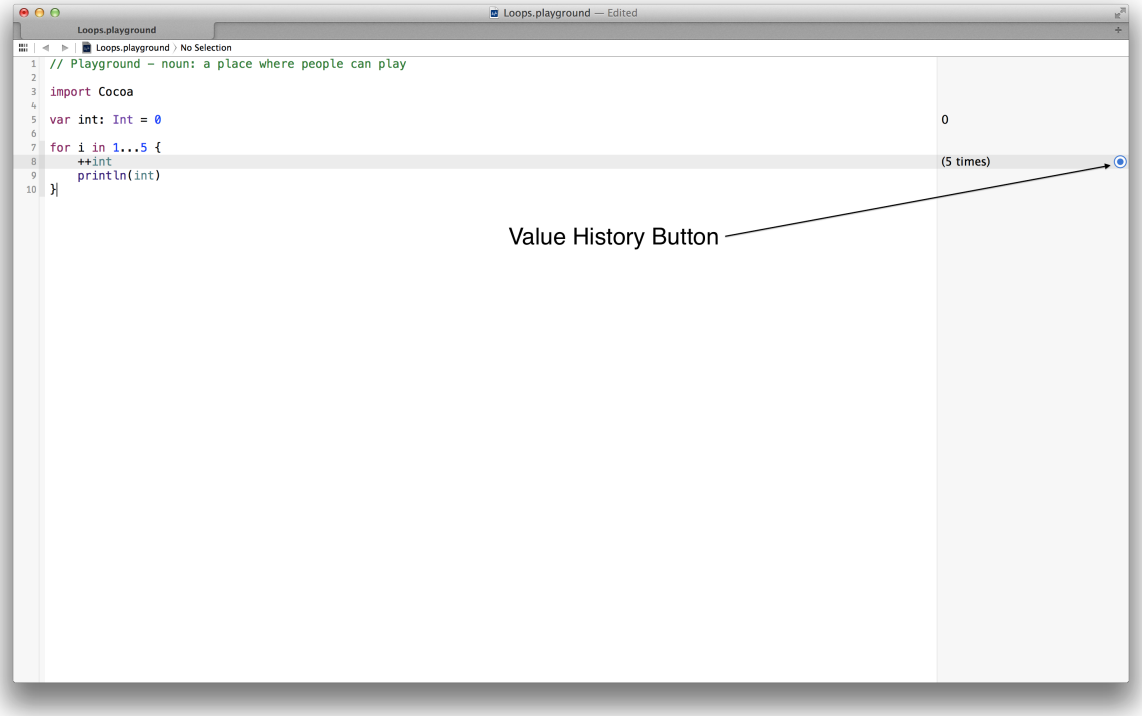
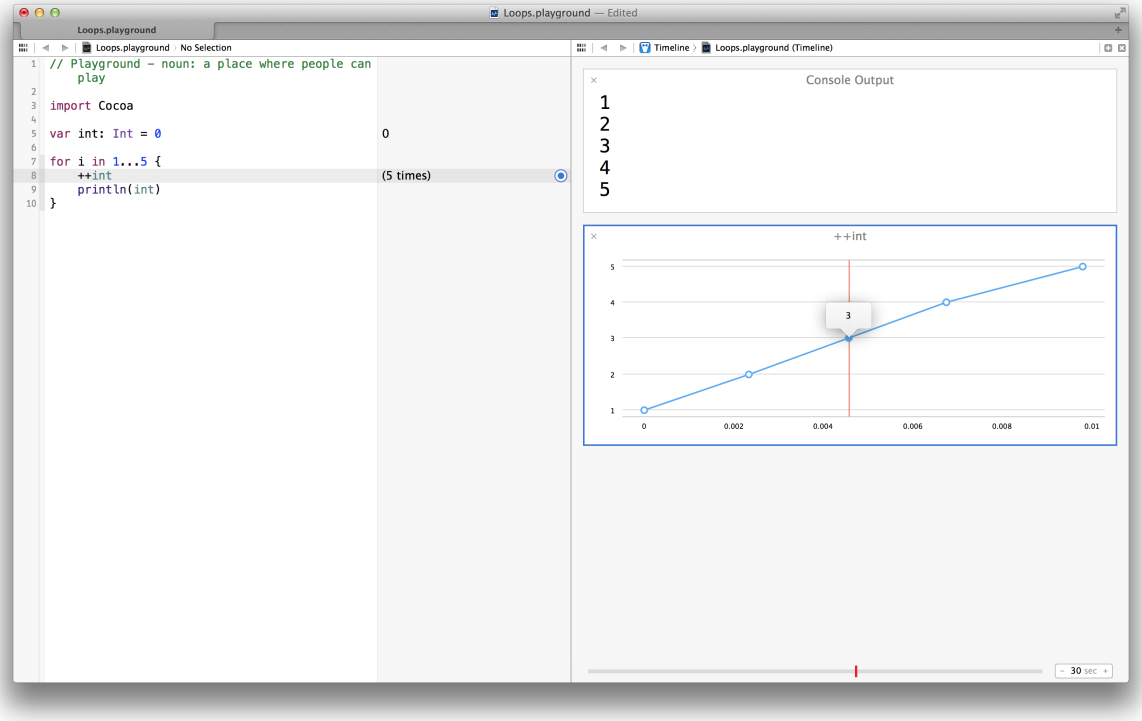


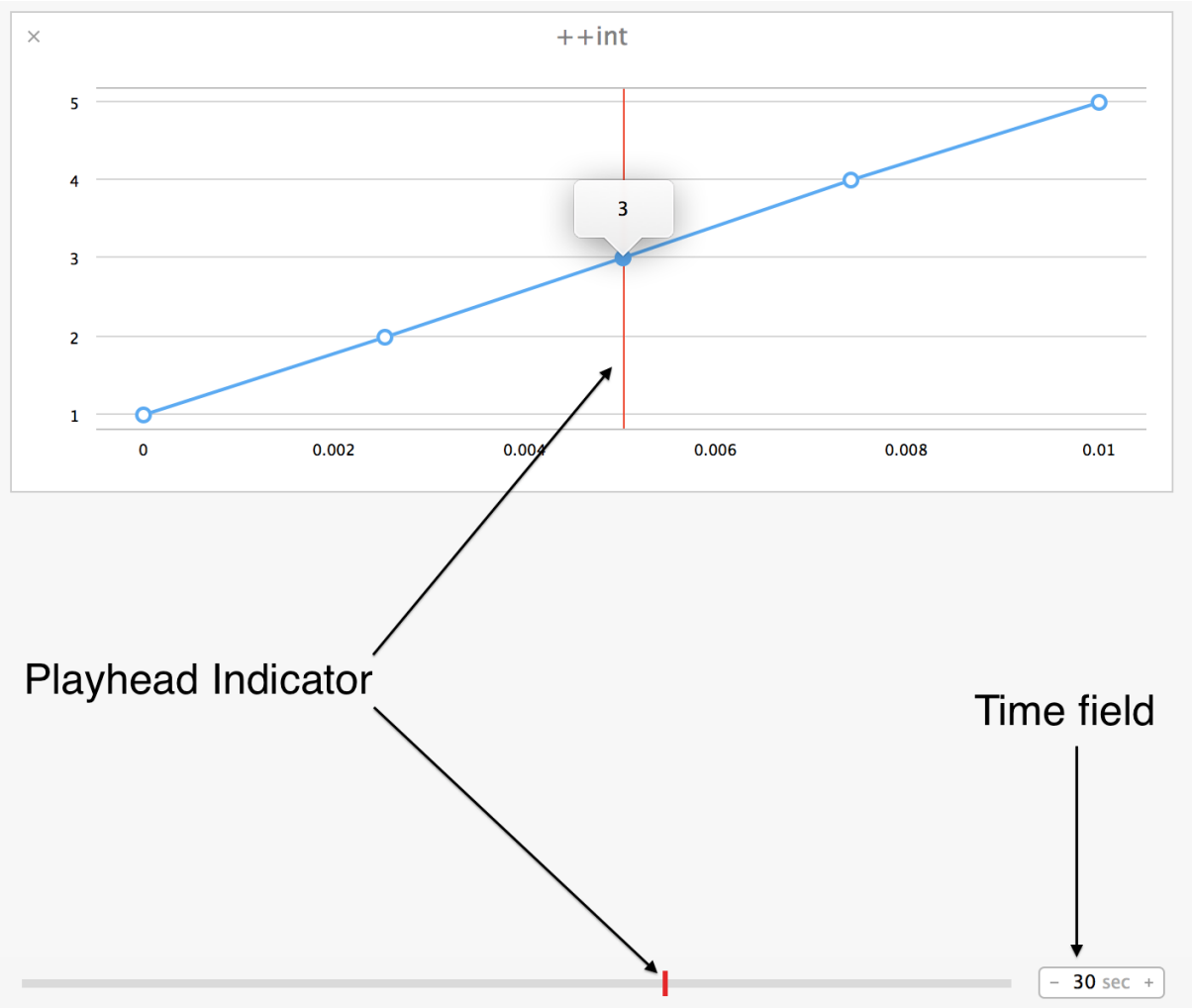
Figure 6.2 shows the result of clicking the value history button. Notice that you see both console and the value history timeline for `int` in the assistant editor. Each point along the line in the timeline shows you the value of `int` at each iteration in the range of the loop above. The x-axis describes execution time, and and the y-axis refers to the value of `int` over the lifecourse of the loop. Click on the third point on the line and you should see the `Playground` highlight and display the value at that particular point. As you can imagine, this feature is extremely nice to have for more complicated or lengthy loops.

Figure 7.2 Displaying the Value History on the Timeline



Notice the slider at the bottom of the assistant editor? You should see a vertical red bar whose position approximately matches your current selection on the line representing `int`'s value history. Slide this red bar horizontally to change the location of the timeline's playhead. You can also manipulate the amount of time that is reported in the timeline. Manipulate the number of seconds displayed in the time field displayed in the lower right to increase or decrease the span of time depicted.

Figure 7.3 The Playhead and the Time Field



Since you declared `i` to be the iterator in the `for-in` loop above, you can access `i` inside of each iteration of the loop. For example, you could have written the above loop like so:

```
for i in 1...5 {
    ++int
    println(int)
    println("int equals \(int) at iteration \(i)")
}
```

The code sample above shows how to access the iterator `i` within each iteration of the loop. This implementation of the `for-in` loop would print the following to the console for the third iteration of the loop: "int equals 3 at iteration 3".

If, for whatever reason, you did not care to have make use of an explicitly declared iterator in the range specified above, then you can completely ignore it by using an _ instead of a named constant. For example, change your loop to match the below. Notice, that you have largely returned to the first implementation for the `for-in` loop at the beginning of the chapter.

```
for i in 1...5 {
for _ in 1...5 {
    ++int
    println("int equals \(int) at iteration \(i)")
    println(int)
}
```

This implementation of the `for-in` loop is nice because it is concerned with ensuring that a specific operation occurs a set number of times. It is not especially concerned with checking and reporting the value of the iterator in each pass of the loop over its range.

A Quick Note on Type Inference

Take a look at the following code.

```
for i in 1...5 {
    ++int
    println("int equals \(int) at iteration \(i)")
}
```

Notice that you didn't have to declare `i` to of the `Int` type? You could if you wanted to (e.g., `for i: Int in 1...5` -- the `let` portion of the declaration is assumed by the syntax for you), but it isn't necessary. The type of `i` is inferred from its context. In this example, `i` is inferred to be of type `Int` because the range specified above contains integers.

As you can see, type inference is quite handy. It makes for less physical typing on the keyboard, and the less typing you have to do, the less typos you will make. There are few cases wherein you will need to specifically declare the type, and we'll be sure to highlight them when they arise. Overall, we advocate that you take advantage of type inference whenever possible, and you will see a lot of it in this book.

for Loops

Swift also supports the classical `for` loop:

```
for initialization; condition; increment {
    // Code to execute at each iteration
}
```

You use semicolons to separate the three parts of the `for` loop. Each parts performs a specific function in the three steps of this loop's execution.

1. When the loop is entered, the initialization expression is evaluated to set up the iterator for the loop.

2. The condition expression is evaluated; if it is false, the loop is ended and execution of the code is transferred to after the loop. If it is true, then the code inside the loop's braces (`{}`) is executed.

3. After the code between the braces is executed, the increment expression is executed. Depending upon the code here, the incrementer can be increased or decreased. Once the incrementer is set, we return to the second step to determine if the loop should continue iterating.

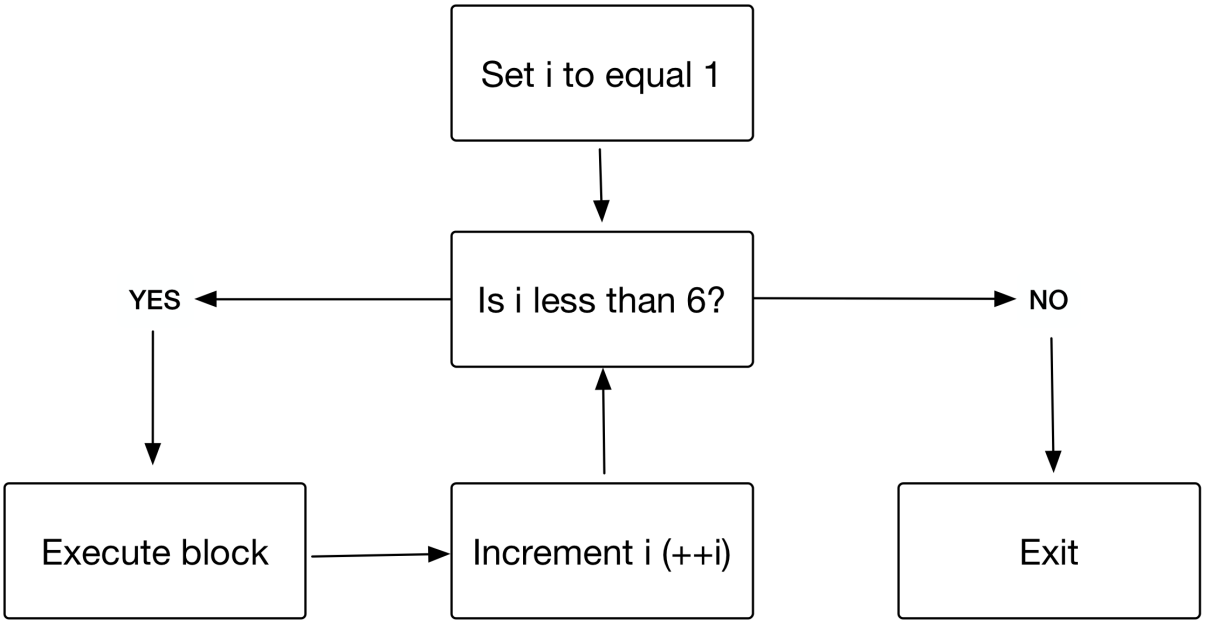
To make this loop more concrete, refactor the implementation of the `for-in` loop we made at the beginning of this chapter to use the more traditional form.

```
for var i = 1; i <= 5; ++i {
    ++int
    println(int)
}
```

As you can see, the implementation is similar and the result is the exact same. Like the `for-in` loop, the incrementer `i` declared in the initialization segment of the `for` loop is available within the scope of the loop (i.e., the code inside the braces of the loop).

Figure 6.x diagrams the flow of execution described in the code listing above.

Figure 7.4 For Loop Diagram



while Loops

The classical `for` loop we discussed above can actually be expressed as a `while` loop.

```
var i = 1
while i < 6 {
    ++int
    println(int)
    ++i
}
```

Like the `for` loop above, this `while` loop initializes an incrementer (`var i = 1`), evaluates a condition (`i < 6`), increments a counter (`++i`), and then returns to the top of the `while` loop to determine if the loop should continue iterating.

`while` loops are best for circumstances wherein the number of iterations the loop will pass through is unknown for some reason at the time it is written. For example, imagine a simple space shooter game with a spaceship that continuously fires its blasters so long as the spaceship has shields. Various external factors may lower or increase the ship's shields, but so long as the ship's shields has a value greater than 0, the spaceship will keep shooting. The code snippet below illustrates a relatively straightforward implementation of how this behavior might be accomplished.

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

do-while Loops

The principle difference between the `while` loop and the `do-while` loop is that the former first evaluates its condition before stepping into the loop. This difference means that the `while` loop may not ever execute depending upon whether or not it can initially pass its condition. In distinction, the `do-while` loop executes its loop at least once, and then evaluates its condition. The syntax for the `do-while` loop demonstrates this difference.

You can refactor the `while` loop above to avoid a somewhat depressing scenario: what if the user's spaceship is created, and by some freak accident, immediately loses all of its shields? Perhaps our spaceship was spawned immediately in front of an oncoming asteroid. In this case, the spaceship wouldn't even get to fire a shot. How boring. We wouldn't have been given a chance to blow anything up! That would be a pretty poor user experience. A `do-while` loop will help to avoid this anticlimactic event.

```
do {
    println("Fire blasters!")
} while shields > 0
```

The `do-while` loop reorganizes our previous `while` loop to ensure that we at least get to fire our ship's blaster one time before the loop's condition is evaluated. As you can see, the code block that contains the line `println("Fire blasters!")` is executed first, and then the `do-while` loop's condition is evaluated to determine if the loop should continue iterating. Thus, the `do-while` loop ensures that our spaceship gets to fire its blasters at least one time.

Control Transfer Statements, Redux

Let's revisit *control transfer statements* in the context of loops. Recall from the chapter on `switch` statements that *control transfer statements* change the typical order of execution. In the context of a loop, this means that you can control whether or not execution recycles once again to the top of the loop or leaves the loops all together.

Consider the former example: imagine that you want to stop the loop where it is, and begin once again from the top. Use the `continue` *control transfer statement* to do accomplish this task. To understand how this works, refactor the `while` loop your wrote above to be the following, and add it to your `Playground`.

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

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

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }

    println("Fire blasters!")

    ++blasterFireCount
}
```

You've added a good bit of code, so let's break it down. First, you added some variables that keep track of the following information: 1) `shields` are of type `Int` and are initialized to be equal to 5, 2) `blastersOverheating` is a `Boolean` initialized to `false` that keeps track of whether or not the blasters need time to cool down, and 3) `blasterFireCount` is a variable of type `Int` that keeps track of the amount of shots that the spaceship has fired so that we know whether or not the blasters will overheat.

Next, you have two `if` statements that check whether or not the blasters are overheating and the fire count respectively. If the blasters are overheating, then you log this information to the console. The **`sleep()`** function tells the system to wait for 5 seconds, which our way of modeling the blasters' cooldown phase. You next log that the blasters are ready to fire again (after waiting for 1 more second, which you do simply because doing so makes it easier to see what logs to the console next), set `blasterOverheating` to be equal to `false`, and also reset `blasterFireCount` to be 0. Last you used `continue` to jump to the beginning of the loop. Thus, with our shields intact, and our blasters cooled down, your spaceship is ready to fire away.

The second conditional checks to see if `blasterFireCount` is greater than 100. If this conditional evaluates to `true`, then you set the `Boolean` for `blastersOverheating` to be `true`. At this point, our blasters are overheated, and so you need a way to jump back up to the top of the loop such that your spaceship doesn't fire when its blasters are in this unstable state. As you saw in the previous paragraph, you used `continue` to jump back up to the top of the loop. Since the spaceship's blasters have overheated, the conditional in the first `if` statement will evaluate to `true`, and the blasters will shut down for 5 seconds to cool off.

After the second conditional is evaluated, you log to the console as before. Your last line, however, is new in that 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.

It's worth noting that this code will execute indefinitely. 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. For example, imagine that the game is over after the user has destroyed 500 space demons. Let's add a `break` *control transfer statement* to jump out of the loop when this event occurs.

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

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

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

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    println("Fire blasters!")
    ++blasterFireCount
    ++spaceDemonsDestroyed
}
```

In the above, you added a new variable called `spaceDemonsDestroyed`. This variable will keep track of the number of space demons that the user destroyed, which we arbitrarily increment each time the blasters fire. You're a pretty good shot, after all. Next, you checked to see if the `spaceDemonsDestroyed` variable is equal to 500, and if so, you logged victory to the console. Note the use of `break` statement above. 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 some sense for our example: if the user has destroyed 500 space demons, then the game is won and the blasters don't need to be fired anymore.

Conclusion

You are now acquainted with Swift's loops! Take a moment to consider what you have learned in this chapter.

1. The `for-in` loop.
2. The traditional `for` loop.
3. Type inference.
4. The `while` loop.
5. The `do-while` loop.
6. *Control transfer statements* in the context of loops.

That's quite a bit of material! Congratulations. You should be feeling pretty good about yourself. Be sure to try the challenge below to further sharpen your skills.

Challenge

Use a loop to count by 2 up to 100 from 0. Use another loop to make sure the first loop is run five times. Hint: one good way to do this is to use a nested loop.

8

Strings

In programming, textual content is represented by strings. As you well know, "Hello, playground" represents an ordered collection of characters, each of which represents a letter in the English alphabet. The combination of these letters forms a word. In fact, you have already seen a string in action. The Playgrounds that you have seen have started out with the string "Hello, playground" at the top.

Working with Strings

In Swift, strings are created with the `String` type. Create a new Playground to create a few `String` instances. Call the Playground `Strings.playground`, and save it somewhere convenient.

Highlight and delete everything in the Playground. You are going to create and work with your own strings. Add the following code to represent a new instance of the `String` type.

```
let playground = "Hello, playground"
```

You have created a `String` instance named `playground`. `playground` was created using the string literal syntax. The string literal syntax 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, then the compiler will give you an error.

Create a new string, but make this instance mutable.

```
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. Do so by adding the following code.

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

You used the addition and assignment operator (+=) to add an exclamation point to `mutablePlayground`. Take a look at the results sidebar on the right hand side of the Playground. You should see that the sidebar is showing that the instance has changed: `"Hello, mutable playground!"`.

String are composed of ordered sequences of characters. In fact, the characters that comprise Swift's strings are of a particular type: the `Character` type. Swift's `Character` type is used to represent Unicode characters, and are used in combination to formulate a `String` instance.

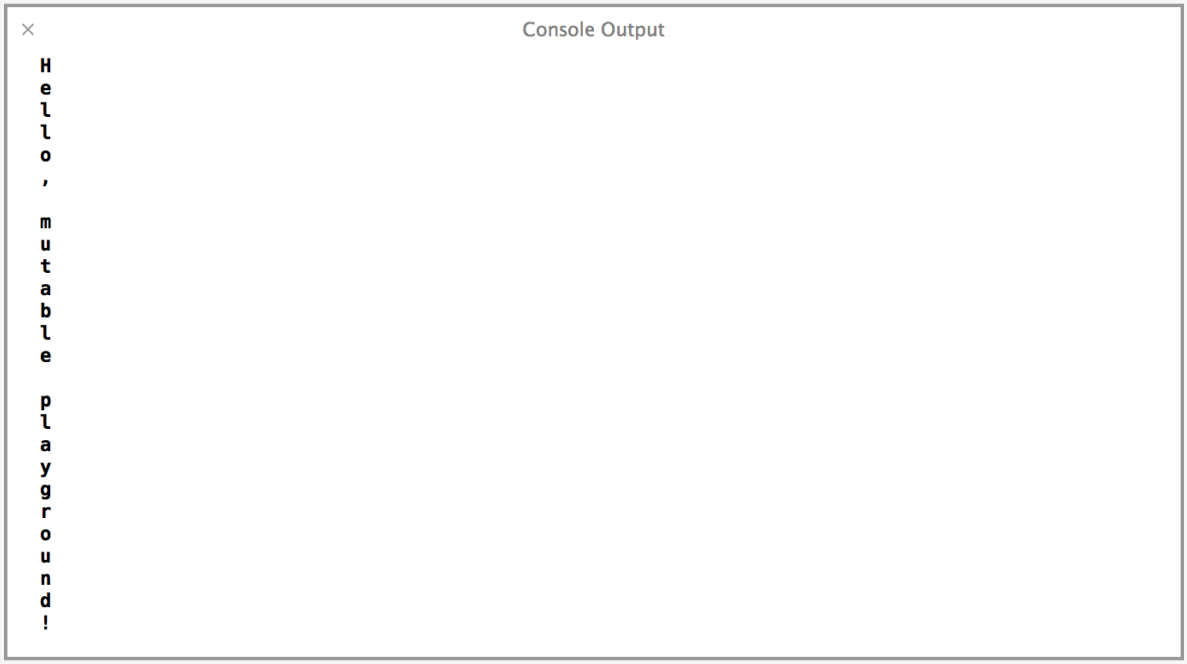
For now, loop through the `mutablePlayground` string to see the `Character` type in action.

```
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"
for c: Character in mutablePlayground {
    println("\(c)")
}
```

The loop iterates through every `Character` `c` in `mutablePlayground`. Each iteration logs the character to the console. Every character is logged to the console on its own line because you used the **`println()`**, which prints a line break after logging its content.

You should see the following log to the console when you reveal the Assistant Editor.

Figure 8.1 Logging Characters in a String



Unicode

Unicode is an internationally recognized standard that encodes characters so that they can be seamlessly processed and represented regardless of the platform. It is designed to represent human language (and other forms of communication like emoticons) on computers. This endeavor means that 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 comprised of what are called *Unicode Scalars*. Unicode scalars describe unique 21-bit numbers that are designated to represent a specific character in the Unicode standard. For example, `U+0061` represents the latin small letter 'a'. `U+1F60E` represents the smiley-faced emoticon 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

Type in the following to see how to use specific Unicode scalars in Swift and the Playground.

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

for c: Character in mutablePlayground {
    println("'\(c)'")
}
let oneCoolDude = "\u{1F60E}"
```

The new code above creates an instance of the `String` type. Notice that you used a new syntax: `"\u{1F60E}"`. The `\u{n}` syntax is written to denote a specific Unicode scalar `n`, where `n` is a unique hexadecimal number that represents a specific character. Therefore, the instance is assigned to be equal to the character representing an emoticon.

How does does this relate to more familiar strings? It turns out that every string in Swift is composed of these Unicode scalars, as indicated above. A few more concepts are useful in explaining this idea.

Every character in Swift is built up from one or more Unicode scalars. One Unicode scalar maps onto one specific fundamental character in a given language. For example, `U+0301` represents the Unicode scalar for the combining acute accent: ´. This scalar is described as combining because it will be placed on top of, or combined with, the character that precedes it. You can use this scalar with the Latin small letter `'a'` to create the character `'a'` with an accent.

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

for c: Character in mutablePlayground {
    println("'\(c)'")
}

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

You should see the letter `'a'` with an accent above it in the results sidebar.

Notice that you created this character by combining two Unicode scalars. The first scalar, `U+0061`, corresponds to the Latin small letter `'a'`. The second scalar, `U+0301`, corresponds to the combining acute accent ´. Combining marks are drawn in association with another Unicode scalar.

This exercise shows that every character in Swift is what is called an extended grapheme cluster. Extended grapheme clusters are a sequence of one or more Unicode scalars. In the most recent example, you decomposed the Latin small letter `'a'` with an accent into its two constituent Unicode scalars: the letter and the accent.

Swift also provides a mechanism that you can use to see the actual Unicode scalars your string is made up of. For example, you can see all of the Unicode scalars that Swift uses to create the instance of `String` named `playground` that you created above. Type in the following code to see.

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

for c: Character in mutablePlayground {
    println("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"

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

The constant `playground` has a property called `unicodeScalars`. Do not worry about what a property is right now; this topic will be covered in detail in Chapter 16. For now, all that you need to know is that a property is a way that a type holds onto data. In this case, `unicodeScalars` holds onto all of the scalars that Swift uses to make the string.

Open on the Assistant editor to view the console. You should see the following output at the bottom: `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 held onto data representing all of the Unicode scalars that were used to create the string instance `playground`. That means each number on the console corresponds to a Unicode scalar representing a single character in the string. Each of these numbers is represented as an unsigned 32-bit integer. Thus, `72` converts to the Unicode scalar value of `U+0048`, or an uppercase `'H'`.

Canonical Equivalence and Applications

Unicode provides a specific scalar for the character Latin small letter `'a'` with an accent above it. That means you do not need to decompose that specific character into its two parts: the letter and the accent. It comes precomposed. The scalar is `U+00E1`. Create a new constant string that uses this Unicode scalar.

```
...
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`. The important difference between the two constants is the `aAcute` was created using two Unicode scalars, and `aAcutePrecomposed` only used one. Indeed, if you check to see if these two characters are the same, you will find that Swift answers yes.

```
let aAcute = "\u{0061}\u{0301}"

for scalar in playground.unicodeScalars {
print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

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

Why is this result the case? It feels somewhat strange that this comparison says that the two characters are the same because you used a different set of Unicode scalars to create each one. The answer to this question involves something called *canonical equivalence*.

Canonical equivalence refers to whether or not two extended grapheme clusters are the same linguistically. Two characters, or two strings, are thereby considered equal if they have the same linguistic meaning and appearance, regardless of the potentially differing Unicode scalars that were used to make them. For example, `aAcute` and `aAcutePrecomposed` are equal strings because both represent the Latin small letter `'a'` with acute accent, even though each character was created with a different set of Unicode scalars. Thus, even though the resulting extended grapheme clusters are different, they are treated as canonically equivalent because the resulting characters have the same linguistic meaning and appearance.

Counting Elements

Canonical equivalence has implications for counting elements of a string. You might think that `aAcute` and `aAcutePrecomposed` would have different character counts because the number of Unicode scalars used to compose each string is different. Write the following code to check.

```
let aAcute = "\u{0061}\u{0301}"

for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"
let b = (aAcute == aAcutePrecomposed)
println("aAcute: \(countElements(aAcute));
         aAcuteDecomposed: \(countElements(aAcutePrecomposed))")
```

You used the **`countElements()`** function to determine the character count of these two strings. This function iterates over a character or string's Unicode scalars to determine the length. The results sidebar reveals that the character counts are the same: both are 1 character long. Canonical equivalence helps to explain why these two strings have the same number of characters. Both represent the Latin small letter `'a'` with an acute accent.

Indices and Ranges

Given that strings are an ordered collection of characters, you might think that you can access a specific character on a string like so:

```
let index = playground[3] // 'y'???
// error: 'subscript' is unavailable: cannot subscript String with an Int
```

As you can see, 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 as extended grapheme clusters. The result is that 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 an opaque type called a `String.Index` to keep track of indices. If you want to find the character at a particular index, then you can use the `String` type's `startIndex` property. This property yields the starting index of a string as a `String.Index`. You can then use this starting point in conjunction with the **`advance()`** to move forward until you arrive at the position of your choosing. Imagine, for example, that you want the fifth character of the `playground` string that you created at the beginning of this chapter.

```
...
let fromStart = playground.startIndex
let toPosition = 4
let end = advance(fromStart, toPosition)
let fifthCharacter = playground[end] // 'o'
```

You used the `startIndex` property on the string to get the very first index of the string. This property yielded an instance of type `String.Index`. Next, you created a constant to hold onto the position within the string that you would like to advance to. 4 represents the fifth character because the string is zero-indexed (i.e., 0 is the first index). Next, you used the **`advance()`** to advance from this starting point to your desired position. The result is a `String.Index` that you can use to subscript your `playground` string instance, which resulted in the character `'o'`.

Ranges, like indices, depend upon the `String.Index` type. Imagine that you wanted to grab the first five characters of `playground`. You can use the `fromStart` and `end` constants that you have already made.

```
...
let fromStart = playground.startIndex
let toPosition = 4
let end = advance(fromStart, toPosition)
let fifthCharacter = playground[end] // 'o'
let range = fromStart ... end
let firstFive = playground[range] // 'Hello'
```

The syntax `fromStart ... end` created a range of type `String.Index`. You were then able to use this new range as a subscript on the `playground` string. This subscript grabbed the first five characters from `playground`. The result is that `firstFive` is a constant equal to `"Hello"`.

Conclusion

In this chapter, you learned how Swift processes and represents strings. You learned that strings are made out of characters. Along the way, you discovered that characters are actually made out of extended grapheme clusters. This discovery led you to explore Unicode, and demonstrated how the standard mapped numerical Unicode scalars to specific characters. Finally, all of this knowledge found practical application in how Swift represents indices and ranges in its `String` type.

Challenge

Bronze Challenge

The `String` type has a method that will return a `Bool` indicating whether or not a string begins with a certain prefix. Find this method, and use it to determine if the `playground` string begins with the string `"Hello"`.

Silver Challenge

Redo the Bronze challenge above by replacing the `"Hello"` string with an instance created entirely out of its corresponding Unicode scalars. You will need to do a little research on the Internet to find the appropriate codes.

9
Optionals

Optionals are a special feature in Swift that are used to indicate that an instance may not have a value. When you see an optional, you know one of two things about that instance: 1) It has a value, and it is ready for use, or 2) it has no value. If an instance has no value associated with it, we say that it is *nil*.

You can use optionals with any type to indicate that an instance may not have a value. Put differently, you can use an optional with any type to signal that it is potentially `nil`. This feature distinguishes Swift from Objective-C, which only allows objects to be `nil`.

This chapter will cover how to declare optional types such that you can begin to use optionals, will show you how to use *optional binding* to check if an optional is `nil` and make use of its value if it does have a value, and will introduce *optional chaining* to query a sequence of optional values.

Optional Types

Optionals in Swift make the language more safe. Recall that an instance that is declared to be an optional type may potentially be `nil`. In distinction, if an instance is not declared as an optional type, then this instance is guaranteed to not be `nil`. Using optionals means that the compiler knows for certain if an instance can be `nil`. This explicit declaration makes your code more expressive and safe as a result. Let's take a closer look at how we declare an optional type.

To begin, create a new `Playground` and name it Optionals. Begin by making sure your code matches the snippet below.

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
```

First, you made a variable named `errorCodeString` to hold onto error code information in a string format. Second, you explicitly declared the type of `errorCodeString` to be `String`. Note, however, that you specified the type of the `errorCodeString` a little differently than you had before. This time you appended a `?` to the end of `String`. The `?` means that `errorCodeString` is an optional of the type `String`. In other words, you declared `errorCodeString` in a way that plans for the contingency that it may contain no value.

Now that you have declared an optional, and have given it a value, it is time to make some use of it. Log the value of the optional to the console like so:

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
println(errorCodeString)
```

Since you have given `errorCodeString` a value of `"404"`, logging its value to the console works fine. If you hadn't given `errorCodeString` a value, then `nil` would have been logged to the console. Go ahead and try it!

But logging `nil` to the console isn't very helpful. More broadly, you will likely want to know when your variables are `nil`, and will want to execute the appropriate code depending upon whether or not there is a value. You can use a conditional to help gain traction on a variable's value in these circumstances. For example, let's

say that if some operation generated an error, then we want to assign that error to a new variable and log it to the console. Add the following code to your `Playground`.

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
println(errorCodeString)
if errorCodeString != nil {
    let theError = errorCodeString!
    println(theError)
}
```

You created a new constant called `theError` to hold the value of `errorCodeString`. To do this, you appended a `!` to `errorCodeString`. The exclamation mark does what is called *forced unwrapping*. The exclamation mark accesses the underlying value of the optional, which allows us to grab `"404"` and assign it to our constant `theError`. We say that it does so "forcibly" because it tries to access the underlying value whether or not there is actually a value there at all. That is, the `!` assumes there is a value; if there is no value, unwrapping the value in this way would lead to a runtime error. Finally, you logged this new constant's value to the console.

What would have happened if we didn't unwrap `errorCodeString`'s value and simply assigned the optional to the constant `theError`? The error would still have been logged to the console correctly. So, why did you unwrap the optional's value and assign it to a constant? The answers to these questions involves having a better understanding of what the optional type is.

If you omitted the exclamation mark at the end of `errorCodeString`, then you would have simply assigned the optional `String` to a constant instead of the actual `String` value for the error code. In fact, `errorCodeString`'s type is `String?`. `String?` not the same type as `String` - if you have a `String` variable, you cannot set it to the value of a `String?` without unwrapping the optional.

The optional `errorStringCode` is `nil` when it is first declared because it was given no value. In the next line, you assigned `"404"` to `errorCodeString`, and so the value for the optional became `Some "404"`. You can compare an optional value to `nil` to determine whether or not it contains a value. In the code above, you first check whether the `errorCodeString` has a value; if it is not equal to `nil`, you know it is safe to unwrap `errorCodeString`.

Creating a constant inside the conditional is a little clunky. Fortunately, there is a better way to conditionally bind an optional's value to a constant. It's called *optional binding*.

Optional Binding

Optional binding is a useful pattern to detect if an optional contains a value. If there is a value, then we will assign it to a temporary constant or variable and make it available within a conditional's first branch of execution. In this manner, it can help to make your code more concise while also retaining its expressive nature. Here is the basic syntax:

```
if let temporaryConstant = anOptional {
    // Do something with temporaryConstant
 } else {
    // There was no value in anOptional; i.e., anOptional is nil
}
```

With this syntax in hand, refactor the example above to make use of optional binding.

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    let theError = errorCodeString!
    println(theError)
}
```

As you can see, the example is more or less the same. In fact, all you have really done is move the constant `theError` from the body of the conditional to its first line. `theError` is now a temporary constant that is

available within the first branch of the conditional. In other words, if there is a value within the optional, then a temporary constant is made available for use in the conditional in the block of code that is executed if the condition is evaluated as `true`. Notice that you did not have to forcibly unwrap the optional. If the conversion is successful, then this operation is done for you and the optional's value is made available to you in the temporary constant you declared. Also note that you could have declared `theError` with the `var` keyword if you needed to manipulate the value inside the first branch of the conditional.

Implicitly Unwrapped Optionals

At this point it is worth mentioning implicitly unwrapped optionals, though you won't make much use of them until we discuss classes and class initialization later on in the book. Implicitly unwrapped optionals are just like regular optional types, but differ in one important respect: you don't need to unwrap them. How is that the case? It has to do with how you declare them. The code below refactors our example above to work with an implicitly unwrapped optional.

```
import Cocoa

var errorCodeString: String!
errorCodeString = "404"
println(errorCodeString)
```

This reimagining of our chapter's example redeclares the optional with the `!`, which signifies that it is an implicitly unwrapped optional, and does away with the conditional. The conditional is removed because using an implicitly unwrapped optional signifies a great deal more confidence than its more humble counterpart. Indeed, much of the power and flexibility associated with the implicitly unwrapped optional is related to the idea that you don't need to unwrap it to access its value.

Note, however, that this power and flexibility comes with some danger: accessing the value of an implicitly unwrapped optional will result in a runtime error if it does not have a value. For this reason, we suggest that you do not use the implicitly unwrapped optional if you believe that the instance has any chance of becoming nil. Accordingly, their use is limited to somewhat special cases. As we indicated above, the primary case concerns class initialization. We'll return to this concept in the chapter on initialization. For now, you know enough of the basics of implicitly unwrapped optionals to understand what is going on if you find them out in the wild.

Optional Chaining

Like optional binding, optional chaining provides a mechanism for querying an optional to determine if it contains a value. One important difference between the two is that optional chaining allows the programmer to chain numerous queries into an optional's value together. If each optional in the chain contains a value, then the call to each succeeds, and the entire query chain will return an optional of the expected type. If any optional in the query chain is `nil`, then the entire chain will return `nil`.

Let's begin with a concise example. Add the following to your `Playground`.

```
var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    println(theError)
}

var errorCodeInt: Int?
errorCodeInt = errorCodeString?.toInt()
```

You first declared `errorCodeInt` to be of the `Int?` type via the familiar question mark. Next, you assigned the integer value of `errorCodeString` to your new variable. To do this, you made use of a function called **`toInt()`**. **`toInt()`** is a function defined on the `String` type that converts a string to its integer form if possible. If this conversion is not possible, then the function returns `nil`. For example, the string `"4"` can be converted to an integer, but the string `"Four"` cannot. Thus, in reality, **`toInt()`** returns an optional of type `Int?`.

The question mark appended to the end of the `errorCodeString` signals that this line of code initiates the optional chaining process. In the example above, you can intuit that `errorCodeInt` contains a value;

however, this intuition may be harder to find in a more complicated program you're sure to encounter in your development. Use optional binding in this case to test if there is a value in the result.

```
var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    println(theError)
}

var errorCodeInt: Int?
errorCodeInt = errorCodeString?.toInt()

if let errInt = errorCodeInt {
    println("The integer error code is \(errInt).")
} else {
    println("errorCodeString was either nil or could not be converted to an integer.")
}
```

If `errorCodeInt` has a value, then we unwrap it via optional binding and assign it to a temporary constant. We can subsequently access the value inside the first branch of the condtional statement. Otherwise, if there is no value, then we handle this scenario in the conditional statement's second branch.

So far we haven't leveraged the fact that optional chaining can be used on a sequence of optionals of an arbitrary length. Since **`toInt()`** returns an optional type, `Int?`, you can chain another function call. Imagine that you want to add an arbitrary integer to the error code for some reason.

```
var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    println(theError)
}

var errorCodeInt: Int?
errorCodeInt = errorCodeString?.toInt()?.advancedBy(100)
if let errInt = errorCodeInt {
    println("The integer error code is \(errInt).")
} else {
    println("errorCodeString is either nil or could not be converted to an integer.")
}
```

The function **`advancedBy()`** returns an `Int` by adding an integer value, provided inside the parentheses as an argument, to another integer. Thus, if `errorCodeString` and **`toInt()`** both have values, then the result is advanced by 100. As before, it is important to note that the result will of type `Int?`. That is, the result will be an optional integer.

The Nil Coalescing Operator

A common operation when dealing with optionals is to want to either get the value if the optional contains value, or to use some default value if the optional is nil. For example, when parsing `errorCodeString`, you might want to default to 500 (the HTTP code for Internal Server Error) if the string does not contain a valid integer. You could accomplish this with optional binding:

```
var actualErrorCode: Int
if let errInt = errorCodeString?.toInt() {
    actualErrorCode = errInt
} else {
    actualErrorCode = 500
}

println("actualErrorCode = \(actualErrorCode)")
```

There are two problems with this technique. The first is that it is just a lot of code to write for what should be a simple operation: get the value from the optional, or use 500 if the optional was nil. The second is that your use of optional binding means that you had to declare `actualErrorCode` as a `var` rather than a `let`, even though you don't want to change the value. Both problems can be solved via the *nil coalescing operator*:

```
var actualErrorCode: Int
if let errInt = errorCodeString?.toInt() {
    actualErrorCode = errInt
} else {
    actualErrorCode = 500
}

let actualErrorCode = errorCodeString?.toInt() ?? 500
println("actualErrorCode = \(actualErrorCode)")
```

`??` is the nil coalescing operator. The left-hand side must be an optional - `errorCodeString?.toInt()` in your case, which is an optional `Int`. The right-hand side must be a non-optional of the same type - 500 in your case, which is an `Int`. If the left-hand side optional is nil, `??` returns the right-hand side. If the left-hand side optional is not nil, `??` returns the value contained in the optional.

Try changing `errorCodeString` so that it does not contain an parsable integer, and confirm that `actualErrorCode` gets the value 500:

```
errorCodeString = "404"
errorCodeString = "an error occurred"
```

Conclusion

This chapter was fairly involved. You learned a lot of new material. Optionals are a new topic regardless of your level of experience in Mac or iOS development. Nonetheless, optionals are a powerful feature of Swift. It is a frequent need to represent `nil` in an instance. Optionals help the developer to keep track of whether or not instances are `nil`, and provide a mechanism to respond appropriately.

If optionals don't quite feel comfortable yet, don't worry. You'll be seeing them quite a bit in future chapters. They are an important part of Swift.

Challenge

Earlier in the chapter we told you that accessing an optional's value when it is `nil` will result in a runtime error. Make this mistake by force-unwrapping an optional when it is `nil`, examine the error, and understand what this error is telling you.

Part III

Collections and Functions

10
Arrays

An important task in programming entails grouping together logically related values. For example, imagine your application keeps a list of a user's friends, favorite books, travel locations, and so on. It's often necessary to be able to keep those values together and pass them around your code. Collections make these operations convenient.

Swift offers two collection types. The first that we'll cover is called an `Array`. The second, and the subject of the next chapter, is called a `Dictionary`. Arrays are an ordered collection of values. Each position in an array is identified by an index, and any value can appear multiple times in an array. Arrays are typically used when the order of the values is important or useful to know for some reason, but it is not a prerequisite that the order of the values be meaningful. Unlike Objective-C, Swift's `Array` type can hold onto any sort of value -- objects and non-objects alike.

To get started, create and save a new Swift `Playground` called Arrays.

Creating an Array

In this chapter, you will create an array that will represent your bucket list, or, the things that you would like to do in the future. In `Arrays.playground`, type the following code:

```
import Cocoa

var bucketList: Array<String>
```

You created a new variable called `bucketList` that is of the type `Array`. Much of that syntax should look familiar to you. For example, the `var` keyword means that `bucketList` is a variable. This means that `bucketList` is mutable, which we use to indicate that the array can be changed. We'll discuss immutable arrays later in this chapter. What is new, however, is the following: `<String>`. This code tells `bucketList` what sort of instances it can accept. In the above, your new array will accept instances of the `String` type.

It's worth mentioning that this code is functionally the same as the above.

```
import Cocoa

~~var bucketList: Array<String>~~
var bucketList: [String]
```

The brackets tell `bucketList` that it is an instance of `Array`, and the `String` syntax tells `bucketList` what sort of values it can accept. Since the array will hold information concerning your future goals, it makes sense that it takes instances of the `String` type. Nonetheless, you could have specified any type you liked.

Your `bucketList` is only declared, but is not yet initialized. This means that it isn't yet ready to accept instances of the `String` type. For example, if you were to try to append an item to your `bucketList` with the code: `bucketList.append("Climb Mt. Everest")`, then you would get the following error: `Playground execution failed: error: <EXPR>:7:12: error: variable 'bucketList' passed by reference before being initialized`. What does this error mean? In short, it tells you that you are trying to add an instance to your `bucketList` before it has been properly prepared.

To fix this error, change your declaration of `bucketList` to initialize the array in the same line.

```
import Cocoa

var bucketList: [String] = ["Climb Mt. Everest"]
```

The = is the assignment operator, which we use in conjunction with the `Array` literal syntax `["Climb Mt. Everest"]`. An `Array` literal is a shorthand syntax that initializes an array for you with whatever instances you include with it. In this case, you initialized `bucketList` with the bucket item to climb Mt. Everest.

As with other types, you can create an instance of `Array` by taking advantage of Swift's type inference capabilities. Remove the type declaration from the code above to use type inference.

```
import Cocoa

var bucketList~~: [String]~~ = ["Climb Mt. Everest"]
```

Notice that nothing has really changed. Your bucket list still contains the same item, and will only accept instances of the `String` type in the future. If you were to try to add an integer to this array, perhaps using the **`append()`** you saw above, then you would see the following error: `Type 'String' does not conform to protocol 'IntegerLiteralConvertible'`. To translate, this error tells you that you cannot add an instance of `Int` to your array because it is expecting instances of the `String`.

Now that you know how to create and initialize an array, it's time to learn about how to access and modify your array's elements.

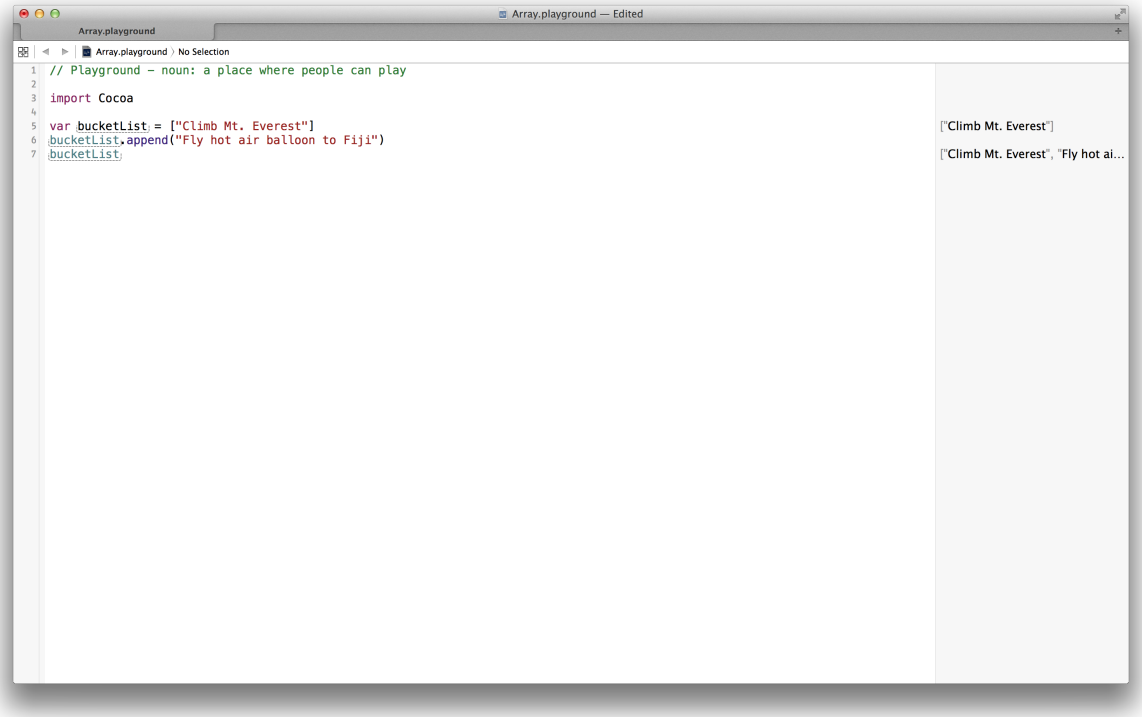
Accessing and Modifying Arrays

So, you have a bucket list? Great! Sadly, you don't have all that many ambitions in there yet. But you're an interesting person with a great zeal for life; so, let's change that. To do so, we need to know how to add values to `bucketList`. In fact, you've already seen how to do this. Go ahead and add the following code to your `Playground`.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList
```

Your `Playground` should look like the following.

Figure 10.1 Appending to Your Bucket List



Now, your bucket list has two future items in it. You use **`append()`** to append a value to `bucketList`. The function takes an argument of any type that will be the new element added to your array. In the above, you added `"Fly hot air balloon to Fiji"` as your new bucket list ambition. Remember, though, that your array only takes instances of the `String` type.

Add some more future adventures to your bucket list using the **`append()`** function.

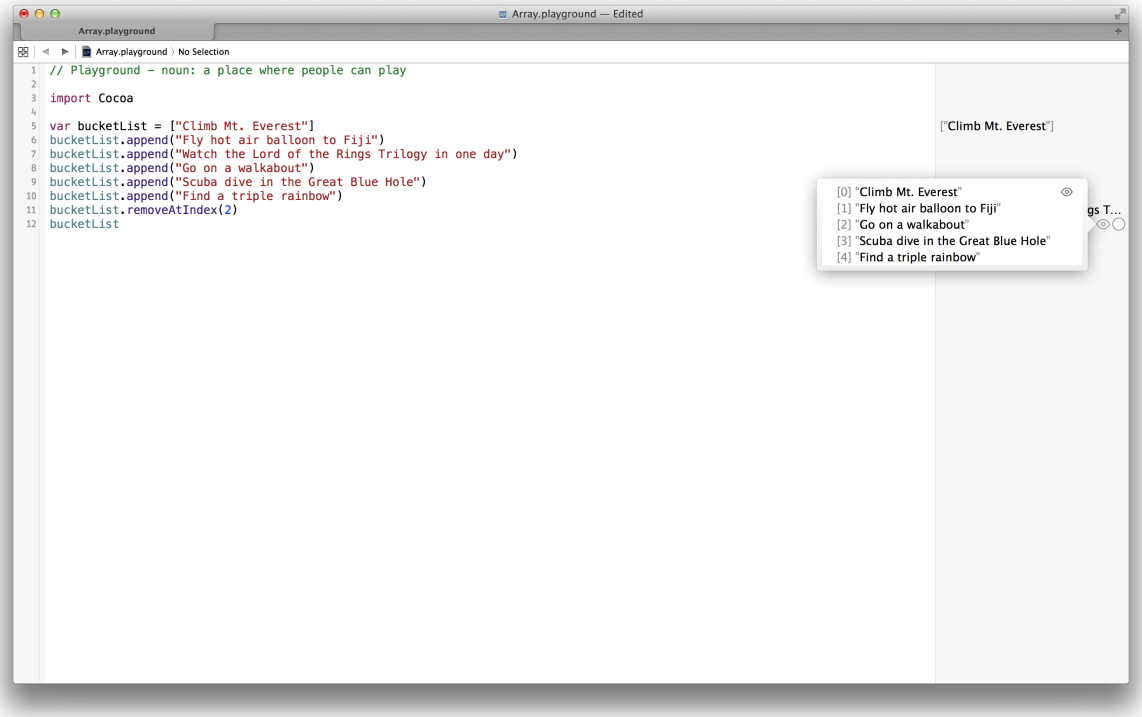
```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList
```

Now you have six items in `bucketList`. But what if you have a change of heart? Or, perhaps more positively, what if you actually do one of these items in your list? Let's imagine you got sick one Saturday, and had lots of free time. As such, you spent the day watching the *Lord of the Rings* series. Remove that item.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
```

To reveal the contents of `bucketList`, highlight line 12 in the results sidebar and click the button that looks like an eye -- this is called the "Quick Look". As you can see in the figure below, our item formerly at the second index was removed. Also note that arrays are zero-indexed. So, `bucketList[0]` is equal to `"Climb Mt. Everest"`.

Figure 10.2 Removing an Item from Your Bucket List



Now imagine that you're at a party, and the topic of bucket lists came up. After hearing what everyone has to say, you are convinced that yours is more interesting. As you begin to share, the crowd starts asking questions. "If your bucket list is sooo interesting, why don't you tell us how many things you want to do?"

No problem! It is easy to find the number of items in an array. Arrays are able to keep track of the number of items in them via the `count` property. Add the following code to your `Playground` to print to the console the number of bucket list items you have.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
println(bucketList.count)
```

"Five items," the crowd gasps, "that is a lot of things to do!" Since the party is winding down, and everyone needs to go home and rethink their personal bucket lists, they ask you simply tell them your top three.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
println(bucketList.count)
println(bucketList[0...2])
```

The bracket syntax (`[1...3]`) is what is called subscripting. Subscripting allows you to access specific indices in our array. Notice that you printed to the console our first three items. You could have logged only your last item via this code: `println(bucketList[3])`.

Subscripting is a powerful feature. Say that during your conversation someone asked, "Where do you want to do your walkabout?" The question made you realize that your third item needed some clarification. After all, you don't want to just go on a walkabout *anywhere*. Indeed, you want to go do your walkabout in Australia. You can use subscripting to change an item at a particular index (or range of indices).

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
~~bucketList~~
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList
```

If you click on the quick look button, you'll see that the third item now details that you want to go on a walkabout in Australia. How did this happen? You used the += addition and assignment operator to add some text to the item at index 2. This assignment worked because the instance at index 2 is of the same type as the instance you added to it; i.e., `"Go on a walkabout"` and `" in Australia"` are both of the `String` type.

Thinking about all of these adventures has actually gotten you thinking, and you're having trouble sleeping. Since reading usually helps, you start to read up on climbing Mt. Everest. In your reading, you find out that it is actually pretty dangerous. You think to yourself, "While I think of myself as fairly adventurous, I would rather have my adventure and be safe at the same time." You decide to update your top bucket item.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

There! That's better. Now your top bucket list item is more safe, and is still quite adventurous. And while you are now happy with the content of your bucket list, you're not all that happy with the fact that you typed `bucketList.append()` five times. You think to yourself: "There has to be a better way!"

And then you remember something: "I know how to use loops! What if I made an array of all the bucket list items I want to add? Then I could loop through that array and use **`append()`** each time the loop iterates. This means that I'll only have to type `bucketList.append()` but one time!" This is exactly what you do.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
~~bucketList.append("Fly hot air balloon to Fiji")~~
~~bucketList.append("Watch the Lord of the Rings trilogy in one day")~~
~~bucketList.append("Go on a walkabout")~~
~~bucketList.append("Scuba dive in the Great Blue Hole")~~
~~bucketList.append("Find a triple rainbow")~~
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]

for item in newItems {
    bucketList.append(item)
}
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

Your new code above creates an array for the bucket list items that you'd like to add called `newItems`. Next, you made a loop that iterated through each `item` in the array and appended it to your `bucketList`. As you have seen before, you took advantage of the fact that you can use the `item` variable in the local scope of the loop to append it to your bucket list array. "That's pretty good," you think, "but I bet I can do better." In fact, you can.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
~~for item in newItems {~~
    ~~bucketList.append(item)~~
~~}~~
bucketList.extend(newItems)
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

The **extend()** function takes an argument that conforms to the Sequence protocol. More on protocols later in this book, but for now, understand that protocols describe a sort of contract that defines the behavior of an instance that conforms to it. It just so happens that the Array type conforms to Sequence, which means that you can pass an instance of the Array to the argument of **extend()**. So, when you pass in your array newItems to the **extend()** function, it adds each item in that new array to your existing bucketList array.

You are about to go to sleep, happy in how you've refactored you code to make it more concise and keep it just as expressive, but then a thought strikes you in a bolt of inspiration. "I can use the addition and assignment operator!" Indeed you can.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
~~bucketList.extend(newItems)~~
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

The +=, what we call the addition and assignment operator, makes for a clean and easy way to concisely add your array of newItems to your existing bucket list.

Finally, say you want to change your number three item. While a walkabout in Australia is clearly very awesome, you think you'd actually prefer to toboggan across Alaska first. Use the **insert()** function to add an element to your array at a particular index.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
bucketList.extend(newItems)
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", atIndex: 2)
bucketList
```

The **`insert()`** function has two arguments. The first argument takes the instance that you would like to add to the array. Recall that your array takes `String` instances. The second argument takes the index for where you would like to add the new element in the array.

With your list fully formed, you lay your head down and dream of flying hot air balloons to mountain islands.

Array Equality and Identity

The next morning you wake up and go to your neighborhood coffee shop. There, you meet a friend, named Myron, you talked to at the party the night before. Your friend was absolutely inspired by your `bucketList`, and wanted to model his list after yours. He went home after the party and wrote out all of your items, but wants to now make sure that he got everything correct. After updating Myron with the changes you made after the party, it is time to compare your arrays of bucket list items to ensure that they are the same.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
bucketList.extend(newItems)
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", atIndex: 2)
bucketList

var myronsList = [
                  "Climb Mt. Kilimanjaro",
                  "Fly hot air balloon to Fiji",
                  "Toboggan across Alaska",
                  "Go on a walkabout in Australia",
                  "Scuba dive in the Great Blue Hole",
                  "Find a triple rainbow"
                 ]

let equal = (bucketList == myronsList)
```

Your new code uses the == to test for equality between two instances. Since the contents of both arrays are the same, `equal` is set to `true`.

Immutable Arrays

After reading the above, a question may have creaped into your mind: "What if I don't want my array to change?" You use immutable arrays for these cases. Here's how.

Let's say you were making an application that allowed users to keep track of their lunches week-to-week. Among many other very useful features, your app will allow its users to log what they ate and generate reports at a later time. You decide to put these meals in an immutable array when you generate these reports. After all, it doesn't make sense to change last week's lunches after you have already eaten them. Add the following code to your `Playground`.

```
var myronsList = [
                  "Climb Mt. Kilimanjaro",
                  "Fly hot air balloon to Fiji",
                  "Toboggan across Alaska",
                  "Go on a walkabout in Australia",
                  "Scuba dive in the Great Blue Hole",
                  "Find a triple rainbow"
                 ]

let equal = (bucketList == myronsList)
let lunches = ["Cheeseburger",
               "Veggie Pizza",
               "Chicken Caesar Salad",
               "Black Bean Burrito",
               "Felafel wrap"
      ]
```

You use the `let` keyword to create an immutable array. If you were to try to modify the array in any way, then the compiler will issue an error stating that you cannot mutate an immutable array. Indeed, if you were to try to reassign a new array to `lunches`, then you would also get an error from the compiler telling you that you cannot reassign an instance to a constant created via the `let` keyword.

Conclusion

This chapter introduced you to Swift's `Array` type. Along the way, you learned to use `var` to create a mutable array, and to use `let` to create an immutable array. Additionally, you learned that Swift's powerful type inferencing system allows you to be more concise while also taking advantage of Swift's type safety and checking. You also used a `for-in` to iterate through one array to add its items to another array. Next, you saw how to modify mutable arrays via the `Array` type's subscripting syntax and a suite of helper functions (e.g., **`append()`**, **`removeAtIndex()`**, **`insert()`**, **`extend()`**, and so on.). Finally, you checked for equality (via ==) between two arrays.

Challenge

Look at the array below. Use the documentation to find a function that will tell you whether or not the array has any elements inside of it. Next, use a loop to reverse the order of the elements of this array. Log the results to the console. Finally, examine the documentation to see if there is a more convenient way to do this operation.

```
var toDoList = ["Take out garbage", "Pay bills", "Cross off finished items"]
```

11
Dictionaries

In the previous chapter, you became familiar with Swift's `Array` type. The `Array` type is a useful collection for scenarios wherein the order of the elements inside the array instance is important. But order is not always important. Perhaps you simply want to hold onto a set of information in a container, and then later retrieve that information as needed. That's what dictionaries are for.

A `Dictionary` is a collection type that organizes its content by key-value pairs. The keys in a dictionary map onto values. A key operates similarly to a ticket you give to the attendant at a coat check. You arrive to the attendant, give your ticket, and the attendant uses the ticket to find your coat. Similarly, you give a key to an instance of the `Dictionary` type, and it will return to you the value that is associated with that key. The main point is that you should use a dictionary when you want to store and retrieve information with a specific key.

In this chapter, you will:

- learn how to create and initialized a dictionary,
- loop through dictionaries,
- access and modify dictionaries via their keys.

You will also learn more about keys and how they work, especially as they pertain to Swift. Last, you'll see how to create arrays out of your dictionary's keys and values.

Creating a Dictionary

The general form of a Swift dictionary is like so: `var dict: Dictionary<KeyType, ValueType>`. This code creates a mutable instance of the `Dictionary` type called `dict`. The declarations for what types the dictionary's keys and values accept are inside the angle brackets (`<>`), as denoted by `KeyType` and `ValueType`. `dict` will only accept instances for its keys and values of the appropriate type after this declaration.

The only requirement for Swift's `Dictionary` type's keys is that the key must be *hashable*. That is, each `KeyType` must provide a mechanism that allows `Dictionary` to guarantee that any given key is unique. Swift's basic types, such as `String`, `Int`, `Float`, `Double`, and `Bool` are all *hashable*.

Before you start typing code, let's take a look at the different ways you can use to get an instance of a `Dictionary`.

```
var dict1: Dictionary<String, Double> = [:]
var dict2 = Dictionary<String, Double>()
var dict3: [String:Double] = [:]
var dict4 = [String:Double]()
```

The listing above demonstrates four ways to get a dictionary instance. The resulting dictionary instance is empty in each of these cases; that is, the instance currently has no keys and no values.

Each yields the same result: a fully initialized instance of the `Dictionary` type with type information associated with its future keys and values. The `KeyType` is set to accept keys of the `String` type, and the `ValueType` is set to accept values of the `Double` type.

What is the difference between the use of the `()` and the `[:]` syntax above? Inasmuch as each line of code creates and prepares an instance of the `Dictionary` type, they are all essentially the same. The `[:]` code uses the literal syntax to create an empty instance of the `Dictionary` type that will infer type information for its keys and

values. For example, `dict1` specifies its type, and then is initialized to be an empty dictionary. The `()` syntax uses the default initializer for the `Dictionary` type, which will prepare an empty dictionary instance. You'll see more about intializers later on in this book.

As before, it is useful to take advantage of Swift's type inference capabilities. Type inference makes for more concise code, and is just as expressive. Accordingly, you'll stick with type inference in this chapter.

Populating a Dictionary

To get started, create a new file called `Dictionary.playground`. Make sure to save it where you can find it later. Add the following code to your `Playground`.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
```

In the code above, you created a mutable dictionary using the `Dictionary` literal syntax. This particular instance will hold movie ratings. Its keys are instances of the `String` type, and represent individual movies. These keys map onto values that are instances of the `Integer` type, which represent individual ratings of the movies in the `Dictionary` instance.

Accessing and Modifying a Dictionary

Now that you have a mutable dictionary, the question becomes how do you work with it? You will want to read from and modify the dictionary. Begin by using `count` to get a little information about your dictionary.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
```

`count` is a read-only property on the `Dictionary` type that keeps track of how many instances are held within the dictionary instance itself. We'll discuss properties in greater detail in a later chapter, but it suffices to say now that properties are variables on a type that store or compute some data about the type that you are interested in. In this case, you use `count` to log to the console: `I have rated 3 movies.`.

Let's now read a value from the `movieRatings` dictionary.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
```

You access values from a dictionary by supplying the key associated with the value you would like to retrieve. In the example above, you supply the key `"Donnie Darko"` to the dictionary of movie ratings. `darkoRating` is now set to be equal to the rating of 4.

Option-click on the `darkoRating` instance - notice that its type is `Int?`, but `movieRatings` has type `[String : Int]`. Why the discrepancy? The `Dictionary` type needs a way to tell you that the value you asked for is not present. For example, you have not rated Braveheart yet, so this: `let braveheartRating = movieRatings["Braveheart"]` would result in `braveheartRating` being `nil`.

Now you'll modify a value in your dictionary of movie ratings. Imagine, for example, that a user enters in a rating, and then later reconsiders his or her original rating.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
```

As you can see, the value associated with the key `"Dark City"` is now equal to 5.

There is another useful way to update values associated with a dictionary's keys. The **`updateValue()`** function takes two arguments: `value: ValueType, forKey key: KeyType`. The first argument, `value`, takes the new value. The second argument, `forKey`, specifies the key whose value you would like to change.

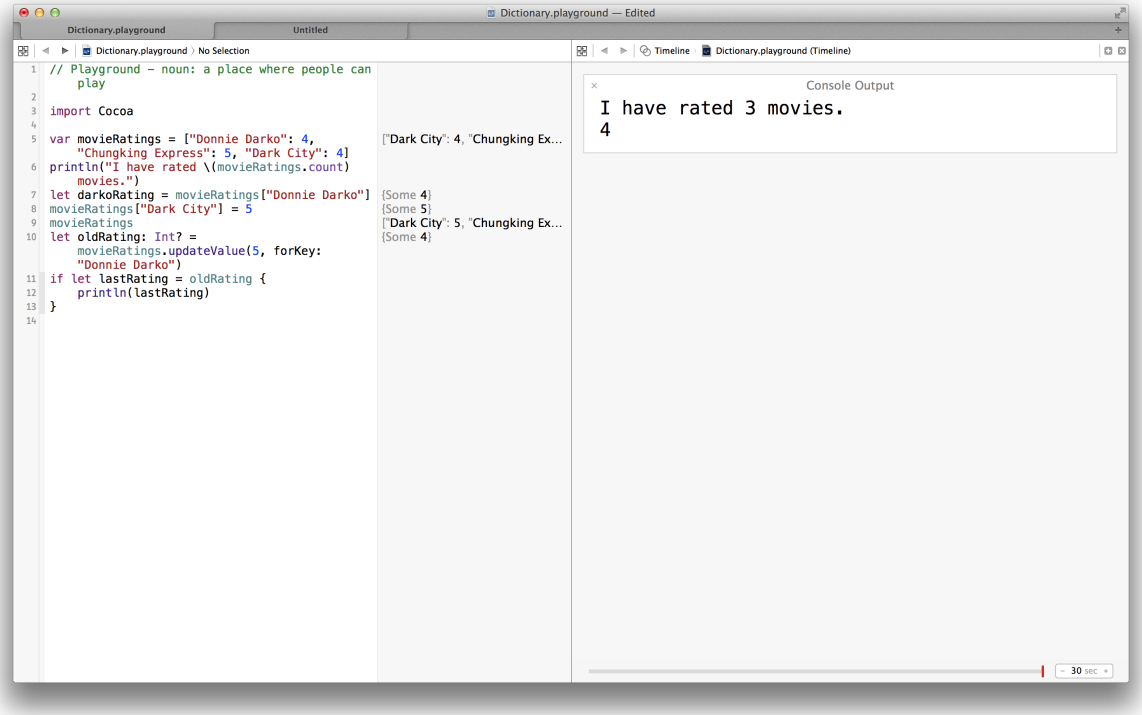
This function is useful because it gives you a handle on the last value the key mapped to. There is one small caveat: **`updateValue()`** returns an optional. This return type is handy because the key may not exist in the

dictionary. As a consequence, it is helpful to assign the return of the **`updateValue()`** function to an optional of the type that you are expecting and use implement optional binding to gain access to the key's old value. See below.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
```

The figure below shows that the old value for "Donnie Darko's" rating was logged to the console.

Figure 11.1 Updating a Value



Adding a Value

Now that you have seen how to update a value, it is natural to consider how you might add a value to a dictionary. Suppose that you have a user that would like to add a new movie rating to the `movieRatings` dictionary.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
```

You can add a new key-value pair to your dictionary by providing a new key to the dictionary via this syntax: `movieRatings["The Cabinet of Dr. Caligari"]`. You next use the *assignment operator* to associate a value this new key. In this case, you give the movie *The Cabinet of Dr. Caligari* a rating of 5.

Removing a Value

Following naturally from adding a value to a dictionary is the topic of removing a value from a dictionary. Imagine, for example, that your user thought she or he had seen the movie *Dark City*, but later realized that this was not the case.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
let removedRating: Int? = movieRatings.removeValueForKey("Dark City")
```

The function **`removeValueForKey()`** takes a key as an argument and removes the key-value pair that matches what you provide. As a result, `movieRatings` no longer contains an entry for *Dark City's* rating. Additionally, this function returns the value the key was associated with if the key was found and removed successfully. **`removeValueForKey()`** returns an optional of the type that was removed. In the example above, `removedRating` is an optional `Int`.

You can also remove a key-value pair by setting a key's value to `nil`.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
~~let removedRating: Int? = movieRatings.removeValueForKey("Dark City")~~
movieRatings["Dark City"] = nil
```

The result essentially the same, with the main difference being that this strategy of removing a key-value pair from a dictionary does not return the removed key's value.

Looping

You can use a `for-in` to loop through a dictionary. Swift's `Dictionary` type provides a convenient mechanism to loop through an instance that allows you to access each entry's key and value at each iteration in the loop. This mechanism breaks each element into its constituent parts by providing a temporary constant representing the key and the value.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
     println("The movie \(key) was rated: \(value).")
}
```

Open up the *assistant editor* to view the console. You should see that each movie and its rating was logged to the console. Notice how you used *string interpolation* to combine the values of `key` and `value` into a single string to log to the console.

A dictionary keeps track of its keys and values via the `keys` and `values` properties. You can use these to access specific information if you do not need both parts of the key-value pair. See the loop below for an example.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
     println("The movie \(key) was rated: \(value).")
}
for movie in movieRatings.keys {
     println("User has rated \(movie).")
}
```

This new loop will iterate through `movieRatings`'s key, and will thereby log each movie the user has rated to the console.

Immutable Dictionaries

Creating an immutable dictionary works much the same as it does in the case of the array. Use the `let` keyword to tell the Swift compiler that you do not want an instance of `Dictionary` to change. The example below creates an immutable dictionary that lists the track names of a fictional album (a short one...let's say it is an EP) along with each track's length in seconds.

```
let album = ["Diet Roast Beef": 268
             "Dubba Dubbs Stubs His Toe": 467
             "Smokey's Carpet Cleaning Service": 187
             "Track 4": 221]
```

The track names are the keys, and the lengths in seconds are the values. If you try to change this dictionary, the compiler will give you an error and prevent the change.

Translating a Dictionary to an Array

Sometimes it is helpful to pull out information from a dictionary and put it into an array. Imagine, for example, that you have a part of your application that doesn't care about how a given user has rated the movies in his or her queue. Instead, this portion of the application only wants to know what movies the user has rated. For example, if the user has rated a movie, this part of the application can assume that the user has already seen it, and will therefore not recommend it to the user to watch. In this case, it makes sense to send the list of movies that a user has rated as an instance of the `Array` type to this portion of the app.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
     println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
     println("The movie \(key) was rated: \(value).")
}
for movie in movieRatings.keys {
     println("User has rated \(movie).")
}
let watchedMovies = Array(movieRatings.keys)
```

You used the `Array()` syntax to create a new `[String]` instance. Inside the `()`, you passed in the dictionary's keys. The result is that `watchedMovies` is a constant instance of the `Array` type representing all of the movies a user has in the `movieRatings` dictionary.

Conclusion

This chapter introduced you to dictionaries. Dictionaries are a collection type that are useful for when you would like to associate a key with a specific value. You give a dictionary a key, and it will hand its value back to you. Like the `Array` type, dictionaries can be mutable and immutable. If the dictionary is mutable, then you can add to, delete from, and modify values within it. If the dictionary is immutable, then these methods are not available. Additionally, Swift provides a helpful looping mechanism that allows you to iterate through a dictionary and pull out key and value information at each pass through the loop. This pattern is helpful because it affords you more granular control over the constituent pieces of the dictionary.

Challenge

It is not uncommon to place instances of the `Array` type inside of a dictionary. For example, imagine that you have a dictionary that represents a state. This dictionary has keys that refer to counties. Each key should map onto an array that holds all of the zip codes within that county. To keep it short, pick a state and give it three counties of your choosing as keys. Next, add five zip codes to each county's array of zip codes. You can make these up if you don't want to add actual zip codes. Finally, log only the dictionary's zip codes. Your result should look something like the below:

```
Georgia has the following zip codes: [30306, 30307, 30308, 30309, 30310,
                                      30311, 30312, 30313, 30314, 30315,
                                      30301, 30302, 30303, 30304, 30305]
```

12
Functions

A function is a name that is related to a set of code that is used to accomplish some specific task. The function's name is created to describe the task the function performs. You have already used some functions. For example, **`println()`** is a function provided to you by Swift.

Functions execute code. They can define arguments that you can use to pass in data to the function to help it do its work. Functions can also return something after it has completed its work. You might think of a function as a hungry, little machine. You turn it on, and it chugs along, doing its work. You can feed it data, and if it is built to do so, it will return a new chunk of data that results from its work.

Functions are an extremely important part of programming. Indeed, a program is mostly a collection of related functions that combine to accomplish some work that provides some set of functionality. Accordingly, there is a lot to cover in this chapter; so, take your time and make sure that you are comfortable before moving on. Let's take a look at some examples.

A Basic Function

Create a new `Playground` called `Functions.playground`. Make sure your code matches what you see below.

```
import Cocoa

func printGreeting() {
     println("Hello, playground.")
}
printGreeting()
```

You define a function with the `func` keyword. After the `func` keyword, you declare the name of the function. In the example above, you specified a function named **`printGreeting()`**. The parentheses are empty because this function doesn't take any arguments. More on those soon. The opening brace ({) denotes the beginning of the function's implementation. This is where you write the code that describes how the function will perform its work. When the function is called, the code inside the opening and closing braces is executed.

The **`printGreeting()`** function is fairly simple. You have one line of code that uses the **`println()`** to log the "Hello, playground." string to the console. You need to call the function, however, to actually execute the code inside of it. To do so, you typed `printGreeting()` on the line following the definition of the function. Calling the function executed its code in the `Playground` and `"Hello, playground."` was logged to the console.

Now that you've written and executed a simple function, it's time to graduate to more sophisticated varieties.

Function Parameters

Functions really begin to take on more life when they have arguments. You use arguments to give a function some inputs. The function will then take the data in the arguments, and use the data to execute a particular task or produce some result. Change your existing function so that you can make a more personal greeting. To do this, you will need to add an argument.

```
import Cocoa

~~func printGreeting() {~~
~~    println("Hello, playground.")~~
~~}~~
~~printGreeting()~~
func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")
```

printPersonalGreeting() takes a single argument that is located within parentheses directly after the function name. In this case, you added an argument called `name` that expects to be an instance of the `String` type. You specified the type for `name` after the **:** that followed the argument's name, just like when you were learning about specifying the types for variables and constants.

So long as the argument is an instance of `String`, it will be interpolated into the string that is logged to the console. Check it out. Your console should say something like: “Hello Matt, welcome to your playground.” If you happened to pass an instance to the argument that wasn't of the `String` type, then the compiler will have given you an error telling you that the type you passed in is incorrect. This behavior is very helpful. It is useful to know what your inputs will look like when you are writing the implementations of your functions.

Functions can take multiple arguments. Indeed, this scenario is quite common. Make a new function that does a little math.

```
import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func divisionDescription(num: Double, den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(9, 3)
```

The function **divisionDescription()** describes some basic division constructed from the instances of the `Double` type that you supplied to the function's two arguments: `num` and `den`. Note that you did some math within the `\()` of the string printed to the console. You should now see, “9 divided by 3 equals 3” logged to the console.

Parameter Names

The functions above that have any arguments have parameter names for their arguments. For example, the function **divisionDescription()** has two arguments and two corresponding parameter names: `num` and `den`. Nonetheless, these names are only local within the function's body. You don't actually see these parameter names when you use **divisionDescription()**. They are there for your reference, but disappear when you give the function arguments values. Sometimes, however, it is useful to have parameter names visible outside of your function's body.

Named parameters can make your functions more readable. For example, **divisionDescription**'s arguments don't really have informative parameter names. This is more or less fine within the function's body; the function's implementation makes it clear what these arguments are for. But what if the function is going to be used in some other file in your application's code base? It may be a little difficult to infer what to give to the function's arguments, which will make the function less useful.

Change **divisionDescription**'s arguments to have explicit parameter names.

```
import Cocoa

func printPersonalGreeting(name: String) {
     println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

~~func divisionDescription(num: Double, den: Double) {~~
~~    println("\(num) divided by \(den) equals \(num / den)")~~
~~}~~
~~divisionDescription(9, 3)~~
func divisionDescription(numerator num: Double, denominator den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(numerator: 9, denominator: 3)
```

Now, **`divisionDescription()`** has two explicitly named parameters. These explicit parameter names, `numerator` and `denominator`, are much clearer than the more terse `num` and `den`. In addition, you have the parameter names visible when you use the **`divisionDescription()`** function. This visibility for the parameter names makes the function more readable.

Swift offers a shorthand syntax for naming external parameters. Shorthand external parameter syntax saves some typing. Prefix the parameter name with the # symbol to use this name as an external parameter name. Refactor your function to use the shorthand syntax.

```
import Cocoa

func printPersonalGreeting(name: String) {
     println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

~~func divisionDescription(numerator num: Double, denominator den: Double) {~~
~~    println("\(num) divided by \(den) equals \(num / den)")~~
~~}~~
~~divisionDescription(numerator: 9, denominator: 3)~~

func divisionDescription(#numerator: Double, #denominator: Double) {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescription(numerator: 9, denominator: 3)
```

As you can see, the shorthand syntax makes it such that you do not have to type two parameter names for the function. Previously, you typed an explicitly named external parameter *and* a named local parameter. The shorthand syntax simplifies the parameter naming process for you.

Variadic Parameters

A variadic parameter takes zero or more input values for its argument. Functions can have one variadic parameter. Additionally, it must be the final parameter in the list.

To make a variadic parameter, use three periods after the parameter's type: e.g., `names: String...`. The values provided to the argument are made available within the function's body as an array. In the previous example, `names` is available within the function's body and has the type `[String]`.

Refactor the **`printPersonalGreeting()`** function to have a variadic paramater.

```
import Cocoa

~~func printPersonalGreeting(name: String) {~~
~~     println("Hello \(name), welcome to your playground.")~~
~~}~~
~~printPersonalGreeting("Matt")~~

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron","Brian","John","Matt")

func divisionDescription(#numerator: Double, #denominator: Double) {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescription(numerator: 9, denominator: 3)
```

The **printPersonalGreeting()** has been replaced with a plural version: **printPersonalGreetings()**. Check the console. You should see that the function logged a personal greeting for each name that was supplied to the variadic parameter.

Default Parameter Values

Swift's parameters can take default values. If a function has parameters with default values, then place them at the end of the function's parameter list. Somewhat conveniently, if a parameter has a default value, then you can omit that parameter when calling the function. Of course, omitting the parameter means that the function will assume that it should use the parameter's default value. Refactor the **divisionDescription()** function to have a new parameter with a default value.

```
import Cocoa

func printPersonalGreeting(name: String) {
     println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron","Brian","John","Matt")

~~func divisionDescription(#numerator: Double, #denominator: Double) {~~
~~    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")~~
~~}~~
~~divisionDescription(numerator: 9, denominator: 3)~~
func divisionDescription(#numerator: Double,
                         #denominator: Double, punctuation: String = ".") {
    println("\(numerator) divided by \(denominator) equals
             \(numerator / denominator)\(punctuation)")
}
divisionDescription(numerator: 9, denominator: 3)
divisionDescription(numerator: 9, denominator: 3, punctuation: "!")
```

Notice the new code: `punctuation: String = "."`. You added a new parameter named `punctuation`, added its expected type, and gave it a default value via the `= "."` syntax. Thus, the string created by the function will conclude with a period by default.

You can use this new parameter inside the functions body as well. Also notice that giving the parameter a default value automatically created an explicitly named external parameter. As the above code snippet suggests, you can ignore the final parameter and default to its value given by the function's definition. You can also substitute this default value for a new punction mark. The first call of the **divisionDescription()** function will log the description with a period, and the second will log the description with an exclamation point.

In-out Parameters

Sometimes there is reason to have a function modify the value of a parameter. In-out parameters are useful if you want a function's impact upon a variable to live beyond the function's body. There are a couple of caveats: 1) in-out parameters cannot have default values, and 2) variadic parameters cannot be marked with `inout`.

Perhaps you have a function that will take an error message as an argument, and will append some information based upon certain conditions. Consider the example below.

```
var error = "The request failed:"
func appendErrorString(#errorCode: Int, inout #errorString: String) {
    if errorCode == 400 {
        errorString += " bad request."
    }
}
appendErrorString(errorCode: 400, errorString: &error)
```

The function **`appendErrorString`** has two arguments. The first one is the error code that the function will compare against, and the second is an `inout` parameter named `errorString`. Both parameters use the shorthand syntax, and each expects a `String` instance. As you can see from the example above, an in-out parameter is denoted by the `inout` syntax in the parameter list.

When you use the function, the variable you pass into the argument for the `inout` parameter is preceded by an ampersand (&). You do this to indicate that the variable will be modified by the function. In the example provided above, the `errorString` will be modified to read: “The request failed: bad request.”

In-out parameters are not the same as a function returning a value. If you'd like your function to produce something to be used later on, then there is a more elegant way to handle this scenario.

Returning from a Function

Functions can give you information after they finish executing the code inside of their implementation. This information is called the return of the function. It is often the case that you write a function to do some work and return to you some data. For example, refactor the **`divisionDescription`** to return an instance of the `String` type.

```
import Cocoa

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron","Brian","John","Matt")

~~func divisionDescription(#numerator: Double,~~
~~                         #denominator: Double, punctuation: String = ".") {~~
~~    println("\(numerator) divided by \(denominator) equals~~
~~             \(numerator / denominator)\(punctuation)")~~
~~}~~
~~divisionDescription(numerator: 9, denominator: 3)~~
~~divisionDescription(numerator: 9, denominator: 3, punctuation: "!")~~
func divisionDescription(#numerator: Double,
                         #denominator: Double, punctuation: String = ".") -> String {
    return "\(numerator) divided by \(denominator) equals
         \(numerator / denominator)\(punctuation)"
}
println(divisionDescription(numerator: 9, denominator: 5, punctuation: "!"))
```

The behavior of this new function is very similar to its previous implementation with an important twist: this new implementation returns a value. This return value is denoted by the `-> String` syntax. The -> syntax indicates that the function returns a value straight away. Whenever you see that syntax, you can expect that the

function will return something. As you can surmise, Swift functions specify the type of the instance they return after the `->` syntax. Since you want to log a string to the console, your function will return an instance of the `String` type. Last, you call this function within a call to **`println()`** to log the string instance to the console.

Swift's function definitions can be nested. Nested functions are declared and implemented within the definition of another function. This feature is useful when you need a function to do some work, but it is not important for it to have a life outside of its enclosing function. In this scenario, the nested function will not be available outside of its more general function. See below for an example.

```
import Cocoa

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron","Brian","John","Matt")

func divisionDescription(#numerator: Double,
                         #denominator: Double, punctuation: String = ".") -> String {
    return "\(numerator) divided by \(denominator) equals
         \(numerator / denominator)\(punctuation)"
}
println(divisionDescription(numerator: 9, denominator: 5, punctuation: "!")

func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)
```

The function **`areaOfTriangle()`** takes two arguments of type `Double`: a `base` and a `height`. **`areaOfTriangle()`** also returns a `Double`. Inside of this function's implementation, you declared and implemented another function called **`divide()`**. This function also takes no arguments and returns a double. Last, the **`areaOfTriangle()`** function calls the **`divide()`** and returns the result of this function.

Scope

The **`divide()`** function above makes use of a constant called `numerator`. Why does this work? This constant is defined within the **`divide()`** function's enclosing scope. Anything that is written within a function's braces ({}) is said to be enclosed by that function's scope. A function's scope describes the visibility an instance or function will have. It can be thought of a sort of horizon. Anything defined within a function's scope will be visible to that function; anything that is not is past that function's field of vision. `numerator` is defined within the **`divide()`** function's enclosing scope. Thus, `numerator` is visible to the **`divide()`** function because they share the same enclosing scope. In distinction, the **`divide()`** function is defined within **`areaOfTriangle()`** function's scope, and will not be visible outside of it. The compiler will give you an error if you try to call the **`divide()`** function outside of **`areaOfTriangle()`** function's scope.

Multiple Returns

Functions can return more than one value. Swift uses a data type called a *tuple* to do so, which you learned about in Chapter 6. To better understand how to use tuples, you are going to make a function that takes an array of integers and sorts them into arrays for even and odd integers.

```
func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)
func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}
```

You first declared a function called **`sortEvenOdd()`**. You specify this function to take an array of integers as its only argument. The function returns what is called a *named tuple*. You can tell that the tuple is named because its constituent parts are named: `evens` will be an array of integers, and `odds` will also be an array of integers.

Next, inside the implementation of the function, you initialized the `evens` and `odds` arrays to prepare them to store their respective integers. You then looped through the array of integers provided to the function's argument, `numbers`. At each iteration through the loop, you use the `%` operator, called the modulus division operator. As opposed to the normal dividend, modulus division returns the remainder of the division. Doing this operation will divide the current number in the array by 2, and check to see if the result is 0. If the result is 0, then the integer is even (because dividing it by 2 had no remainder) and you will append it to the `evens` array. If the result is not 0, then the division yielded a remainder and the integer at the current iteration is odd. Accordingly, you will add this integer to the `odds` array.

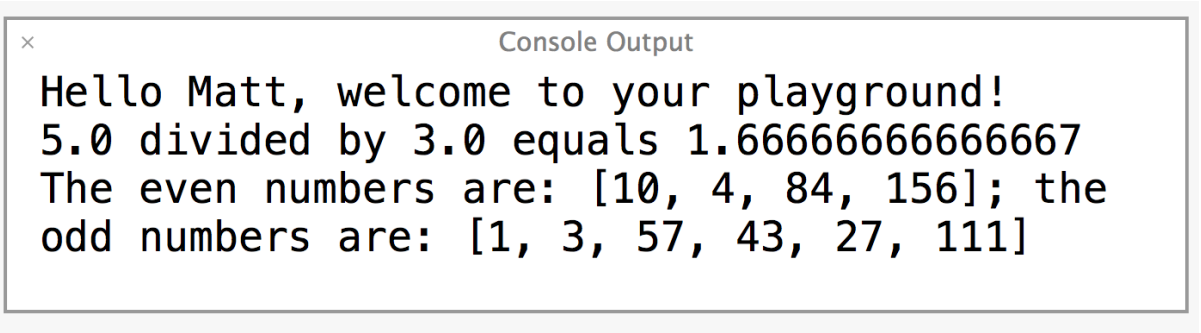
Add the new code below to exercise this function.

```
func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)

func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortEvenOdd(aBunchOfNumbers)
println("The even numbers are: \(theSortedNumbers.evens);
         the odd numbers are: \(theSortedNumbers.odds)")
```

First, you created an instance of the `Array` type to house a number of integers. Second, you give that array to the **`sortEvenOdd()`**, and assign the return value to a constant called `theSortedNumbers`. Because the return value is specified above as `(evens: [Int], odds: [Int])`, this is the type the compiler infers for your newly created constant. Finally, you log the result to the console. Notice that you used string interpolation in combination with a tuple. You can access a tuple's members by name if they are defined. For example, `theSortedNumbers.evens`

inserted the contents of the `evens` array into the string logged to the console. Your console output should match Figure 12.1.

Figure 12.1 Using a Tuple



Optional Return Types

Sometimes, you want a function to return an optional. In other words, you believe there is a chance that a function will need to return `nil` occasionally, but will also have a value to return at other times. In this case, you will use an optional return. Imagine, for example, you have need for a method that looks at a person's full name, and pulls out and returns that person's middle name. Not all people have a middle name. Thus, your function will need some mechanism to return the person's middle name if they have one, and return `nil` otherwise. Optionals are perfect for these occasions. See below for more.

```
func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortEvenOdd(aBunchOfNumbers)
println("The even numbers are: \(theSortedNumbers.evens);
         the odd numbers are: \(theSortedNumbers.odds)")
func grabMiddleName(name: (String, String?, String)) -> String? {
    return name.1
}

let middleName = grabMiddleName(("Matt",nil,"Mathias"))
if let theName = middleName {
    println(theName)
}
```

In the code above, you created a function called **`grabMiddleName()`**. This function looks a little different when compared to what you've seen before. In this example, your function takes one argument: a tuple of type `(String, String?, String)`. The tuple will contain three `String` instances, one for the first, middle, and last name each. The instance listed for the middle name is declared as an optional type because a person may potentially not have a middle name. Likewise, the return type of the function is also optional for this same reason.

The **`grabMiddleName()`** function's one argument is called `name`. You access this argument inside the implementation of the function using the index of the name that you want to return. Since the tuple is zero-indexed, you use 1 to access the middle name provided to the argument. Everything holds nicely here because that instance will have the appropriate type: `String?`.

You then use the **`grabMiddleName()`** by providing to it a first, middle, and last name (feel free to change the names). Note, since you declared that the middle name component of the tuple to be of type `String?`, you

can pass `nil` to that portion of the tuple. You cannot do this for the first or last name portion of the tuple. Furthermore, consider why nothing is logged to the console: the middle name for the provided person is `nil`. As such, the boolean used in the optional binding does not evaluate to true. In other words, the value of `middleName` is `nil` after the function executes its code.

Try giving the middle name a valid `String` instance and note the result.

Function Types

Each function you have been working with in this chapter has had a specific type. All functions do. Function types are made up of the function's parameter and return types. Consider the **`sortEvenOdd()`** function described above. This function takes an array of integers as an argument, and returns a tuple with two arrays of integers. Thus, the function type for **`sortEvenOdd()`** looks like so: `([Int]) -> ([Int], [Int])`.

The function's parameters are listed on the left inside the parentheses, and the return type comes after the `->`. You can read this function type as: “A function with one parameter that takes an array of integers, and returns a tuple with two arrays containing integers.”. For posterity, it is instructive to see that a function with no arguments and no return has the following type: `() -> ()`.

Function types are particularly useful because you can assign them to variables. This feature will become particularly handy in the next chapter when you see that you can use functions in the arguments and returns of other functions. For now, let's just take a look at how you may assign a function type to a constant.

```
let evenOddFunction: ([Int]) -> ([Int], [Int]) = sortEvenOdd
```

The above example creates a constant named `evenOddFunction` whose value is the function type of the **`sortEvenOdd()`** function. Pretty cool, right? Now you can pass this constant around just like any other. You can even use this constant to call the function: `evenOddFunction([1,2,3])` will sort the numbers in the array supplied to the function's sole argument into a tuple of two arrays - one for even and odd integers.

Conclusion

You accomplished a lot in this chapter. There was a lot of material here, and it may make sense to go through it all a second time. As usual, we suggest that you type out all of the code in this chapter. Indeed, try to extend the examples to different cases; try also to break the examples and then fix them. If you're still a little fuzzy on functions, then don't worry. They're also a major focus in the next chapter.

Challenge

Write a function called **`beanSifter`** that takes a grocery list (as an array of strings) and "sifts out" the beans from the other groceries. The function should take one argument that has an external parameter name called `groceryList`, and it should return a named tuple of the type `(beans: [String], otherGroceries: [String])`. Here is an example of how you should be able to call your function and what the result should be:

```
let result = beanSifter(grocerList: ["green bean",
                                     "milk",
                                     "black bean",
                                     "pinto bean",
                                     "apples"])

result.beans == ["green bean", "black bean", "pinto bean"]
result.otherGroceries == ["milk", "apples"]
```

HINT: You may need to make use of a function on the `String` typed called **`hasSuffix()`**.

13
Closures

It's time to let you in on a secret. You already know what closures are; in fact, you just learned quite a bit about them in the last chapter. You see, functions are really just a special case of closures. That is to say, functions *are* closures. And like functions, closures are discrete bundles of functionality that can be used in your application to accomplish specific tasks. As you'll see in this chapter, you can pass closures into function arguments and even return closures from other functions.

In the previous chapter, you worked primarily with global and nested functions. In both of these cases, the functions you created were named. This is what we mean when we say that functions are a special case of closures: you can think of a function as a named closure.

Closures are also different from functions in that they have a compact and lightweight syntax. As you will see in this chapter, closures allow you to write a "function-like" construct without having to go through the trouble of giving it a name and a full function declaration. This point makes closures easy to pass around in function arguments and returns.

Let's get started. If you haven't already, created a new `Playground` called `Closures.playground` and save it in a good spot.

Closure Syntax

Let's imagine that you are a community organizer managing a number of organizations. You want to keep track of how many volunteers there are for each organization, and have created an instance of the `Array` for this task. Make your code match the below.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]
```

As it happened, you just entered in the number of volunteers for each organization as they were provided to you. This means that the array is completely disorganized. It would be better if your array of volunteers was sorted from least to most. There is good news: Swift provides a function called **`sorted()`** that allows you to specify how an instance `Array` will be sorted.

`sorted()` takes two arguments. The first argument takes the instance of the `Array` type that you'd like sorted, and the second argument takes a closure that describes how the sorting should be done. The closure itself takes two arguments that must be of the same type. These arguments are compared to produce a Bool representing whether or not the instance in first argument should be sorted before the instance in the second argument. Use < if you would like argument one to be sorted before argument two; use > if you would like argument two to come before argument one.

Since your array of volunteers count by organization is filled with integers, the function type for **`sorted()`** will look like this in your code: `([Int], (Int, Int) -> Bool) -> [Int]`. In words, you can read this function type as the following. "Sorted is a function that takes two arguments: 1) an array of Integers, and 2) a closure that takes two integers to compare, and returns a boolean value specifying which integer should come first. Sorted returns a new array of integers that have been ordered according to how the closure was told to organize the." This means that you can pass a function to the second argument of the **`sorted()`** function.

Function types, and their use, should now be a little more clear. Add the following code to sort your array.

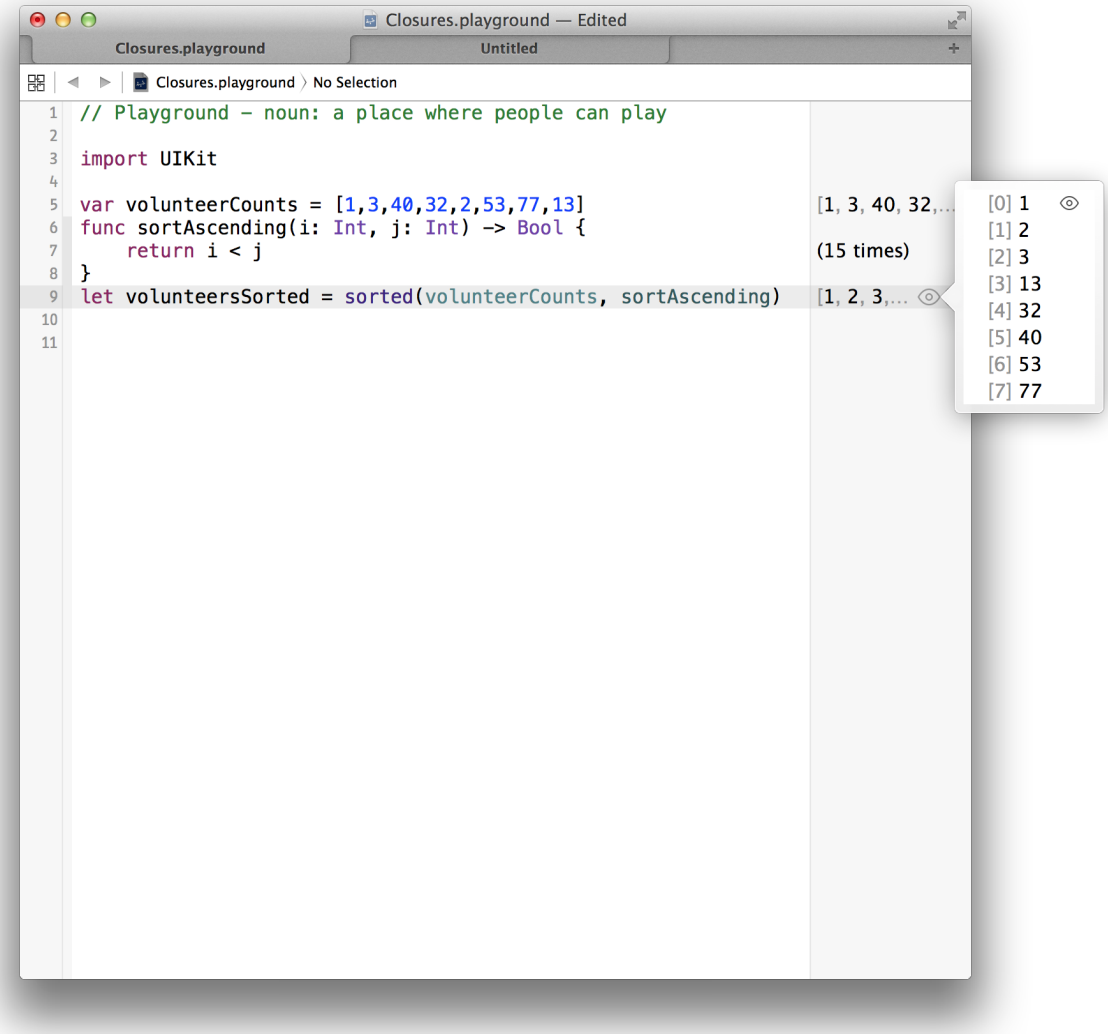
```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

func sortAscending(i: Int, j: Int) -> Bool {
    return i < j
}
let volunteersSorted = sorted(volunteerCounts, sortAscending)
```

First, you created a function called **`sortAscending()`** that has the required type. It compares two integers, and returns a Bool that indicates whether or not `Int i` should be placed before `Int j`. For example, **`sortAscending()`** will return `true` if `i` should be placed before `j`. Remember, since this global function is simply a named closure, you can provide this function as the value of the second argument in the **`sorted()`** function.

Second, you called the **`sorted()`** function, and passed in your array of volunteer counts for the first argument, and the **`sortAscending()`** function for the second argument. Since **`sortAscending()`** returns a new array, you assign that result to a new constant array called `volunteersSorted`. This instance will serve as your new record for the organizations' number of volunteers sorted by number of volunteers.

Look in the results sidebar of your `Playground`. You should see that the values inside `volunteersSorted` are sorted from smallest to largest.

Figure 13.1 Sorting Volunteer Counts



Closure Expression Syntax

Closure syntax follows this general form:

```
{(parameters) -> return type in
     // code
}
```

You write a closure expression inside of the braces ({}). The closure's parameters are listed inside of the parentheses immediatiately after the opening brace. A closure's return type comes after the parameters, and uses the regular syntax. The keyword `in` is used to separate the closure's parameters and return type from the statements inside of its body.

Refactor the example above to use a closure expression. You will create a closure inline instead of defining a separate function outside of the **`sorted()`** function.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

~~func sortAscending(i: Int, j: Int) -> Bool {~~
    ~~return i < j~~
~~}~~

let volunteersSorted = sorted(volunteerCounts, {
    (i: Int, j: Int) -> Bool in
    return i < j
    })
```

This code is a bit cleaner and more elegant than the first pass through. Instead of providing a function defined elsewhere in the `Playground`, you implemented a closure inline in the **`sorted()`** function's second argument. You defined the parameters and their type (`Int`) inside of the closure's parentheses, and also specified its return type. Next, you implemented the closure's body by providing the logical test (i.e., is `i` less than `j`?) that will inform the closure's return. The result is just as before: the sorted array is assigned to `volunteersSorted`.

The example above is great, but it is a little verbose. Closures can take advantage of Swift's type inference system just like you might suspect. Clean up that closure by trimming out the type information.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

~~let volunteersSorted = sorted(volunteerCounts, {~~
    ~~(i: Int, j: Int) -> Bool in~~
    ~~return i < j~~
    ~~})~~

let volunteersSorted = sorted(volunteerCounts, { i, j in i < j })
```

There are three new developments. First, the type information has been removed for both the parameters and the return. Second, the keyword `return` has been removed. This removal is possible because the closure only has one expression (previously, it was this: return `i < j`). If more work needed to be done, then you wouldn't have been able to remove it. Furthermore, removing `return` works because the operation (is `i` less than `j`?) returns a Boolean value. Third, you moved the entire closure expression to be one line. The result is just the same.

Your closure is getting fairly compact, but it can become even more so. Swift provides shorthand argument names that you can refer to in inline closure expressions. These shorthand argument names behave similarly to when you declared the arguments explicitly in the example above: they have the same types and values. The compiler's type inference capabilities helps it to know the number and types of arguments your closure takes, which means it is not necessary to name them.

For example, the compiler knows that the **`sorted()`** takes a closure in the second argument. That closure itself takes two parameters that are of the same type inside of the array you pass into the **`sorted()`** function's first argument. Since the closure at hand has two arguments, the values of which are compared to determine the order

of their values, you can refer to the values of both like so: `$0` for the first, and `$1` for the second. Adjust your code to take advantage of the shorthand syntax.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

~~let volunteersSorted = sorted(volunteerCounts, { i, j in i < j })~~

let volunteersSorted = sorted(volunteerCounts, { $0 < $1 })
```

Since your inline closure expression makes use of the shorthand argument syntax, you don't need to explicitly declare the parameters as you did for `i` and `j`. The compiler knows that the values in the closure's arguments are of the correct type, and knows what to infer based upon the < operator. Incidentally, if your closure has more than two arguments, you can refer to these as: `$2, $3` and so on.

Before you think this closure couldn't possibly get any more slim, just wait, there is more! If a closure is a passed to a function's final argument, it can be written inline outside of and after the function's parentheses. See below.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

~~let volunteersSorted = sorted(volunteerCounts, { $0 < $1 })~~

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }
```

Trailing closure syntax is especially helpful if the closure's body is long. In the code above, a trailing closure simply makes for even less typing, though only marginally. There really is not much to gain here by using a trailing closure.

Truly, “Brevity is the soul of wit.” In the end, the code above works well in this terse form. After all, there is really only one thing that you care about (is one integer less than another?), and that can be easily expressed succinctly. Don't go too crazy with these tricks. It is important to make sure that your code is readable and maintainable.

Functions as Return Types

In Swift, functions are what is called a first class object. What this means is that functions can return other functions as their return type. Recall the town Knowhere that you worked with earlier in the book. It is time to make a function to improve your town. You are going to build some roads.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
        return toLights + lightsToAdd
    }
    return buildRoads
}
```

The function **`makeTownGrand()`** takes no arguments. It's like your grandfather. It does, however, return a function. This function takes two arguments, both integers, and returns an integer. Inside the **`makeTownGrand()`** function's body, you implement the function you return. In terms of implementation details, this function you return is a nested function called **`buildRoads()`**. As you can see, its arguments and return type match what was declared above in **`makeTownGrand()`**.

Exercise your new function and build some roads. Add the following code.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
        return toLights + lightsToAdd
    }
    return buildRoads
}
var stopLights = 4
let townPlan = makeTownGrand()
stopLights = townPlan(4, stopLights)
println("Town has \(stopLights) stop lights.")
```

First, you set up a variable called `stopLights`. This instance is declared as a variable because you are going to build some roads that will add to the town's number of stop lights. Second, you made a constant called `townPlan` that refers to the function that is created by the **`makeTownGrand()`** function. Third, you call that function and pass into it the number of lights to add (the first argument), and the current number of stop lights (the second argument). The result of this function, an instance of type `Int` is reassigned to the `stopLights` variable. Last, you print this new value to the console. Check your console now. It should read, “Town has 8 stop lights.”

Functions as Arguments

Functions can serve as arguments to other functions. Recall, for example, that you initially gave **`sorted()`** the **`sortAscending()`** function as an argument early on in this chapter.

Imagine that your town cannot just arbitrarily build roads. Practicality suggests that your town can only build roads when it has a suitable budget. Adjust your previous **`makeTownGrand()`** function to take a budget parameter and a condition parameter. The budget parameter will serve as your town's budget, and the condition parameter will evaluate whether or not this budget is suitable to build the new roads.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

~~func makeTownGrand() -> (Int, Int) -> Int {~~
~~    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {~~
~~        return toLights + lightsToAdd~~
~~    }~~
~~    return buildRoads~~
~~}~~

~~let townPlan = makeTownGrand()~~
~~stopLights = townPlan(4, stopLights)~~

func makeTownGrand(budget: Int, condition: Int -> Bool) -> ((Int, Int) -> Int)? {
    if condition(budget) {
        func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
            return toLights + lightsToAdd
        }
        return buildRoads
    } else {
        return nil
    }
}
func evaluateBudget(budget: Int) -> Bool {
    return budget > 10000
}
var stopLights = 4
if let townPlan = makeTownGrand(1000, evaluateBudget) {
    stopLights = townPlan(4, stopLights)
}
println("Town has \(stopLights) stop lights.")
```

There are three important changes.

First, the new **makeTownGrand()** function takes two arguments. The first is an instance of the `Int` representing the town's budget. The second is called `condition` and takes a function. This function will be used to determine if the town's budget is sufficient. Thus, it will take an integer and return a Boolean. If the integer budget is high enough, then this function will return `true`. If the budget is not high enough, then the function will return `false`.

Notice that the **makeTownGrand()** function has a different return type? The return type is now `((Int, Int) -> Int)?`. The previous implementation of the **makeTownGrand()** returned a function that took two integers and returned an integer. In this revised version, **makeTownGrand()** returns the same thing, but in an optional incarnation. The reason for this is plain: if the town has the appropriate budget, then the **buildRoads()** will be created and returned. If, on the other hand, the budget is not sufficient, then the **buildRoads()** will not be created and `nil` will be returned.

The implementation of **makeTownGrand()** runs the function passed into the `condition` parameter. If it evaluates to `true` then the **buildRoads()** is created and returned. If `condition` evaluates to `false`, then `nil` is returned.

Second, you create the aptly named **evaluateBudget()** function. This function takes an integer and returns a Boolean. Its implementation simply evaluates the integer to see if it is greater than an arbitrary threshold: 10000 units of some currency.

Third, you used optional binding to conditionally set `townPlan`. If the budget provided to the **makeTownGrand()** function is sufficiently large, then the **buildRoads()** will be created, returned, and assigned to `townPlan`. In this case, your town's number of `stopLights` will be increased by 4. If, however, the budget is not large enough, then **makeTownGrand()** will return `nil`. In this case, your town's number of stop lights will not be increased.

Check your console. Unfortunately, your town's budget doesn't appear to be large enough. A budget of 1000 is certainly smaller than the requisite 10000. Thus, **makeTownGrand()** returned `nil` and **buildRoads()** was never executed. And so, your town will not be able to build any new roads.

Closures Capture Values

Closures and functions can keep track of internal information encapsulated by a variable defined in its enclosing scope. For example, imagine that your town, Knowhere, is booming. Since growth can be erratic, you have decided to create a function that will allow you to update the town's population data depending upon recent growth. Your town planner has decided that it makes the most sense to update the town's census data every time the population grows by 500 people.

```
println("Town has \(stopLights) stop lights.")

func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {
    var totalGrowth = 0
    func growthTracker() -> Int {
        totalGrowth += growth
        return totalGrowth
    }
    return growthTracker
}
var currentPopulation = 5422
let growBy500 = makeGrowthTracker(forGrowth: 500)
```

The function **makeGrowthTracker()** is a function that builds a **growthTracker()** function. **makeGrowthTracker()** takes one argument, an integer representing the growth to track, and returns a function that takes no arguments and returns an integer. This integer is a running total of the growth your town is experiencing. The **growthTracker()** function captures the value of the `totalGrowth` variable from its enclosing scope. After **growthTracker()** is created, the `totalGrowth` variable will be incremented by the amount specified in the argument passed to the **makeGrowthTracker()** function. Exercise and test this function by calling it a few times.

```
println("Town has \(stopLights) stop lights.")

func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {
    var totalGrowth = 0
    func growthTracker() -> Int {
        totalGrowth += growth
        return totalGrowth
    }
    return growthTracker
}
var currentPopulation = 5422
let growBy500 = makeGrowthTracker(forGrowth: 500)
growBy500()
growBy500()
growBy500()
currentPopulation += growBy500() // currentPopulation is now 7422
```

As you can see, **`growBy500()`** is called four times. This translates to a growth of 2000 people for your town. Notice that the first three calls to **`growBy500()`** does not assign its result to any constant or variable. This is fine because the function is keeping an internal running total of your town's growth. Thus, all you have to do to update your town's population is assign the result of the function to your `currentPopulation` variable when your town planner is ready.

Closures are Reference Types

Closures are reference types. When you assign a function to a constant or variable you are actually setting that constant or variable to point to the function. You are not creating a distinct copy of that function. One important consequence of this fact is that any information captured by the function's scope will be changed if you call the function via your new constant or variable. For example, imagine that you created a new constant and set it equal to your **`growBy500()`** function.

```
func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {
    var totalGrowth = 0
    func growthTracker() -> Int {
        totalGrowth += growth
        return totalGrowth
    }
    return growthTracker
}
var currentPopulation = 5422
let growBy500 = makeGrowthTracker(forGrowth: 500)
growBy500()
growBy500()
growBy500()
currentPopulation += growBy500() // currentPopulation is now 7422
let anotherGrowBy500 = growBy500
anotherGrowBy500() // totalGrowth now equal to 2500
```

`anotherGrowBy500` now points to the same function that `growBy500` points to. The significance of this is that if you call the function by typing `anotherGrowBy500()`, then the variable `totalGrowth` will be incremented by 500. Thus, Knowhere's total population growth is increased to become 2500 in the code above.

In disctinction, consider that a neighboring city has fallen in love with your town planner's function. This city is quite a bit larger, and its mayor feels confident that using your town's growth function will help to reduce costs. Accordingly, the city will need its own growth tracker function because its growth is quite a bit larger than your small town's growth. Create another town's population, and use the **`makeGrowthTracker()`** function to create another growth tracker function for the larger city. This new growth function will increase the city's population by 10,000.

```
let anotherGrowBy500 = growBy500
anotherGrowBy500() // totalGrowth now equal to 2500
var someOtherPopulation = 4061981
let growBy10000 = makeGrowthTracker(forGrowth: 10000)
someOtherPopulation += growBy10000()
currentPopulation
```

You now have another population that you are keeping track of, and have a new growth tracker function called **`growBy10000()`** to help you do it. As above, you use the **`growBy10000()`** to grow the city's population:

`someOtherPopulation += growBy10000()`. The city's population is increased to 4,071,981 after this line. Notice that your town's population doesn't change; it is still 7,422. This is because you used the **`makeGrowthTracker()`** function to create a new growth tracker function. This new growth tracker function is separate and distinct from **`growBy500()`**.

Conclusion

Closures and functions bear a strong relationship to each other: functions are named closures. Both are powerful constructs, being able to serve as arguments and return types for other functions. You can use functions and closures to build other functions or closures. Additionally, closures are extremely helpful in streamlining your code to make it more elegant and readable.

Make sure to get comfortable with using these tools. In particular, make sure to understand the grammar of functions and closures. This skill will help you read other people's code faster and will serve you well in writing your own. To be sure, closures and functions are central to working effectively with Swift.

Challenge

Swift adopts some patterns from functional programming. For example, Swift provides several higher-order functions that are well-known to programmers fond of functional programming: **`filter`**, **`map`**, and **`reduce`**. These higher-order functions take at least one function as an input. This quality is what makes them higher-order functions in the definitional sense. In fact, you have already worked with higher-order functions in this chapter.

Let's say that somebody asked you how many volunteers you are keeping track of. How would you figure this out? Would you go and look inside your `Playground` and simply add up the numbers at each index in the array? Ew! You are a programmer! Write some code. Or, maybe you would write a for-in loop to do the counting for you. Yuck! That is so much typing for such a simple task. It turns out that **`reduce`** provides an even better way. Take a look.

```
let totalVolunteers = volunteerCounts.reduce(0, combine: {
    (i: Int, j: Int) -> Int in
    return i + j
    }
)
```

You can use **`reduce`** on collection types such as arrays. Its job is to *reduce* the values in the collection to a single value that is returned from the function. The first argument refers to an initial amount (or some other value) that can be added at the outset. The second argument is a closure that defines how the values inside the collection should be combined. In the scenario presented above, all you want to accomplish is to add up the integers at each index of the array. When the function is done, `totalVolunteers` is set to be equal to 221.

Now, here is the challenge: Use what you have learned in this chapter and clean up the implementation of **`reduce()`** presented above. In fact, the implementation can be shortened quite significantly. Your solution should be expressed in one line. When you are done, go ahead and take a look at the other higher-order functions and practice with them.

Part IV

Enumerations, Structures, and Classes

14
Enumerations

As you have worked through the book up to this point, you have been using all the built-in types that Swift provides, like `Int`, `String`, arrays, and dictionaries. The next couple of chapters will show the capabilities the language provides to create your own types. The focus of this chapter is enumerations, which allow you to create instances that are allowed to be one of a predefined list of cases. If you have used enumerations in other languages, much of this chapter will be familiar, but Swift's enums also have some advanced features that make them unique.

Basic Enumerations

Create a new `Playground` called `Enumerations.playground`. Type the following code into the playground to define an enumeration of possible text alignments:

```
enum TextAlignment {
    case Left
    case Right
    case Center
}
```

You define an enumeration with the `enum` keyword followed by the name of the enumeration. The opening brace ({) opens the body of the enum, and it must contain at least one `case` statement that declares the possible values for the enum. The name of the enumeration (`TextAlignment` in this case) is now usable as a type, just like `Int` or `String` or the various other types you have used so far. Types are written with a capital first letter by convention. If multiple words are needed, then use camel-casing; for example, `UpperCamelCasedType`. Variables and functions begin with a lowercase first letter, and also use camel-casing as needed.

Because the enumeration declares a new type, you can create instances of that type:

```
enum TextAlignment {
    case Left
    case Right
    case Center
}

var alignment: TextAlignment = TextAlignment.Left
```

The compiler can still infer the type for `alignment` even though `TextAlignment` is a type that you have defined. Therefore, you can omit the explicit type of the `alignment` variable:

```
~~var alignment: TextAlignment = TextAlignment.Left~~
var alignment = TextAlignment.Left
```

The compiler's ability to do type inference on enumerations isn't limited to just variable declarations. If you have a variable known to be of a particular enum type, you can omit the type from the case statement when assigning a new value to the variable:

```
var alignment = TextAlignment.Left
alignment = .Right
```

Notice that you had to specify the enum's type and value when initially creating the `alignment` variable, because that line gives `alignment` both its type and its value. In the next line, you can omit the type and simply reassign `alignment` to be equal to a different value within its type. You can also omit the enum type when passing its values to functions or comparing them:

```
alignment = .Right

if alignment == .Right {
    println("we should right-align the text!")
}
```

While enum values can be compared in `if` statements, you typically use a `switch` statement to handle enum values. Enter the following code which prints the alignment in a human-readable way:

```
var alignment = TextAlignment.Left
alignment = .Right

~~if alignment == .Right {~~
~~    println("we should right-align the text!")~~
~~}~~
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

case .Center:
    println("center aligned")
}
```

Recall from Chapter 6 that all switch statements must be exhaustive. In that chapter, every switch statement you wrote included a `default` clause. When switching on enumeration values, that isn't necessary: the compiler knows all possible values the enumeration can check, and if you have included a case for each one, the switch is exhaustive. You could include a `default` clause when switching on an enum type:

```
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

~~case .Center:~~
default:
    println("center aligned")
}
```

This code works the same way, but we recommend avoiding `default` clauses when switching on enum types. It isn't as "future proof". Suppose you have written this code, and later you come back and realize you want to add another alignment option for justified text:

```
enum TextAlignment {
    case Left
    case Right
    case Center
    case Justify
}

~~var alignment = TextAlignment.Left~~
~~alignment = .Right~~
var alignment = TextAlignment.Justify
```

Notice that your program still runs, but it now prints the wrong value! The `alignment` variable is set to `Justify`, but the switch statement prints "center aligned". Go back to the switch statement and change it back to listing each case explicitly instead of using `default`:

```
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

~~default:~~
case .Center:
    println("center aligned")
}
```

Now, instead of your program running and printing the wrong answer, you have a compile-time error that your switch statement is not exhaustive. It may seem odd to suggest that a compiler error is desirable, but that's exactly the situation here. If you use a `default` clause when switching on an enum and then later add a new case to the enum, the switch statement will fall to the `default` when it encounters the new case. Listing each case in the switch means that the compiler can help you to find all of the places in your code that need to be updated if you later add an additional case to your enum.

```
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

case .Center:
    println("center aligned")

case .Justify:
    println("justified")
}
```

Raw Value Enumerations

If you have used enumerations in a language like C or C++, you may be surprised to learn that unlike in those languages, Swift enums do not have an underlying integer type. You can, however, choose to get the same behavior by using what Swift refers to as a "Raw Value". To use `Int` raw values for your text alignment enumeration, change the declaration of the enum:

```
~~enum TextAlignment {~~
enum TextAlignment : Int {
    case Left
    case Right
    case Center
    case Justify
}
```

Specifying a raw value type for `TextAlignment` gives a distinct raw value of that type (`Int`) to each case. The default behavior for integral raw values is that the first case gets raw value 0, the next case gets raw value 1, and so on, which you can confirm:

```
var alignment = TextAlignment.Justify

println("Left has raw value \(TextAlignment.Left.rawValue)")
println("Right has raw value \(TextAlignment.Right.rawValue)")
println("Center has raw value \(TextAlignment.Center.rawValue)")
println("Justify has raw value \(TextAlignment.Justify.rawValue)")
println("The alignment variable has raw value \(alignment.rawValue)")
```

Alternatively, you can specify the raw value for each case manually:

```
enum TextAlignment : Int {
    ~~case Left~~
    ~~case Right~~
    ~~case Center~~
    ~~case Justify~~
    case Left    = 20
    case Right   = 30
    case Center  = 40
    case Justify = 50
}
```

When is a raw value enumeration useful? The most common reason for using a raw value is that you need to store or transmit the enum. Instead of having to write functions to transform a variable holding an enum, you

can use `rawValue` to convert the variable to its raw value. This brings up another question: if you have a raw value, how do you convert it back to the enum type? Every enum type with a raw value can be created with a `rawValue:` argument, which returns an Optional (see Chapter 9) enum:

```
println("Justify has raw value \(TextAlignment.Justify.rawValue)")
println("The alignment variable has raw value \(alignment.rawValue)")

// Create a raw value.
let myRawValue = 20

// Try to convert the raw value into a TextAlignment
if let myAlignment = TextAlignment(rawValue: myRawValue) {
    // Conversion succeeded!
    println("successfully converted \(myRawValue) into a TextAlignment")
} else {
    // Conversion failed.
    println("\(myRawValue) has no corresponding TextAlignment case")
}
```

What's going on here? You start with `myRawValue` that is a variable of type `Int`. Then, you try to convert that raw value into a `TextAlignment` case using `TextAlignment(rawValue:)`. Because `TextAlignment(rawValue:)` has a return type of `TextAlignment?`, you are using Optional Binding to determine whether you get a `TextAlignment` value or `nil` back. The raw value you used here happens to correspond to `TextAlignment.Left`, so the conversion succeeds. Try changing it to a raw value that doesn't exist, and note that you get a message that conversion isn't possible:

```
~~let myRawValue = 20~~
let myRawValue = 100
```

So far, you have been using `Int` as the type for your raw values. Swift allows a variety of types to be used, including all the built-in numeric types and `String`. Create a new enum that uses `String` as its raw value type:

```
enum ProgrammingLanguage : String {
    case Swift      = "Swift"
    case ObjectiveC = "Objective-C"
    case C          = "C"
    case Cpp        = "C++"
    case Java       = "Java"
}

let myFavoriteLanguage = ProgrammingLanguage.Swift
println("My favorite programming language is \(myFavoriteLanguage.rawValue)")
```

You didn't have to specify values when you first used a raw value of type `Int` - the compiler automatically set the first case to 0, the second case to 1, and so on. This is not true for `String` (or any other raw value type that isn't integral): you must always provide distinct raw values for all cases.

Methods

A method is a function that is associated with a type. In some languages, methods can only be associated with classes (which we'll discuss in Chapter 15). In Swift, methods can also be associated with enums. Create a new enum that represents the state of a light bulb:

```
enum LightBulb {
    case On
    case Off
}
```

One of the things you might want to know is the temperature of the light bulb. (For simplicity, assume that the bulb heats up immediately when it's turned on and cools off to the ambient temperature in the room immediately when it turns off.) Add a method for computing the surface temperature:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
        case .On:
            return ambient + 150.0

        case .Off:
            return ambient
        }
    }
}
```

You added a function inside the definition of the `LightBulb` enumeration. Because of the location of the definition of this function, it is now a method associated with the `LightBulb` type, or more commonly a "method on `LightBulb`". The function appears to take a single argument (`ambient`); however, because it is a method, it also takes an implicit argument named `self` of type `LightBulb`. Create a variable to represent a light bulb and call your new method:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
        case .On:
            return ambient + 150.0

        case .Off:
            return ambient
        }
    }
}

var bulb = LightBulb.On
let ambientTemperature = 77.0

var bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")
```

First you create `bulb`, an instance of the `LightBulb` type. When you have an instance of the type, you can call methods on that instance using the syntax `instance.methodName(arguments)`. Another method that seems like it might be useful is a method to toggle the light bulb. To toggle the light bulb, you need to modify `self` (to change it from On to Off or Off to On). Try to add the `toggle` method, which takes no arguments and doesn't return anything:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
        case .On:
            return ambient + 150.0

        case .Off:
            return ambient
        }
    }

    func toggle() {
        switch self {
        case .On:
            self = .Off

        case .Off:
            self = .On
        }
    }
}
```

After typing this in, you will get a compiler error that states that you cannot assign to `self` inside a method. In Swift, an enumeration is a value type, and methods on value types are not allowed to make changes to `self` (there will be more discussion of value types in Chapter 15). If you want to allow a method to change `self`, you need to mark it as a `mutating` method:

```
    ~~func toggle() {~~
    mutating func toggle() {
        switch self {
        case .On:
            self = .Off

        case .Off:
            self = .On
        }
    }
```

Now you can toggle your light bulb and see what the temperature is when the bulb is off:

```
var bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")

bulb.toggle()
bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")
```

Associated Values

Everything you have done so far with enumerations falls into the same general category of defining static cases that enumerate possible values or states. Swift also sports a much more powerful flavor of enumeration: cases with associated values. Associated values allow you to attach data (of any type you specify) with a case, and different cases in an enumeration can have different types of associated values.

Create an enumeration that allows for tracking the dimensions of a couple of basic shapes. Each kind of shape has different types of properties: to represent a square, you need a single value (the length of one side); to represent a rectangle, you need a width and a height.

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its with and height
    case Rectangle(width: Double, height: Double)
}
```

You have defined a new enumeration type, `ShapeDimensions`, with two cases. The `Square` case has an associated value of type `Double`. The `Rectangle` case has an associated value with type `(width:Double, height:Double)`, a named tuple (first seen in Chapter 12). To create instances of `ShapeDimensions`, you must specify both the case and an appropriate associated value for the case:

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its with and height
    case Rectangle(width: Double, height: Double)
}

var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
```

You can use a switch statement to unpack an associated value and make use of it. Add a method to `ShapeDimensions` that computes area:

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its with and height
    case Rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
        case let .Square(side):
            return side * side

        case let .Rectangle(width: w, height: h):
            return w * h
        }
    }
}
```

In the implementation of `area()`, you switch on `self` just like you did earlier in this chapter. The difference is that the switch's cases use Swift's *pattern matching* to bind `self`'s associated value with a new variable (or variables). Call the `area()` method on the instances you created earlier to see it in action:

```
var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
```

Not all enum cases have to have associated values. For example, you could add a `Point` case, but geometric points don't have any dimensions! That's fine; just leave off the associated value type. Also update the `area()` method to include the area of a `Point`:

```
enum ShapeDimensions {
    // Point has no associated value – it is dimensionless
    case Point

    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its with and height
    case Rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
        case let .Point:
            return 0

        case let .Square(side):
            return side * side

        case let .Rectangle(width: w, height: h):
            return w * h
        }
    }
}
```

Create an instance of a point and confirm that it works:

```
var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
var pointShape = ShapeDimensions.Point

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
println("point's area = \(pointShape.area())")
```

assert() and precondition()

There might be a small problem with this implementation. What happens if some yahoo (possibly you, in a moment of weakness) creates a shape with a negative dimension? Try it and see:

```
var squareShape = ShapeDimensions.Square(10.0)
~~var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)~~
var rectShape = ShapeDimensions.Rectangle(width: -5.0, height: 10.0)
var pointShape = ShapeDimensions.Point

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
println("point's area = \(pointShape.area())")
```

Now `area()` is returning a negative number, which definitely doesn't make sense! How can you solve this problem? The simplest solution is to document a warning in a comment: "Don't create a shape with a NEGAIVE dimension," but that doesn't help if someone isn't paying attention. A more complicated solution might be to try to handle negative dimensions correctly, perhaps by assuming they should've been positive and taking their absolute values, but that's problematic too: since a shape with a negative dimension shouldn't exist, it probably represents a bug in the code.

Software failures can be categorized many different ways, but one of the most general is to split them into one of two categories: programmer errors and non-programmer errors. A programmer error is a bug in the code, an example being creating a rectangle with a negative width. A non-programmer error is a problem that isn't caused by a bug in the code but still has the potential to cause the program to behave incorrectly. An example of a non-programmer error is trying to fetch a web site from a device that doesn't have an internet connection (such as from a phone during a flight) - the program will fail to fetch the web site, but not due to any programmer fault.

Solutions to these two categories of failures are very different. Non-programmer errors need to be handled in the program; *not* handling a non-programmer error is often a programmer error (e.g., if an app that tries to fetch a web site crashes when no internet connection is available, that's definitely a bug)! Programmer errors generally cannot be handled, because you don't even know they exist or you would've fixed them. The best you can do is try to catch these and immediately stop the program before it continues on in some unknown state.

Swift provides two functions that can be used to catch programmer errors: `assert()` and `precondition()`. When a program is built in Debug mode (the default mode while running in Xcode), the two functions do the same thing: check a condition that you expect to be true, and if it's false, *trap*, which immediately drops you into the debugger. When a program is built in Release mode (e.g., when you submit a Mac or iOS app to the App Store), `assert()` calls are removed, but `precondition()` calls remain, and the app will halt if a precondition fails.

When should you use `assert()` versus `precondition()`? A general rule of thumb is to use `assert()` to check for conditions that will only be false if something "local" has gone wrong. The definition of "local" is a little nebulous and requires some judgment. You might take local to mean "any code you have control over", but a more common definition would be "within the same type". For example, if another method of `ShapeDimensions` called `area()`, it would make sense to use `assert()` to confirm that `area()` returned a non-negative value. `precondition()`, on the other hand, should be used to check for errors caused outside the scope of the check; for example, Swift's arrays use `precondition()` (or something much like it) to check that you don't try to read past the end of the array.

In the case of `ShapeDimensions`, it makes sense to use `precondition` to check that a shape has non-negative dimensions, since an error of that kind would be caused by code using `ShapeDimensions`, not code within `ShapeDimensions` itself. Do that now:

```
    func area() -> Double {
        switch self {
        case let .Square(side):
            precondition(side >= 0, "side cannot be negative")
            return side * side

        case let .Rectangle(width: w, height: h):
            precondition(w >= 0, "width cannot be negative")
            precondition(h >= 0, "height cannot be negative")
            return w * h
        }
    }
```

After adding the preconditions, open up the Timeline, and you should see: `precondition failed: width cannot be negative`. Now that `ShapeDimensions` traps programmer errors, go back and fix the root cause:

```
~~var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)~~
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
```

Conclusion

You are now familiar with Swift enumerations and its variety of options:

- Plain enums for listing the cases of a group of related things.
- Raw value enums for groups of related things where each case has a constant raw value, and every case has the same type of raw value.
- Associated value enums, a powerful feature of Swift that allows you to attach data to instances of an enumeration.

In addition, you have learned to create methods, which are functions associated with a type. Methods will feature prominently in the next chapter as well. Finally, you know how to use `assert()` and `precondition()` to guard against programmer errors.

Challenge

Add another case to the `ShapeDimensions` enum for a right triangle. You can ignore the orientation of the triangle - just keep track of the lengths of its sides. Are there any preconditions that make sense for this shape beyond checking that the sides aren't negative? (Hint: think back to high school geometry and the Pythagorean theorem.)

Add a `perimeter()` method to the `ShapeDimensions` enum. This method should compute the perimeter of a shape (the sum of the length of all its edges). Make sure you handle all the cases!

For the More Curious: Error Handling in Cocoa

A common idiom for writing functions that might fail with non-programmer errors in Objective-C on the Mac and iOS platforms is to return an optional (which is nil if the function failed) and accepts an extra parameter of type `NSError` that gets set to details of the error. In Swift, here's an example of what this might look like for a function that tries to fetch a web page. Please note that this code is valid Swift syntactically but will not run in its current state - it assumes the existence of other functions and types that are not shown here.

```
var error: NSError? = nil
let url = "http://www.bignerdranch.com"

let webSiteContents: Contents? = fetchWebSite(url, error: &error)

if let contents = webSiteContents {
    println("successfully fetched \(contents)")
} else {
    println("fetching \(url) failed with error: \(error)")
}
```

This somewhat odd pattern is due to Objective-C's inability to return multiple values. Since Swift supports returning multiple values via tuples, you might think this idiom would tranlsate to `fetchWebSite` returning a tuple of `(Contents?, NSError?)` instead, like so:

```
let url = "http://www.bignerdranch.com"

let (webSiteContents: Contents?, error: NSError?) = fetchWebSite(url)

if let contents = webSiteContents {
    println("successfully fetched \(contents)")
} else {
    println("fetching \(url) failed with error: \(error!)")
}
```

That's a definite improvement, but it still has problems. The return type of `fetchWebSite` doesn't clearly express the intention of the function; for example, the function might return both a `Contents` and an `NSError`, or even

`(nil, nil)`! Instead, Swift enumerations with associated values can be used to make the meaning clear: the function will return either a `Contents` or an `NSError`, but never both or neither:

```
enum FetchResult {
    case Success(Contents)
    case Failure(NSError)
}

let url = "http://www.bignerdranch.com"

let result: FetchResult = fetchWebSite(url)

switch result {
case let .Success(contents):
    println("successfully fetched \(contents)")

case let .Failure(error):
    println("fetching \(url) failed with error: \(error)")
}
```

This results in marginally longer code, but is ultimately more readable because the possible results of `fetchWebSite` are clearly listed in its return type. It is now impossible for `fetchWebSite` to return an ambiguous value: you are guarnteed to receive either the contents or an error. This guarantee is checked by the compiler, and it is obvious to both the caller (the code here) and the implementer of `fetchWebSite` itself what the possible return values are.

15
Structs and Classes

Structures, commonly referred to as structs, and classes are the pillars on which you build your applications. They provide an important mechanism to model the things you wish to represent in your code.

In this chapter, you will transition from the `Playground` and create a Command Line Tool. The Command Line Tool will describe a project that represents a town undergoing a serious monster infestation. You will use both structs and classes to model these entities, and will give them properties to store data, and functions such that these entities can do some work. Along the way, you will see the differences and similarites between classes and structs, which will help you to decide when to use each.

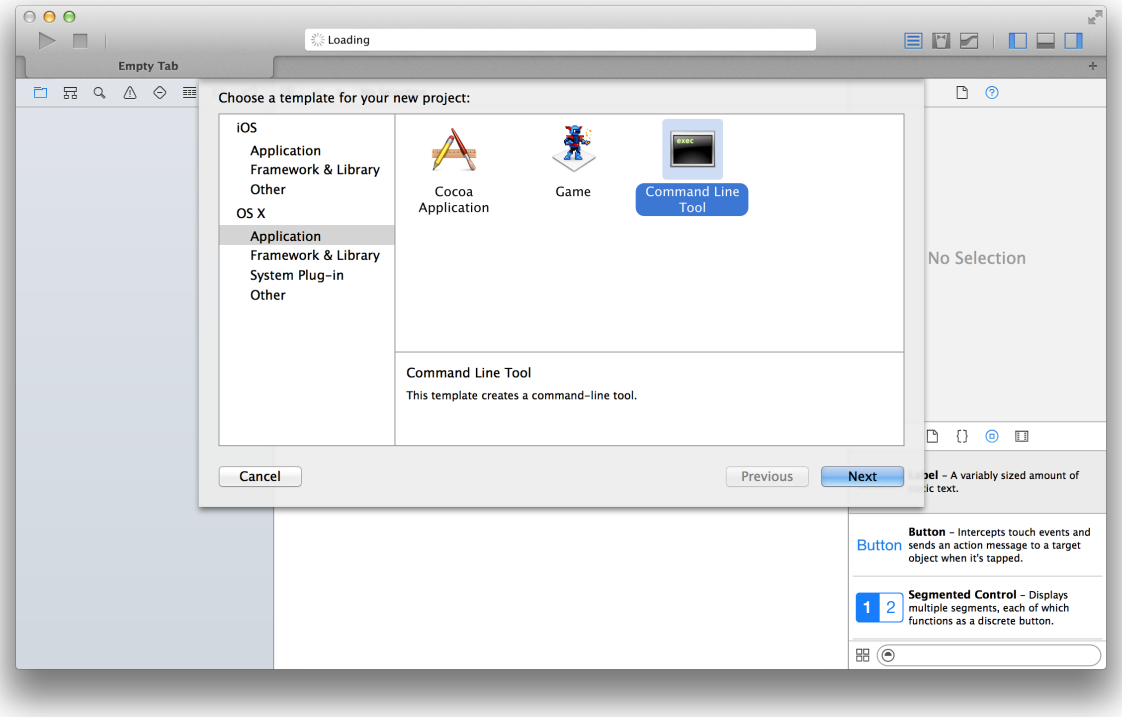
A New Project

Create a new project by clicking on your Xcode icon. The first screen that you see is the Welcome screen. See Figure 15.1 for details. Click the button that reads, "Create a new Xcode Project".

Figure 15.1 Welcome Window

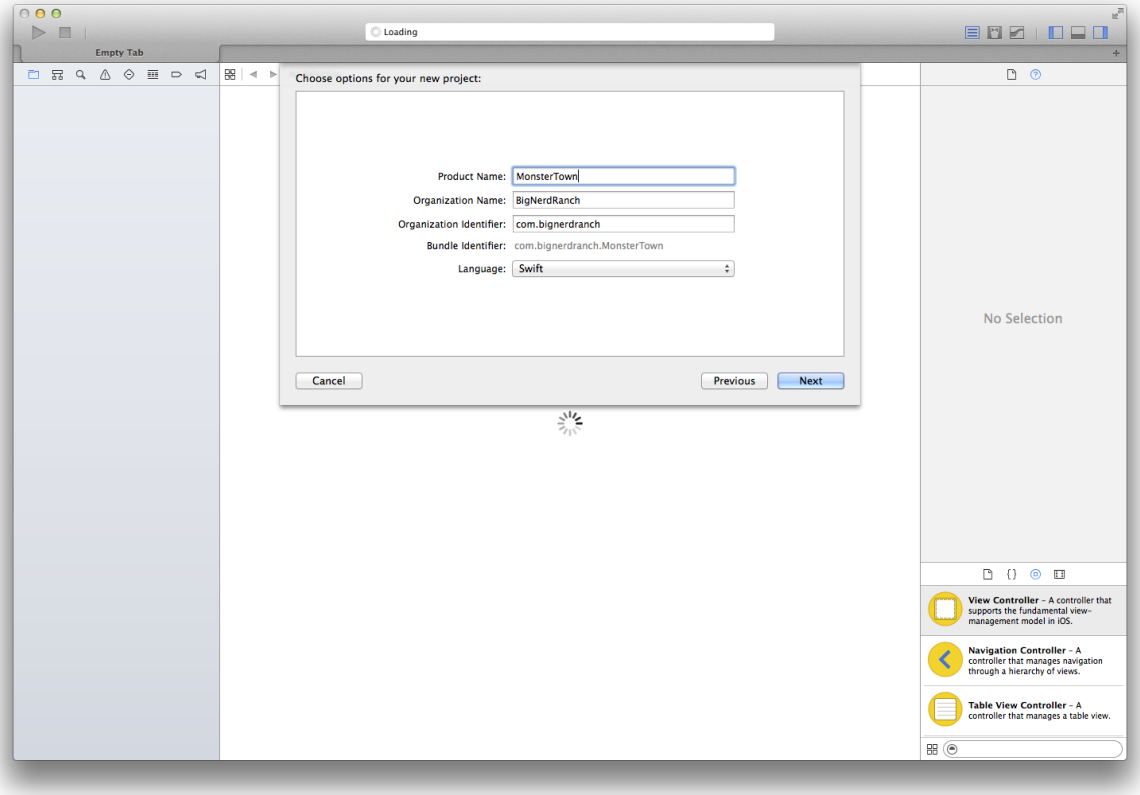
Next, you will see a screen for selecting a project template. A template formats your project with a number of presets and confurations common to a given style of application. On the left hand side of the window, you should see two sections: 1) iOS, and 2) OS X. Select "Application" from within the OS X section, and you should a window very similar to Figure 15.2. Choose the "Command Line Tool" template option. This template will create a very basic project file for you.

Figure 15.2 Choosing a New Project



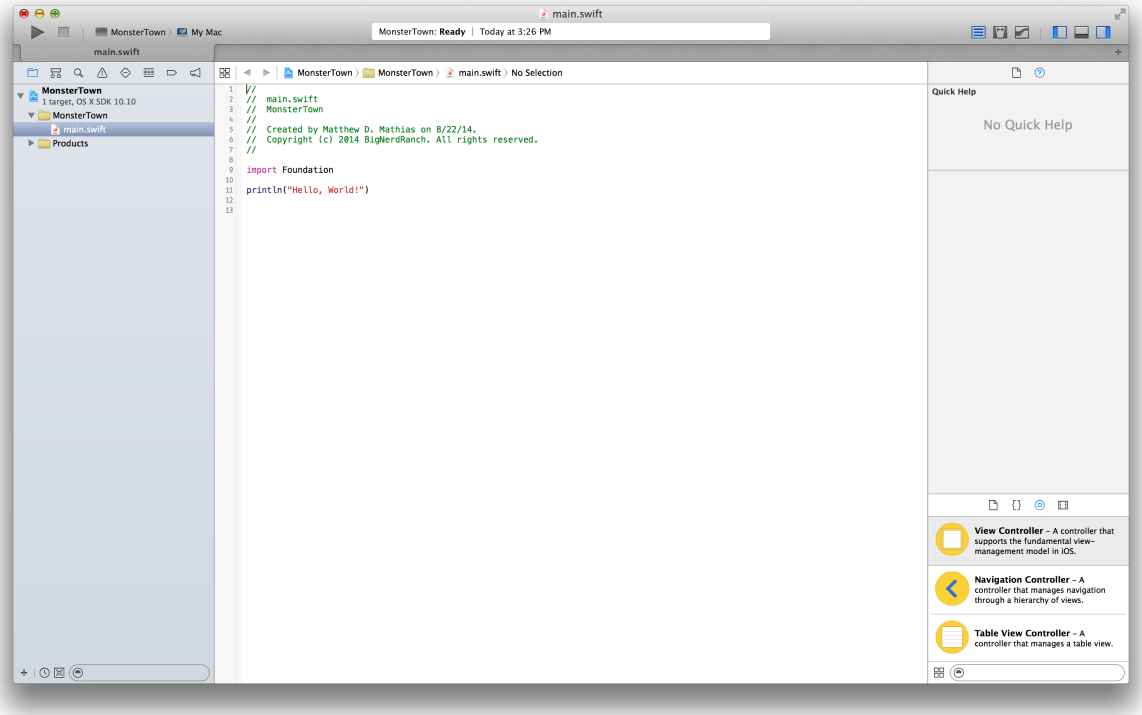
Now you need to name your project. You should see a window similar to Figure 15.3. In the "Product Name" field, type in "MonsterTown". You are welcome to type whatever you like for the project's "Organization Name". We use BigNerdRanch. The "Organization Identifier" typically uses "Reverse Domain Name Service" notation (reverse DNS), and is used with the "Product Name" to create a "Bundle Identifier". The bundle ID is used to identify your application on iTunes Connect when you are ready to distribute your application. The last time to accomplish on this window is to select Swift for the "Language" option.

Figure 15.3 Naming Your Project



Last, Xcode will ask you where to save the project. Select a good location for you, and click "Create". Now that you have created your project, you should see it open in Xcode with the `main.swift` file selected. See Figure 15.4.

Figure 15.4 main.swift



Notice that the `main.swift` file already has the following code:

```
import Foundation

println("Hello, World!")
```

The `import Foundation` code brings the `Foundation` framework into the `main.swift` file. This framework consists of a number of classes primarily designed to do work in and with Objective-C. Next, the `println("Hello, World!")` code should be fairly familiar. It logs the string `"Hello, World!"` to the console. Go ahead and run your program. You can do this several ways: 1) click "Product" from the toolbar above and then select "Run", 2) tap command-R on your keyboard, or 3) click the play button in the upper lefthand corner in Xcode.

After you run your program, you should see that `"Hello, World!"` is logged to the console. That is great, but you have seen this behavior before. Let's make your program more interesting. First, remove the code listed in `main.swift`. You won't be needing this code.

```
~~import Foundation~~

~~println("Hello, World!")~~
```

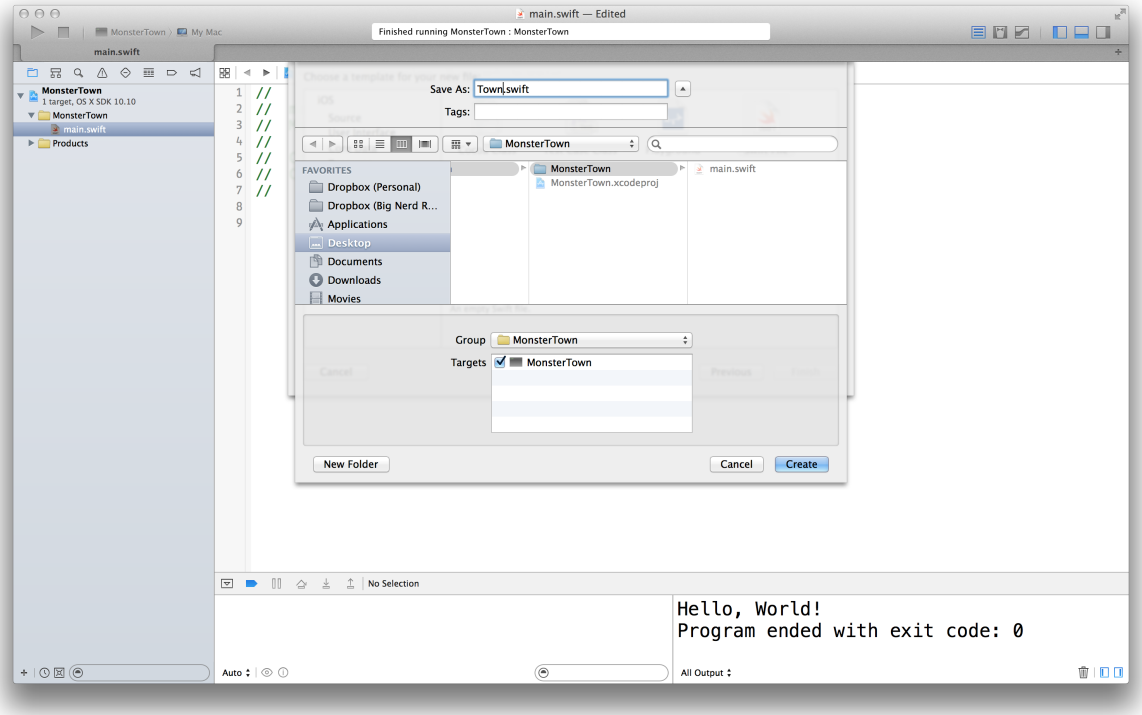
Next, you will move onto making your own structs and classes .

Structures

A struct is a type that groups a set of related chunks of data together in memory. They are used when a developer would like to group data together under a common type. There are other important considerations to make when choosing between a class and a struct, and these will visited throughout this chapter.

Add a new file to your project. Click on File > New > File... You could otherwise type command-N from your keyboard. This will bring up a new window prompting you to select a template for your new file. Select "Source" in the OS X section on the left, and then choose "Swift File" and click "Next". Next, you'll be asked to name the new file and set its location. Call this file `Town.swift` and make sure to check the box to add it to the MonsterTown target. See Figure 15.5 for reference.

Figure 15.5 main.swift



On the left hand side of the Xcode application window, you should see the Project Navigator. In the Project Navigator, you should see a listing of your files: 1) `main.swift`, and 2) `Town.swift`. Select the file `Town.swift`. When this file appears, you should notice that it is completely blank (except for the commented out code at the top of the file). Add the following to create a declaration for your Town struct.

```
struct Town {

}
```

The keyword `struct` signals that you are declaring a struct named "Town". You will add code between the braces ({}) to define the behavior of this struct. For example, you can add variables to your new struct so that it can hold onto some data.

Technically, these variables are called properties, which is the subject of the next chapter. Properties can be variable or constant, as you have seen before using the `var` and `let` keywords. Add some properties to your struct.

```
struct Town {
     var population = 5422
     var numberOfStopLights = 4
}
```

If you recall from previous chapters, you were attempting to model a town in a `Playground`. Since the example was relatively small, this was not all that limiting. Nonetheless, it is better to encapsulate the definition of the town within its own type. In this case, you are now defining a new struct called `Town`. `Town` has two properties: 1) `population`, and 2) `numberOfStopLights`. Both of these properties are mutable - a town's population and number of stop lights may change. These properties are also default values for the sake of simplicity. When a new instance of the `Town` struct is made, it will default to having a `population` of 5422, and 4 stop lights.

Create a new instance of `Town` to see your struct in action. Switch to your `main.swift` file, and add this code:

```
var myTown = Town()
println("Popuation: \(myTown.population),
         number of stop lights: \(myTown.numberOfStopLights)")
```

This code does three things.

First, you create an instance of the `Town` type. You did so by typing the name of the type that you wanted to get an instance of (in this case, it is `Town`) followed by empty parentheses `()`. Second, you set this new instance equal to a variable you called `myTown`.

Third, you used string interpolation to print the values of the `Town` struct's two properties to the console. Notice that you used dot syntax to access the properties' values. For example, the syntax `myTown.population` retrieves the population of the `myTown` instance. Hence, the output should read: "Popuation: 5422, stop lights: 4".

Instance Methods

The **`println()`** function above is a fine way to print a description of `myTown`, but a town should know how to describe itself. Create a function on the struct `Town` that prints the values of its properties to the console. Navigate to your `Town.swift` file and add the following function definition.

```
struct Town {
     var population = 5422
     var numberOfStopLights = 4

     func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfStopLights)")
     }
}
```

The **printTownDescription()** function is called a method because it is a function that is associated with a particular type. **printTownDescription()** takes no arguments and returns nothing. Instead, its main purpose is to log a description of the town's properties to the console.

To make use of your new instance method, you need to call the function on an instance of `Town`. Navigate to `main.swift` and replace the **println()** function with your new instance method.

```
var myTown = Town()
~~println("Population: \(myTown.population),~~
~~         number of stop lights: \(myTown.numberOfStopLights)")~~
myTown.printTownDescription()
```

You use dot-syntax to call a function on an instance: `myTown.printTownDescription()`. Go ahead and run your program; you should see the same console output as you did before.

Mutating Methods

Your **printTownDescription()** function is great for displaying your town's current information, but what if you need a function that changes your town's information? You will need to write a *mutating* function to do this. If an instance method on a struct mutates any of the struct's properties, then that instance method needs to be marked as mutating. In `Town.swfit`, add a new function to the `Town` type to increase a town instance's population.

```
struct Town {
     var population = 5422
     var numberOfStopLights = 4

     func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfStopLights)")
     }

     mutating func changePopulation(amount: Int) {
        population += amount
     }
}
```

Note that you marked the instance method **changePopulation()** with the `mutating` keyword. As in Chapter 14, doing so means that this method can change the values in the struct. The point to make here is that both structures and enumerations are value types and require mutating.

The method has one argument, `amount`, that is of the `Int` type. This parameter is used to increase the town's population: `population += amount`. Switch over to `main.swift` to exercise this function.

```
var myTown = Town()
myTown.changePopulation(500)
myTown.printTownDescription()
```

As before, you used dot syntax to call the function on your town. If you build and run the program, you should see that `myTown`'s population has been increased by 500: `Population: 5922; number of stop lights: 4`.

Classes

A Monster Class

Now that you have a struct representing a town, it is time to make things a little more interesting. Imagine, for example, that your town is in Transylvania. In this case, your town would have to worry about any number of monsters that wander the country. Begin by making a new file called `Monster.swift`. Add the code below to make your file match.

```
class Monster {

}
```

You will notice that the syntax to define a new class instance is very similar to the syntax used to define a new struct instance. The difference lies in the use of the `class` keyword. As before, the definition of the class takes place between the braces: `{}`.

For reasons relating to inheritance (discussed in the next section), the class `Monster` is defined in very general terms. This means that the `Monster` type will describe the general shape of a monster.

```
class Monster {
    var town: Town? = Town()
      var name = "Monster"

    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

Historically speaking, it is well known that monsters do one thing very well: they wander around terrorizing towns. Hence, the `Monster` type has an optional property for the town that the monster will terrorize. Recall that optionals are used when an instance may potentially become `nil`. Thus, the `town` property is an optional of type `Town?` because the monster may or may not yet have found a town to terrorize. You also created a property for the `Monster`'s name, and gave it a generic default value.

Next, you defined a basic stub for a function called **`terrorizeTown()`**. This function will be called on an instance to terrorize the monster's town. Notice that you used optional binding to check if the instance has a `town`. If it does, then it will log to the console the name of the monster wreaking havoc. If the instance does not have a town yet, then the method will log that information.

As each sort of monster will terrorize a town differently, subclasses should provide their own implementation of this method.

Switch to your `main.swift` to exercise the `Monster` class. Add the following code to get an instance of this type, give it a town, and call the **`terrorizeTown()`** on it.

```
var myTown = Town()
myTown.changePopulation(500)
myTown.printTownDescription()
let gm = Monster()
gm.town = myTown
gm.terrorizeTown()
```

First, you create an instance of the `Monster` type called `gm` (for generic monster). This instance is declared as a constant because there is no need for it to be mutable. Next, you assign `myTown` to `gm`'s `town` property. Finally, you call the **`terrorizeTown()`** function on the `Monster` instance. You should see `Monster is terrorizing a town!` log to the console.

Inheritance

One of the main features that classes have that structures do not is inheritance. Inheritance refers to a relationship wherein one class, a subclass, is defined in terms of another, a superclass. The nature of this relationship means that a subclass *inherits* the properties and methods of its superclass. In a sense, inheritance defines the genealogy of class types.

The fact that classes can take advantage of inheritance is the primary reason why you made the `Monster` type a class in the first place. Make a `Zombie` type that inherits from the `Monster` type to see this relationship in more practical terms.

A Zombie Subclass

Create a new Swift file called `Zombie`. This file will hold the definition of a new class describing a zombie. The `Zombie` type will inherit from the `Monster` type. Add the following class declaration to see how.

```
class Zombie: Monster {
    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
}
```

The class above defines a new type called `Zombie`. It inherits from the `Monster` type, which is indicated by the colon (`:`) after `Zombie`. Inheriting from `Monster` means that `Zombie` has all of `Monster`'s properties and methods (i.e., the `town` property and the **`terrorizeTown()`** function).

`Zombie` also adds a new property. The property is called `walksWithLimp` of type `Bool`. This type is inferred from the property's default value: `true`.

Last, `Zombie` overrides the **`terrorizeTown()`** method. Note the use of `override`. The `override` keyword is needed when a subclass is providing its own definition of a method that is defined on a superclass. In fact, failing to add `override` will result in a compiler error.

Recall that `Zombie`'s `town` property is an optional of type `Town?` that is inherited from the `Monster` type. The fact that `town` is an optional means that it can potentially be `nil`. Thus, you need to make sure that an instance of `Zombie` has a town to terrorize before calling any methods on the `town`. How can you make this check?

One possible solution is to use optional binding. You may be tempted something like this:

```
if let terrorTown = town {
     // do something to terrorTown
}
```

In the code above, if the `Zombie` instance has a `town`, then the value in the optional is unwrapped and put into the constant `terrorTown`. After which, this value is ready to be terrorized, but with an important caveat: the value semantics of structs means that the `terrorTown` instance will not be the same as the `town` instance. The problem with this fact is that any changes made on `terrorTown` will not be reflected in the `Zombie` instance's `town` property. In addition to this limitation, this code can also be more concise.

As you have seen in the chapter on Optionals, optional chaining allows this check to be done on a single line. It is just as expressive, and is also more concise. For example, `town?.changePopulation(-10)` performs the same function as optional binding. If the optional `town` has a value, then the method **`changePopulation()`** is called on that instance.

This code helps you to make sure it is safe to call a function on the instance. Thus, when an instance of the `Zombie` type terrorizes a town, its population will be decreased by 10 people.

Last, notice the line: `super.terrorizeTown()`. `super` is a prefix you can use to access a superclass's implementation of a method. In this case, you used `super` to call the `Monster` type's implementation of **`terrorizeTown()`**. Thus, you should once again see `Monster is terrorizing a town!` log to the console.

Since `super` is predicated upon the idea of inheritance, it is not available to value types like enums or structs. It is typically invoked to borrow or override functionality from a superclass.

Preventing Overriding

Sometimes you want to prevent subclasses from being able to override methods or properties. The need to do this is rare in practice, but it comes up occasionally. In these cases, you use the `final` keyword to prevent a method or property from being overridden.

Imagine, for example, that you specifically do not want subclasses of the `Zombie` type to provide their own implementation of the **`terrorizeTown()`** function. Perhaps you have good reason to assume that all subclasses of `Zombie` should terrorize their towns in the exact same way. Add the `final` keyword to this function's declaration.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true
    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    final override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

Now, subclasses of the `Zombie` class will not be able to override the **`terrorizeTown()`** function. Go ahead and create a new subclass of `Zombie`; name it `ZombieBoss`. Try to override the **`terrorizeTown()`** function.

```
class ZombieBoss: Zombie {
    override func terrorizeTown() {
        println("terrorizing town...")
    }
}
```

You should see the following error on the line where you try to override the **`terrorizeTown()`** function: `Instance method overrides a 'final' instance method.`. The error is telling you that you cannot override this function because it is marked as `final` in the superclass. Go ahead and delete this new file; you will not be using it later on.

Our Town Has a Zombie Problem

Now is a good time to exercise the `Zombie` type. Choose the `main.swift` file from the project navigator on the left. Write some code to create an instance of the `Zombie` class. Note that you should delete the code that prints the town's description to free the console from clutter; you also delete the code that created a generic instance of the `Monster` type as you no longer need it.

```
var myTown = Town()
myTown.changePopulation(500)
myTown.printTownDescription()
let gm = Monster()
gm.town = myTown
gm.terrorizeTown()
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
```

You first create a new instance of the `Zombie` type named `fredTheZombie`. Next, you assign your preexisting instance of the `Town` type, `myTown`, to the `Zombie` type's property `town`. At this point, `fredTheZombie` is free to terrorize a town, which he will do wtih alacrity.

After `fredTheZombie` has terrorized the time, it is good to check the results with the **`printTownDescription()`**. Notice, however, that this is not as straightforward has it has been before. Unfortunately, for reasons discussed soon, you cannot call the **`printTownDescription()`** function on the `myTown` instance. Thus, you want to print out the description of `fredTheZombie`'s town.

Since `fredTheZombie`'s `town` property is an optional of type `Town?`, you have to unwrap it before you can call the **`printTownDescription()`** function on it. As before, you can do this operation with optional chaining with the question mark operator: `fredTheZombie.town?`. This code ensures that the zombie has a town inside of the optional before you try to use the description function.

After `fredTheZombie` is done terrorizing its town, the console output should read: `"Population: 5912; number of stop lights: 4"`.

Method Parameter Names

At this point, you may have noticed that the conventions that Swift follows for external parameter names is different between the global functions that were used at the beginning of this book and the methods used in this chapter. A global function's parameters are not given external names by default. For example, remember the **`divisionDescription`** global function -- that is, a function that is not defined on a specific type -- from Chapter 12:

```
func divisionDescription(num: Double, den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(9, 3)
```

`divisionDescription` has two arguments: `num` and `den`. Notice that you did not use these parameter names when you called the function. You would have needed to use explicit parameter names or the shorthand syntax if you wanted that behavior.

Methods on types work a little differently. The first argument of a type or instance method does not get an external name, but all other arguments get external names as though you prefixed them with `#`. If a type or instance method only has one argument, the same rule applies: that argument does not get an external name.

To see this difference in action, open the `Zombie.swift` file to add a new function to that class. Add a new function to change the name and limp status of an instance of the `Zombie` class.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

The new function **`changeName`** has the following signature: `changeName(_: walksWithLimp:)`. It is a simple function that allows a developer to change a `Zombie`'s `name` and `walksWithLimp` properties. The underscore (_) signifies that the first parameter of the function is unnamed; that is, you will not use this parameter name when calling the function. In contrast, `walksWithLimp` is used when calling the function, as indicated by the function's signature.

Switch to `main.swift` to exercise this function and see how it is called. Add a call to this new function to the bottom of the file.

```
...
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

You call **`changeName(_: walksWithLimp:)`** on `fredTheZombie`. Notice that the name of the first argument is omitted, but the name of the second argument is used. You wrote the name of the first argument in the actual function name -- the "Name" portion of **`changeName`**. This convention is a common pattern in naming methods in iOS and Mac OS X frameworks. The word 'name' maps onto the first property that you would like to modify with this function, and `walksWithLimp` matches the second property. Naming the function in this way makes it easy to see that `"Fred the Zombie"` is the `String` that you want to assign to the `name` property.

The discrepancy between global functions and type methods may feel strange. The reason for the difference is related to Swift's relationship to Objective-C and C. For now, all you need to know is that global functions do not give any external parameter names by default, while type methods do not give free external parameter names for the first argument, but do give external names to all subsequent parameters.

Value Types vs. Reference Types

In the above example of a `Zombie` instance terrorizing a town, you called **`printTownDescription()`** on `fredTheZombie`'s `town`. You might think that this instance is the same instance of `Town` that is in `myTown`, but that is not true. For posterity, call **`printTownDescription()`** on `myTown`.

```
var myTown = Town()
myTown.changePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
myTown.printTownDescription()
```

You should see that the console logs different values for the `population`. That is, `myTown` has a population of 5922 and not 5912 like `fredTheZombie`'s town. Why are these two values different? The answer has to do with the difference between value types and reference types.

Value types are types whose values are copied when they are assigned to another instance or passed in the argument of a function. This means that assigning an instance of a value type to another actually assigns a copy of the first instance to the second instance. Value types play an important role in Swift. For example, arrays and dictionaries are both value types. All enums and structs you write are value types as well.

Note that `Town` is a struct. Thus, when `myTown` was assigned to `fredTheZombie` (`fredTheZombie.town = myTown`), a copy of `myTown` was created, and it was that copy that was assigned to `fredTheZombie`'s `town`. Thus, when `fredTheZombie` terrorizes his town, it is only his instance of `Town` that is updated. The original `myTown` is left unchanged.

References types are not copied when they are assigned to an instance or passed into an argument of a function. Instead, a reference to the same instance is passed. Classes and functions are reference types. Accordingly, since `Zombie` is a class, it is therefore a reference type. Make a couple of more instances of `Zombie` inside of `main.swift` to see how reference types differ from value types.

```
var myTown = Town()
myTown.changePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
~~myTown.printTownDescription()~~
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
println("z1 walks with limp? \(z1.walksWithLimp); z2 walks with limp? \(z2.walksWithLimp)")
```

First, you created a new instance of `Zombie` and put a reference to that instance into a constant called `z1`. Next, you set `walksWithLimp` to `false` on `z1`. You then created a new constant by assigning `z1` to `z2`. Doing so set `z2` to point to the same instance of the `Zombie` type that `z1` pointed to. This point is confirmed by the console output generated by the **`println()`** function above: `"z1 walks with limp? false; z2 walks with limp? false"`.

To underline the point about reference types, change the value of `walksWithLimp` on `z2`. Be sure to set `walksWithLimp` on `z2` before you log the output to the console.

```
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \(z1.walksWithLimp); z2 walks with limp? \(z2.walksWithLimp)")
```

As you can see, both `z1` and `z2` walk with a limp: `"z1 walks with limp? true; z2 walks with limp? true"`. This result may feel confusing; after all, you only set `walksWithLimp` on `z2`. Nevertheless, recall that reference types do not create a copy of the instance during assignment. Instead, the new constant or variable is given a reference to the same instance as the constant or variable on the righthand side of the assignment. Thus, both of the `z1` and `z2` instances have the same reference to the instance of the `Zombie` type in memory. Changing the value of the `walksWithLimp` on one of these constants also changes the value of the other because they both point to the same instance of the `Zombie`.

Interestingly, you can change the value of `walksWithLimp` on `z1` and `z2` even though they are declared as constants. Why does this work? Understanding that reference types do not actually store the instance of the type itself helps to clear up this question. Put differently, the two constants `z1` and `z2` contain a reference to the same instance of the `Zombie` class. If you change a value on that instance, you are actually leaving the reference to it intact and unchanged. This distinction is important since `z1` and `z2` are merely constants holding onto a reference that points to an underlying instance somewhere in memory. If you try to reassign the reference held onto by either of these constants, perhaps by creating a new instance of the `Zombie` class, then the compiler will issue an error. Thus, it is important to remember that there is a distinction between the reference to an instance and the instance itself.

Identity vs. Equality

Now that you understand the difference between value and reference types, you are in a great position to learn about equality and identity. Equality refers to two instances having the same values for their observable characteristics; e.g., two instances of the `String` type that both have the same text. Identity, on the other hand, refers to whether or not two variables or constants point to the same instance in memory. For example, take a look at the following code.

```
let x = 1
let y = 1
x == y // true
```

Two constants, `x` and `y`, are created. They are both of type `Int` and hold onto the same value, 1. Not surprisingly, the equality check, done via ==, evaluates to `true`., which makes sense because `x` and `y` hold onto exactly the same value. This is exactly what we want to know from an equality check: do two instances have the same values? All of Swift's basic data types (`String`, `Int`, `Float`, `Double`, `Array`, and `Dictionary`) can be checked for equality.

`z1` and `z2` are both reference types because they point to an instance of the `Zombie` class. Thus, you can check for identity on these two constants using the identity operator: ===. See below:

```
z1 === z2 // true
```

The above identity check works because both `z1` and `z2` point to the same location in memory where an instance of the `Zombie` class lives.

But what if you wanted to check for identity on `x` and `y`? You might think you could use the identity operator: `x === y`. This code will generate an error from the compiler. Why? The reason for the error comes from the fact that value types are passed by value. Since the `Int` type is implemented in Swift as a struct, both `x` and `y` are value types. Thus, you cannot compare these two constants based upon their location in memory.

Incidentally, what happens if you try to check for equality on `z1` and `z2`: `z1 == z2`? You should see a compiler error telling you that you cannot invoke == with an argument list of type `(Zombie, Zombie)`. In English, the compiler is telling you that it does not know how to call the **==** function on the `Zombie` class. If you want to check for equality on classes that you make, then you will have to teach your classes how by implementing the **==** function. Doing so entails conforming to a protocol called `Equatable`, which is a subject for a later chapter.

As a final note, it is important to realize that two constants or variables may be equal (i.e., they have the same values), but they may not be identical (i.e., they may point to distinct instances of a given type). Notably, it does not work the other way around. If two variables or constants point to the same instance in memory, then they will be equal as well.

What Should I Use?

Structures and classes are well-suited for defining many custom types. Time was that structs were so distinct from classes that the use-cases for both of these types were obvious. In Swift, however, the functionality added to structs makes their behavior more similar to that of classes. This similarity renders the decision of which to use somewhat more complicated.

Nonetheless, there are important differences between structs and classes that help to give some guidance of which to use when. Here are some recommendations.

1. If you want a type to be passed by value, then use a struct. Doing so will ensure that the type is copied when assigned or passed into a function's argument.

2. If you do not want the type to be inherited, then use a struct. Structs do not support inheritance, and so they cannot be subclassed.

3. If the behavior you would like to represent in a type is relatively straightforward and encompasses on a few simple values, the consider starting out with a struct. You can always change the type to be a class in future.

4. Use a class in all other cases.

The items above represent a set of general guidelines; they are by no means explicit rules. To be sure, hard and steadfast rules on this topic are hard to define. A few examples can help to make this issue a little bit more clear. Structs are commonly used when modeling shape sizes (e.g., rectangles have a width and a height), ranges (e.g., a race has start and an end), and points in some coordinate system (e.g., a point in a 2-Dimensional space has an X and Y value). They are also great for defining data structures; indeed, the `String`, `Array`, and `Dictionary` types are all defined as structs in Swift's standard library.

These examples are by no means exhaustive. There is a lot of room for experimentation and creativity here. In general, we suggest starting out with a struct unless you absolutely know you need the benefits of a reference type. Value types are a bit easier to reason about because they are somewhat "safer" to work with (e.g., you do not need to worry about what happens to an instance when you change values on a copy).

Challenge

Bronze Challenge

There is currently a bug in the `Zombie` type. If an instance of `Zombie` terrorizes a town with a population of 0, then its population will decrement to -10. This result does not make sense. Fix this bug by changing the **`terrorizeTown()`** function on the `Zombie` type to only decrement the town's population if its population is greater than 0. Also make sure that the town's population is set to 0 if the amount to decrement is greater than the current population.

Silver Challenge

Create another subclass of the `Monster` type. Call this one `Vampire`. Override the **`terrorizeTown()`** function so that every time an instance of the `Vampire` type terrorizes a town, it adds a new vampire thrall to an array of vampires on the `Vampire` type. This array of vampire thralls should be empty by default. Terrorizing a town should also decrement the vampire's town's population by one. Last, exercise this `Vampire` type in `main.swift`.

For the More Curious: Type Methods

In this chapter, you defined some instance methods that were called on instances of a type. For example, **`terrorizeTown()`** is an instance method that you can call on instances of the `Monster` type. You can additionally define methods that are called on the type itself. These are called type methods. Type methods are useful for working with type level information. For example, imagine the following struct named `Square`.

```
struct Square {
    static func numberOfSides() -> Int {
        return 4
    }
}
```

For value types, you indicate that you are defining a type method with the `static` keyword. The method **`numberOfSides()`** simply returns the number of sides a `Square` can have.

In distinction, type methods on classes use the `class` keyword. Imagine, for example, that you would like to a type method to the `Zombie` class that represents the spooky noise that all zombies make.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
    }
}
```

To use these type methods, you will simply call them on the type itself. Take a look at the code below for examples:

```
let sides = Square.numberOfSides() // sides is 4
let spookyNoise = Zombie.makeSpookyNoise() // spookyNoise is "Brains..."
```

Type methods can work with type level information on a given type. This means that type methods can call other type methods, and can even work with type properties, which we will cover in the Properties chapter. Note, however, that type methods cannot call instance methods or work with any instance properties. The reason for this limitation is that an instance is not available for use at the type level.

For the More Curious: Function Currying

After working through this chapter, you might be wondering about the `mutating` keyword. Why is it needed to allow you to modify a struct or enum? It turns out that understanding a new concept called *function currying* helps to explain the answer.

Create a new playground called `Curry.playground` and save it where you like.

Function currying allows you to rewrite an existing function that takes multiple parameters as a new function that takes one parameter and returns another function. The function you return takes the original function's remaining parameters and also returns what the original function returns. This process of nesting functions, each with the remaining number of parameters, continues until there are no remaining parameters.

The rewritten function is called a *curried function*. A curried function partially applies an existing function. That is, a curried function allows you to bind values to a function's arguments before you call it. This feature of curried functions is similar to supplying default values to a function's parameters, but is far more dynamic.

Consider a simple function that returns a `String` greeting.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}
```

The **`greetName`** function takes two arguments: a `greeting` and a `name`. It constructs and returns a greeting based on these two arguments. This function is straightforward to use.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greetName("Hello,", "Matt")
println(personalGreeting)
```

You can rewrite **`greetName`** to be a curried function.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greetName("Hello,", "Matt")
println(personalGreeting)

func greetingForName(greeting: String) -> (String) -> String {
    func greet(name: String) -> String {
        return "\(greeting) \(name)"
    }
    return greet
}
```

The function **greetingForName** takes one argument, the `String` `greeting`, and returns a function. This returned function itself takes a `String`, representing the `name` to greet, and returns a `String` greeting for the given `name`. You defined a nested function called **greet** inside of the implementation for **greetingForName**. **greet**'s function type matches the type specified by **greetingForName**; it takes a `String` and returns a `String`. Notice that you combine the `greeting` parameter with the `name` parameter from the two functions to construct the personalized greeting. Finally, you returned the **greet** function.

Add the following code to use the curried function.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greetName("Hello,", "Matt")
println(personalGreeting)

func greetingForName(greeting: String) -> (String) -> String {
    func greet(name: String) -> String {
        return "\(greeting) \(name)"
    }
    return greet
}

let greeterFunction = greetingForName("Hello,")
let greeting = greeterFunction("Matt")
println(greeting)
```

You call the **greetingForName** function and pass in the desired greeting (`"Hello,"`). The result is assigned to a constant named `greeterFunction`. `greeterFunction` holds a function that matches the return type of **greetingForName**: it takes a `String` and returns a `String`. The specific greeting, `"Hello,"`, is passed along in the enclosing scope of the **greet** function that is returned by **greetingForName**.

To make a personalized greeting for a specific name, you call the **greet** function and pass it a name (`"Matt"`) to its only parameter. The result of this function is assigned to `greeting`, which you log to the console. You should see the same result log to the console.

Thankfully, Swift supplies a more convenient syntax for writing curried functions. The example below is equivalent to what you have just written.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greetName("Hello,", "Matt")
println(personalGreeting)

func greetingForName(greeting: String) -> (String) -> String {
    func greet(name: String) -> String {
        return "\(greeting) \(name)"
    }
    return greet
}

let greeterFunction = greetingForName("Hello,")
let greeting = greeterFunction("Matt")
println(greeting)

func greeting(greeting: String)(name: String) -> String {
    return "\(greeting) \(name)"
}

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)
```

The **greeting** function's syntax looks a little different from what you have just seen. This time, you separated each argument by enclosing each parameter within its own parentheses. Notice that this syntax is much more concise, but it works just the same.

Calling this curried function works very similarly to what you have just written. You called **`greeting`** and passed in a `String` greeting to its first argument. The resulting function was assigned to a constant called `friendlyGreeting`. Next, you called the function in `friendlyGreeting`, and passed in a name to greet. Note, the second argument, `name`, was automatically given an external name that you had to use.

Check the console at this time. You should see that the results are the same.

Now that you understand how function currying works, you are in a good position to examine how the `mutating` keyword works. To begin, create a new struct called `Person`.

```
...

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}
```

There is nothing very special or unfamiliar taking place here. The `Person` struct has properties for a person's first and last names. It also defines a mutating function to change these properties.

Create a new instance of the `Person`.

```
...

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}
var p = Person()
```

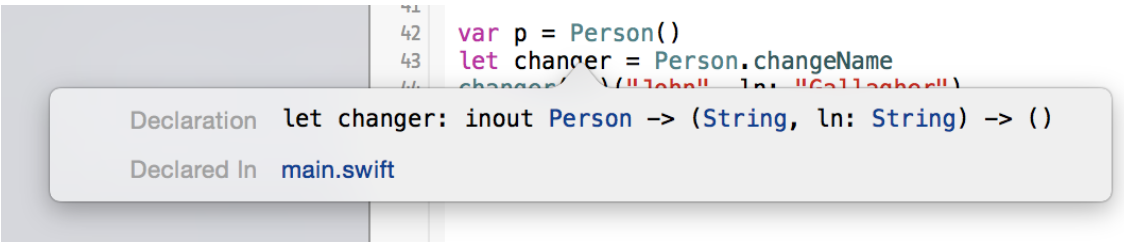
There is nothing new here either, but here is where things start to get interesting. It turns out that Swift's instance methods, the very ones that you learned about in this chapter, are actually curried functions. Type in the following code to see this fact in action.

```
...

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
```

You can access the **changeName** function on the Person struct. Notice that you are not calling the **changeName** function. Instead, you are assigning it to a constant called changer.

Just what is changer? To find out, hold down the option key and click on it. You should see something similar to Figure 15.6.

Figure 15.6 A Curried Function Signature



What does that signature mean? In short, it tells you that changer is a curried function. More specifically, changer holds a function whose only argument is an instance of the Person struct passed in as an inout parameter. This function returns a function that takes two arguments, a String for the new first name and a String for the new last name. The resulting function returns nothing.

Recall from Chapter 12 that an inout parameter allows a function to modify the value passed into that parameter. The changes on the inout parameter made within the function also persist outside of the function after it is called. In other words, the modifications replace the parameter's original value.

Putting all of this information together, a mutating function is simply a curried function whose first argument is self passed in as an inout parameter. Since value types are copied when they are passed, for non-mutating methods, self is actually a copy of the value. In order to make changes, self needs to be declared as inout, and mutating is the way Swift allows you to accomplish that.

Type in the following to demonstrate this point and to see changer in action.

```
...

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
changer(&p)("John", ln: "Gallagher")
println(p.firstName) // John
```

You call changer's function, passing in the instance of Person that you want to modify. Remember that you need to prefix inout parameters with an & to ensure that you pass in the instance's reference to the function. Next, you give two strings ("John" and "Gallagher") to the curried function's final two parameters, one each for the first and last names. These strings are used to modify the Person instance's values for those properties. Last, you print out the result of the function call to confirm that p's first name has been changed to John.

The point of this section was to show what the keyword mutating does. In practice, you will likely not want to use function currying to mutate a struct. Go ahead and remove the function currying code, and make use of the **changeName** function more directly. The result will be the same.

```
...

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
changer(&p)("John", ln: "Gallagher")
p.changeName("John", ln: "Gallagher")
println(p.firstName) // John
```

16
Properties

The previous chapter introduced properties to a very limited extent. Its focus was on structures and classes. Nonetheless, types need properties so that they can model data in your application. Thus, you gave your custom types some very basic stored properties so that they had data to represent. This chapter will further develop your understanding of properties, and will deepen your understanding of how to use them with your custom types.

As you have already seen, properties are used to associate values with a given type. Properties can take constant or variable values for a type. Classes, structures, and enumerations can all have properties. In short, properties are used to model the characteristics of the entity that the type is developed to represent.

Properties can be of two varieties: 1) stored, and 2) computed. Stored properties can be given default values, and computed properties can return the result of some calculation based upon available information. You can observe properties for when they are changed, and can then subsequently execute specific code when the property is set to a new value. You can even establish rules that determine properties' visibility to other files in your application. To be sure, properties have a lot of power and flexibility. Let's get to it.

Basic Stored Properties

Stored properties are propterties in their most basic form. To see how they work, you will be expanding the behavior of the types you developed in the Chapter 15. Make a copy of your `MonsterTown.xcodeproj` project, and save it to a convenient location.

Open up the `Town.swift` file once you have made a copy of the project. Recall the `population` property: `var population = 5422`. This code signifies three important items.

- `var` marks this property as variable, which means that it can be mutated.
- `population` has a default value of 5422.
- `population` is a stored property whose value can be read and set.

`population` is referred to as a stored property because it is used to hold onto a specific sort of information: the town's population. That is what stored properties do; they store data.

One important point to reiterate is that the above stored property, `population`, is "read/write". In other words, you can both read the property's value *and* set the property's value. You can also make stored properties "read-only"; that is, you can make a property that does not allow a user to change its value.

Use `let` with a stored property to create a "read-only" property. Imagine, for example, that the specific town that you need to model in your application will always be in the South of the USA. A constant stored property is a good use for this case.

```
struct Town {
    let region = "South"
    var population = 5422
    var numberOfStopLights = 4

    func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfStopLights)")
    }

    mutating func changePopulation(amount: Int) {
        population += amount
    }
}
```

`region` is a stored property that stores the region the town is in. In this case, you made `region` a constant, which means that the property cannot be changed. This immutability effectively makes `region` a read-only property.

Nested Types

Nested types are types that are defined within another enclosing type. They are often used to support the functionality of a type, and are not intended to be used separately from that type. Enumerations are frequently used in this manner.

Inside of `Town.swift`, create a new enumeration called `Size`. You will be using this enumeration, in coordination with another new property to be added later, to calculate whether or not a town can be designated as small, medium, or large. Make sure that you define the enum within the definition for the `Town` struct:

```
struct Town {
    let region = "South"
    var population = 5422
    var numberOfStopLights = 4

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }

    func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfStopLights)")
    }

    mutating func changePopulation(amount: Int) {
        population += amount
    }
}
```

Notice that `Size` is defined within the braces (`{}`) of the definition of the `Town` struct. `Size` has raw `String` values associated with each case that describe the town's size. The case's default values will help to describe the town's size when printing to the console.

`Size` also provides a function that will transform its current enum case into a helpful `String` value.

`Size` will not be used outside of `Town`. Indeed, its definition suggests that `Size` will use an instance of the `Town` type to determine what size of town the instance is. This reality suggests that the instance of `Town` will need a value in its `population` property before this nested type is used. Yet, all of the properties you have worked with thus far have calculated the property's value when the instance was created. The next section introduces a new sort of property that delays the computation of its value such that it is calculated when the correct information is available.

Lazy Stored Properties

Sometimes a stored property's value cannot be assigned immediately. A developer may not want to compute the values of properties immediately when the instance is created because the computation may be costly in terms of memory or time. Another potential scenario may entail a property that depends on factors external to the specific type that will be unknown until after the instance is created. These circumstances call for something named "lazy loading".

In terms of properties, lazy loading means that the calculation of the property's value will not occur until the first time it is needed. This delay will defer computation of the property's value until after the instance is initialized. As such, `lazy` properties must be declared with `var` because their values will change.

Create a new `lazy` property called `townSize`. This property will be of type `Size`; that is, its value will be an instance of the `Size` enum. Make sure to define this new property inside of the `Town` type.

```
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
lazy var townSize: Size = {
    precondition(self.population >= 0, "Town cannot have negative population.")
    switch self.population {
    case 0...10000:
        return Size.Small

    case 10001...100000:
        return Size.Medium

    default:
        return Size.Large
    }
}()
```

Compared to the properties that you have written before, `townSize` looks different for several reasons. First, notice that you marked `townSize` as `lazy`. Doing so means that `townSize`'s value will only be calculated once it is first accessed. Delaying the computation of this property makes sense since its value depends upon the instance's `population`.

Second, the type of the property is `Size`, as mentioned above. What is unique here is that you will not be setting the value of this property directly. For example, you will not be writing code like this: `myTown.townSize = Size.Small`. Instead, you will take advantage of the nested type `Size` in coordination with a closure to calculate the town's size given its population.

Last, notice this line of code: `lazy var townSize: Size = {`. What is going on here? To understand, it is important to recall that functions and closures are "first-class types". A consequence of this fact is that properties can reference functions and closures. With this knowledge refreshed and in hand, take another look at the definition of `townSize`.

`townSize` sets a default property value, i.e., the town's size, with the result returned by a closure. A closure works well here because the value of the town's `population` is needed in order to determine the town's size. Thus, the closure, which is defined in between the braces (`{}`), uses a `switch` statement to determine how large the town is. Thus, the closure switches over the instance's `population` (`self.population`).

Notice that you added a `precondition` before the `switch` statement. This `precondition` ensures that the town instance has a viable population to `switch` over. That is, it ensures that the town does not have a negative `population`, which is impossible, and would not be captured by the `switch`'s cases.

Inside of the `switch` statement, you specified three cases: 1) `0...10000` is a small town, 2) `10001...100000` is a medium-sized town, and 3) a `default` case that will capture any `population` larger than 100000 and describe it as a large town. Each case returns an instance of the enum `Size`.

Also notice that the closure for `townSize` ends with an empty parentheses `()`. These parentheses, combined with the lazy marking, ensure that Swift will call the closure and assign the result it returns to `townSize` when the property is accessed for the first time. If you had omitted the parentheses, then you would simply be assigning a closure to the `townSize` property. Thus, the closure will be executed the first time you access the `townSize` property.

Finally, `townSize` needs to be marked `lazy` because the closure needs to `switch` over the instance property `population`. If a closure works with an instance's properties, then the compiler requires that the closure references `self` when accessing any property on that instance. The reason for this requirement is that referencing `self` within a closure may cause a memory leak. (Do not worry about memory management issues right now; they will be discussed later on in the book.) Furthermore, recall that `lazy` means that computation of the property's value is deferred until after the instance is initialized. Therefore, because the closure needs to reference `self` in order to gain access to the instances's `population` property, then that means the property `townSize` needs to be marked as `lazy` in order to ensure that the instance (what `self` references) is fully prepared to do work.

Switch to `main.swift` to exercise this `lazy` property.

```
var myTown = Town()
let ts = myTown.townSize
println(ts.rawValue)
myTown.changePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \(z1.walksWithLimp); z2 walks with limp? \(z2.walksWithLimp)")
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

After you created an instance, you created a constant named `ts` to hold `myTown`'s size information. This line accesses the lazy property `townSize` and causes its closure to execute. After the closure switches over `myTown`'s `population`, an instance of the `Size` enum is assigned to `ts`. Next, you used the `rawValue` property on `ts` to log its raw value to the console. As a result, you should see `Small` log to the console.

It is important to note that properties marked with `lazy` will only be calculated one time. This feature of `lazy` means that changing the value of `myTown`'s `population` will never cause `myTown`'s `townSize` to be recalculated. Increase `myTown`'s `population` by 1000000, and then check `myTown`'s size by logging it to the console. Go ahead and include `myTown`'s `population` in the log statement to reference it against what is reported for `townSize`.

```
var myTown = Town()

let ts = myTown.townSize
println(ts.rawValue)

~~myTown.changePopulation(500)~~
myTown.changePopulation(1000000)
println("Size: \(myTown.townSize.rawValue); population: \(myTown.population)")
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \(z1.walksWithLimp); z2 walks with limp? \(z2.walksWithLimp)")
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

You should see the following log to the console: `Size: Small; population: 1005422`. As predicted, `myTown`'s size has not changed even though its `population` increased dramatically. This discrepancy is due to `townSize`'s `lazy` nature. The property will only be calculated upon first access and will not be recalculated thereafter.

Such a discrepancy between `myTown`'s `population` and `townSize` is undesirable. Furthermore, it seems like `townSize` shouldn't be marked `lazy`, especially if `lazy` means that `myTown` will not be able to recalibrate its `townSize` to reflect `population` changes. It turns out that a *computed property* is a better option.

Computed Properties

You can use computed properties with any class, struct, or enum that you define. Computed properties do not store values like the properties that you have been working with thus far. Instead, a computed property provides a getter and optional setter to retrieve or set the property's value. To make this point more practical, replace your definition of the `townSize` property on the `Town` type with a computed read-only property.

```
lazy var townSize: Size = {
    precondition(self.population >= 0, "Town cannot have negative population.")
    switch self.population {
    case 0...10000:
        return Size.Small

    case 10001...100000:
        return Size.Medium

    default:
        return Size.Large
 }
}()
```

```
var townSize: Size {
    get {
        precondition(self.population >= 0, "Town cannot have negative population.")
        switch self.population {
        case 0...10000:
            return Size.Small

        case 10001...100000:
            return Size.Medium

        default:
            return Size.Large
        }
    }
}
```

`townSize` is now defined as a computed property. It offers a custom getter that uses the same `switch` statement that was used in the previous closure-based definition of this property. Notice that you explicitly declared the type of the computed property to be `Size`. You must provide computed properties with their type information. This information helps the compiler to know what the property's getter should return. Also of note is that computed properties must be declared as variables with the `var` keyword.

As before, you access this property via dot syntax: `myTown.townSize`. Doing so will execute the getter for `townSize`, which will result in using `myTown`'s `population` to calculate the `townSize`. Run your program again as it is currently written. You should see `Size: Large; population: 1005422` logged to the console after you increase the the town's population by 1000000.

`townSize` is a read-only computed property. In other words, `townSize` cannot be set directly. It can only retrieve and return a value based upon the calculation you defined in the getter. A read-only property is perfect in this case because you want `myTown` calculate its `townSize` based upon the instance's `population`.

A Getter and a Setter

Computed properties can also be declared with both a getter and a setter, making the property a read/write property. Open your `Monster.swift` and add the following computed property to the declaration for the `Monster`.

```
class Monster {
    var town: Town? = Town()
    var name = "Monster"
    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

Imagine that you need each instance of the `Monster` to be able to keep track of it potential pool of victims. Naturally, this number will match the population of the town that the monster haunts. Accordingly, `victimPool` is a new computed property with both a getter and a setter. As before, you declared it as a `var` and gave it specific type information. In this case, `victimPool` is an `Int`.

You follow the property's type information with a brace ({). The property's definition is written in between this opening brace and a closing brance (}). You defined a getter for the property via the same `get`. The getter uses the `nil` coalescing operator to check if the `Monster` instance has a town that it is currently occupying. If it does, then it returns the value of that town's population. If the instance has not yet found a town to terrorize, then you simply return 0.

The setter for the computed property is written within the `set` block. Notice the new syntax: `set(newVictimPool)`. Specifying `newVictimPool` within the parentheses means that you are supplying an explicitly named new value. You can refer to this variable within the setter's implementation. For example, you used optional chaining to ensure that the `Monster` instance had found a town, and then set that town's popuation to match the `newVictimPool`. If you had not explicitly named the new value, then Swift would have provided a variable to you called `newValue` that held onto the same information.

Switch back to `main.swift` to use this new computed property. Add the subsequent code to the bottom of the file.

```
println("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
println("Victim pool: \(fredTheZombie.victimPool)")
```

The first new line exercises the getter for the computed property. Run the program and you should see `Victim pool: 1005412` log to the console. The next new line uses the setter to change `fredTheZombie`'s `victimPool`: `fredTheZombie.victimPool = 500`. Last, you once again log the `victimPool` to the console via the property's getter. Take a look at the console. The `victimPool` should be updated to be 500.

Property Observers

Swift provides an interesting feature called *Property Observation*. Property observers watch for and respond to changes in a given property. Property observation is available to any stored property that you define, and is also available to any property that you inherit. You cannot use property observers with computed properties that you define. This reality is not all that limiting considering that you have full control over the definition of that computed property's setter and getter, and can therefore respond to changes in whatever manner you like there.

Imagine that the citizens of your beleaguered town are getting very restless. They demand to the mayor that something is done to protect the people from the monstrous pox patrolling the countryside. The mayor's first action is to track the attacks on the townspeople. Property observers are perfect for this task.

You can observe changes to a property in one of two ways: 1) when a property is about to change: via `willSet`; and 2) when a property did change: via `didSet`. In order to keep track of how many attacks the town is suffering, the mayor decides to pay close attention to when the `population` of the town changes. Thus, you should use a `didSet` observer, as that observer will be notified right after the property receives a new value.

Add the following code to `Town.swift` to watch for changes to the `population` property.

```
var population: Int = 5422 {
    didSet(oldPopulation) {
        println("The population has changed to \(population) from \(oldPopulation).")
    }
}
```

The syntax for property observers looks similar to computed properties' getters and setters. The response to the change is defined within the braces. In the example above, you created a custom parameter name for the old population: `oldPopulation`. The `didSet` observer gives you a handle on the property's old value. In distinction, the `willSet` observer gives you a handle on the new value of the property. If you hadn't specified a new name, then Swift would have given you the parameter `oldValue` automatically. Likewase, in the case of the `willSet` observer, Swift generates a `newValue` parameter for you.

This property observer simply logs the town's `population` information to the console every time it changes. For example, you should see a log for this change after `fredTheZombie` terrorizes his town. Run the program and take a look at the console. You should see a log for every time the `population` changes.

```
Small
The population has changed to 1005422 from 5422.
Size: Large; population: 1005422
The population has changed to 1005412 from 1005422.
Population: 1005412; number of stop lights: 4
z1 walks with limp? true; z2 walks with limp? true
Victim pool: 1005412
The population has changed to 500 from 1005412.
Victim pool: 500
```

Type Properties

Up till now, you have been working with instance properties. Whenever you create a new instance of a type, that instance gets its own properties that are distinct from other instances of that type. Instance properties are useful for storing and computing values on an instance of a type, but what about values that belong to the type itself?

You can define *type properties*. These are properties that are universal to the type; the values in these properties will be shared across all of the type's instances. Typically, these properties store information that will be the same across all instances. For example, all instances of a `Square` type will have exactly four sides.

Value types (i.e., structures and enumerations) can take both stored and computed type properties. As with type methods, type properties on value types will begin with the `static` keyword.

Recall earlier in the chapter that you created a constant read-only property on the `Town` for the town's `region`. This use of a constant instance property is a little strange. After all, every instance of `Town` will be in the same region: the South. A type property would work better for the `region` property. Change the `Town` to reflect this revision.

```
~~let region = "South"~~
static let region = "South"
```

Stored type properties have to be given a default value. For example, `region` was given the value `South`. This requirement makes sense because types do not have initializers, which is the topic of Chapter 17. In other words, the stored type property needs to have all the information it needs in order to vend its value to any caller.

Unlike value types, classes can only have computed type properties. The syntax for a class type property uses the `class` keyword.

In a previous chapter, you created a type method on the `Zombie` type to make a spooky noise: **`makeSpookyNoise()`**. Notice that **`makeSpookyNoise()`** doesn't take any arguments. This fact makes the type method a great candidate for being rewritten as a computed type property. Revise the type method to be a computed type property.

```
class Zombie: Monster {
    ~~class func makeSpookyNoise() -> String {~~
        ~~return "Brains..."~~
    ~~}~~
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
    }

    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

Since you revised the type method to be a computed property, you changed the name to be `spookyNoise`. The definition of the computed property is very similar to the type method. The main differences are that you have replaced the `func` keyword with `var`, and have removed the parentheses indicating that this property is not a method. Otherwise, the computed type property is written just like any other computed property. For example, you still have to declare the computed type property as a `var`.

One new element in the code above is that you used the shorthand getter syntax. If you do not want to provide a setter for a computed property, then you can omit the `get` block of the computed property's definition and simply return the computed value as needed.

Switch to `main.swift`. Add a line at the bottom of the file to print the `Zombie` type's `spookyNoise` property to the console. You should see that `Brains...` was logged.

```
println("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
println("Victim pool: \(fredTheZombie.victimPool)")
println(Zombie.spookyNoise)
```

Access Control

The principle intuition behind access control is that you do not always want elements of your program's code to be visible to all other elements. In fact, you will frequently want to have much more granular control over your code's access. You can grant specific levels of access to various components of your code.

You can think of access control as defining the level of access other parts of a program have to your code. For example, you can hide or expose a method on a class depending upon what you want to share. Imagine that you have a property that is used only within a class's definition. It could be problematic if another, external type modified that property by mistake. With access control, you can manage the visibility of that property to hide it from other parts of the program. Doing so will encapsulate the property's data and prevent external code from meddling with it.

Access control is organized around two important and related concepts: modules and source files. In terms of your project's files and organization, these two are the central building blocks of your application.

A module is a unit of code that is distributed together. You may recall seeing `import UIKit` or `import Cocoa` at the top of your `Playgrounds`. These are frameworks, which bundle together a number related types that help to perform a series of related tasks. For example, `UIKit` is a framework designed to facilitate the development of user interfaces. Modules are brought into another module by using Swift's `import` keyword, as suggested by the above examples.

Source files, on the other hand, are more discrete units. They represent a single file and live within a specific module. You will typically define a single type within a source file. Doing so will help to keep your project organized; however, this practice is not a requirement. You can define multiple functions and types within a single file, though doing so is not a common organizational strategy.

Table 16.1 Swift Access Control

Access Level	Description
Public	Public access makes entities visible to all files in the module or those that import the module.
Internal	Internal access (the default) makes entities visible to all files in the same module.
Private	Private access makes entities visible only within their defining source file.

As you can probably infer from the table, `public` access is the least restrictive, and `private` access is the most restrictive level of access control. In general, a type's access level needs to be consistent with the access levels of its properties and methods. A property cannot have a less restrictive level of access control than its type. For example, a property with an access control level of `internal` cannot be declared on a type with `private` access. Likewise, the access control of a function cannot be less restrictive than the access control listed for its parameters. In both case, the thinking is that an entity cannot have less visibility than what is declared for its constituent parts.

Swift specifies `internal` as the default level of access control for your app. Having a default level of access means that you do not need to specifically declare access controls for every type, property, and method in your code. While this convenience is nice, there will be scenarios wherein you would like to manage access.

With this knowledge in hand, it is time to make some use of it. Create an `isFallingApart Boolean` property defined on the `Zombie` type. Give it a default value of `false`. As you can surmise, this property will keep track

of an instance's physical integrity. Accordingly, this property really does not need to be exposed to the rest of the program; it is an implementation detail of the `Zombie` class.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp = false
    private var isFallingApart = false

    override func terrorizeTown() {
        ~~town?.changePopulation(-10)~~
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }

    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

After you created the property, you also made use of it in the **`terrorizeTown()`** function. If the instance is falling apart, then it cannot terrorize its town. Thus, you check to see if `isFallingApart` is `false`. If it is `false`, then the instance is free to terrorize its town. If the instance is falling apart, then it will not be able to terrorize its town.

Conclusion

This chapter introduced a lot of material. Make sure to take a moment to let all of the ideas sink in. As a high-level overview, you learned about:

- Property syntax
- Stored vs. computed properties
- Read-only and read-write properties
- Lazy loading and lazy properties
- Property observers
- Type properties
- Access control

As you have seen throughout the book thus far, properties are truly a central concept in Swift programming. It is a good idea to get comfortable with the items listed above. The challenges described below will help you master the important concepts.

Challenges

Bronze Challenge

Your town's mayor is busy. Every birth and relocation does not require the mayor's attention. After all, the town is in crisis! Only log changes to the town's population if the new population is less than the old value.

Silver Challenge

Make a new type called `Mayor`. It should be a struct. The `Town` type should have a property called `mayor` that holds an instance of the `Mayor` type. Furthermore, the town should inform the `mayor` every time the property

for `population` changes. If the town's `population` decreases, then the instance of the `Mayor` should log this statement to the console: `"I'm deeply saddened to hear about this latest tragedy. I promise that my office is looking into the nature of this rash of violence."` If the population increases, then the `mayor` should do nothing. (HINT: You should define a new instance method on the `Mayor` type to complete this challenge.)

Gold Challenge

Mayors are people too. An instance of the `Mayor` type will naturally get nervous whenever its town loses some `population` due to a `Zombie` attack. Create a stored instance property on the `Mayor` type called `anxietyLevel`. It should be of type `Int`, and should also start out with a default value of 0. Increment the `anxietyLevel` property every time a `Mayor` instance is notified of a `Zombie` attack. Last, as a mayor will not want to outwardly display anxiety, mark this property as `private`. Verify that this property is not accessible in `main.swift`.

17
Initialization

Initialization is the operation of setting up an instance of a type. It entails giving each stored property an initial value, and may involve any other preparatory work. After this process, the instance is prepared and available to use.

The types that you have been creating up to this point have all been created in more or less the same way. The values for the properties were either given default stored values, or were computed on demand. Initialization was not customized, and it wasn't particularly considered.

It is very common to want control over how an instance of type is created. For example, it would be ideal for the instance to have all of the correct values in its properties right after you get an instance. Previously, you have given an instance default values to its stored properties, and then later changed these properties' values after you created an instance. This strategy is inelegant. *Initializers* help you create an instance with the appropriate values.

Initializer Syntax

Structures and classes are required to have a initial values for their stored properties by the time initialization completes. This requirement explains why you have been giving all of your stored properties default values. If you had not given these stored properties default values, then the compiler would have given you errors that the type's properties were not ready to use. Defining an initializer on the type ensures that properties have values when the instance is created.

The syntax for writing an initializer is a little different from what you have already seen. Initializers are written with the `init` keyword. Even though they are methods on a type, initializers are not preceded with the `func` keyword. The following informal example illustrates the initializer syntax.

```
struct CustomType {
    init() {
        // Initialization code here...
    }
}
```

In the example above, the initializer takes no arguments, as indicated by the empty parentheses. The initializer's implementation is defined within the braces, just as you have been doing with regular functions and methods throughout this book. Contrary to other methods, initializers do not return values. Instead, initializers are tasked with giving values to a type's stored properties. This general syntax does not differ between structures (or enumerations) and classes.

Struct Initialization

Default Initializers

Make a copy of the MonsterTown project, and put it somewhere convenient. When you are ready, open up the project and navigate to `main.swift`.

Remember how you were previously getting instances of your `Town` type? You did not create your own initializer, but did give the type's stored properties default values. Doing so meant that you could take advantage of a free "empty" initializer provided to you by the Swift compiler. Thus, all you needed to do was type `var`

`myTown = Town()`. This syntax calls the free, empty initializer and sets the new instance's properties to the default values you specified in the type's definition.

If you don't provide an initializer for your custom struct, then Swift will give you a "memberwise" initializer for free. This memberwise initializer will include arguments for all of the stored properties that need values. Remember, one of the principle goals of initialization is to give all of the type's storedp properties values such that the new instance is ready to use. The compiler will enforce the requirement that your new instance has values in its stored properties either through default values or memberwise initialization.

In `main.swift`, replace your use of the empty initializer on the `Town` type with a call to the free memberwise initializer. Leave the remainder of this file unchanged.

```
~~var myTown = Town()~~
var myTown = Town(population: 10000, numberOfStopLights: 6)
myTown.printTownDescription()
```

Click the play button in the upper lefthand corner to build and then run your program. Notice that your output is a little different? `myTown`'s description is different than the default values you gave to the `Town` type's stored properties. You should see `Population: 10000; number of stop lights: 6` logged to the console. How did these properties' values change from the default values you gave them?

As you can see above, the instance `myTown` was created with the free memberwise initializer. The `Town` type's stored properties are listed in the initializer, which allowed you to specify new values for the instance's properties. Afterward, printing the description of `myTown` to the console revealed that the default values for these properties had been replaced by the values you gave to the initializer.

You will also notice that the `Town`'s property names are used as external parameter names in the call to this initializer. Swift provides default external parameter names to every initializer automatically, one for each parameter given by the initializer. This convention is important because Swift's initializers all have the same name: `init`. Therefore, the function name cannot be used to identify which specific initializer should be called. The parameter names, and their types, help the compiler to differentiate between initializers so that it knows which initializer to call.

Custom Initializers

It is time to write your own initializer for the `Town` type. Note, if you write any of your own initializers for your type, then Swift will not give you any free initializers. The expectation is that if your are taking responsibility for your type's initialization, then you are also responsible for ensuring that instances' properties are all given their appropriate values.

You are going to remove all of default values for the properties. These were helpful before you knew about initializers, as they ensured that the properties for instances of your type had values when an instance was created. Now, however, they do not really add much value to the `Town` struct. Additionally, you will change `region` back to an instance property - the monster infestation is starting to spread outside the South.

Open the `Town.swift` file and remove the properties' default values.

```
struct Town {
    ~~static let region = "South"~~
    let region

    ~~var population: Int = 5422 {~~
    var population: Int {
        didSet(oldPopulation) {
            println("The population has changed to \(population) from \(oldPopulation).")
        }
    }
    ~~var numberOfStopLights = 4~~
    var numberOfStopLights

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }
...
```

After deleting the default values for these types, you may have noticed that the compiler issued an error to you: `Type annotation missing in pattern`. Previously, you took advantage type inference for these properties, which worked well with the default values you gave them. Without the default values, however, the compiler does not know what type information to give to the properties. Thus, the solution is to explicitly declare their type.

```
struct Town {
    ~~let region~~
    let region: String
    var population: Int {
        didSet(oldPopulation) {
            println("The population has changed to \(population) from \(oldPopulation).")
        }
    }
    ~~var numberOfStopLights~~
    var numberOfStopLights: Int

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }
...
```

Remember that the Swift compiler gives you a free memberwise initializer on structs that do not define a custom initializer. You are going to create this initializer by hand. Later, you will call this initializer from another initializer defined within this same type. For now, add the following initializer to your `Town` type.

```
...
var numberOfStopLights: Int
init(region: String, population: Int, stopLights: Int) {
    self.region = region
    self.population = population
    numberOfStopLights = stopLights
}
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
...
```

The **`init`** method written above takes three arguments, one for each of the stored properties on the `Town` type. One item of interest in the initializer above is that you passed the values given to the arguments of the initializer to the actual properties of the type. For example, the value passed to the `region` argument of the initializer was set as the value for the `region` property. Since the parameter name in the initializer is the same as the property name, you needed to explicitly access the property via `self`. `numberOfStopLights` does not have this problem, and so you were able to simply set the value of the initializer's argument for `stopLights` to the `numberOfStopLights` property.

Another item of interest is that you were able to set the value for the `region` property even though it was declared as a constant. You can do this because the Swift compiler allows you to modify the value of a constant property at any point during initialization. Remember, the goal of initialization is to ensure that a type's properties have values after initialization completes.

At this point, you may be noticing errors in `main.swift` and `Monster.swift` relating to the free initializers that the compiler was giving you by default. If you look closely at `main.swift`, you will notice that the memberwise initializer that the compiler gave you used the actual property name for `numberOfStopLights` for the argument name in the initializer. In the above initializer, you shortened this parameter name to `stopLights`. Switch to `main.swift` and change the parameter name, and add the `region` parameter:

```
~~var myTown = Town(population: 10000, numberOfStopLights: 6)~~
var myTown = Town(region: "West", population: 10000, stopLights: 6)
```

In `Monster.swift`, you were relying on the free empty initializer the compiler gave you because all of `Town`'s properties had default values. The free empty initializer is gone now that you are requiring `Town`s to be explicitly

initialized. It doesn't really make sense for a monster to create his own town anyway, so change `Monster` to start out with town to terrorize:

```
class Monster {
    var town: Town? = Town()
    var town: Town? = nil
...
```

Build and run the program. The error should disappear, and you should see the same console log output.

Initialization Delegation

You can define initializers to call other initializers on the same type. This procedure is called *initializer delegation*. It is typically used to provide multiple paths for creating an instance of a type.

In value types (i.e., enumerations and structures), initializer delegation is relatively straightforward. Since value types do not support inheritance, initializer delegation only involves calling another initializer defined on the type. It is somewhat more complicated for classes, as you will soon see.

Switch to `Town.swift` to write a new initializer on this type that makes use of initializer delegation. Add the following code:

```
init(region: String, population: Int, stopLights: Int) {
    self.region = region
    self.population = population
    numberOfStopLights = stopLights
}
init(population: Int, stopLights: Int) {
    self.init(region: "N/A", population: population, stopLights: stopLights)
}
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
```

In the code above, you defined a new initializer on the `Town` type. This initializer, however, is different from the previous one that you created. It only takes two arguments: 1) `population`, and 2) `stopLights`. What about the `region` property? How is that getting set?

Look at this new initializer's implementation. You are calling `Town`'s other initializer on self: `self.init(region: "N/A", population: population, stopLights: stopLights)`. Notice that you pass in the supplied arguments for `population` and `stopLights`. Since you did not have an argument for `region`, you had to supply your own value. In this case, you specified the string `"N/A"` to signify that there was no region information given to the initializer.

Initializer delegation helps to avoid duplication of code. Instead of retyping the same code to assign the values passed into the initializer's arguments to the type's properties, you can simply call across to another initializer on the type. Not only does duplicated code mean you type the same thing twice, but now if you need to change it you have to remember to change it both places (which introduces the possibility of bugs when you only update one). Initializer delegation also helps to define a path by which a type creates an instance.

Since you defined your own memberwise initializer, the compiler will give you no free initializers. This reality is not all that limiting. It can even be a benefit. For example, you might want to use this new initializer if there is no region information available for a given town that you would like to create. In these cases, you would use your handy new initializer with arguments for `population` and `stopLights` to set the corresponding properties while also giving `region` a placeholder value.

Make use of this new initializer in `main.swift`.

```
var myTown = Town(region: "West", population: 10000, stopLights: 6)
var myTown = Town(population: 10000, stopLights: 4)
myTown.printTownDescription()
```

If you build and run your application, the result is the same, but with one key difference. You are no longer setting the instance's `region` to anything specific.

Class Initialization

The general syntax for initialization in classes looks very similar to initialization in value types. Nonetheless, the are some different rules for classes that must be observed. These additional rules are mainly due to the fact that classes can inherit from other classes, which necessarily adds some complexity to initialization.

In particular, classes add the concepts of designated and convenience initializers. Designated initializers are responsible for making sure that an instance's properties all have values before initialization completes, thus making the instance ready to use. Convenience initializers are auxiliary to designated initializers; they supplement designated initializers by calling across a class to its designated initializer. The role of convenience initializers is typically to create an instance of a class for a very specific use case.

Default Initializers

You have already seen examples of using a class's default initializer. Classes get a default, empty initializer if you provide default values to all properties and do not write your own initializer. Classes do not get a free memberwise initializer like structs. This point explains why you gave your classes default values before; it allowed you to take advantage of the free empty initializer. Thus, you were able to get an instance of the `Zombie` like so: `let fredTheZombie = Zombie()`, with the parentheses indicating that you were using the default initializer.

Initialization and Class Inheritance

Open the `Monster.swift`. Modify the class to give it an initializer. Note, you are also removing the default value of `"Monster"` from the `name` property.

```
class Monster {
    var town: Town? = nil
    ~~let name = "Monster"~~
    let name: String

    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }
    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

This initializer has two arguments: one for an optional instance of the `Town` type, and another for the name of the monster. The values for these arguments are assigned to the class's properties within the initializer's implementation. Once again, note that the argument for the `town` in the initializer matched the property name on the class, so you have to set the property's value by accessing it through `self`. You were able to circumvent accessing `name` through `self` variable because the initializer's parameter had a different name.

Now that you have added this initializer, you may notice that there are two compiler errors in `main.swift`. Switch to this file to examine the errors.

You should see that your previous use of `Zombie()` to get an instance of this class is no longer satifying the compiler. Why not? The error states: `Missing argument for parameter 'town' in call`.

The error signifies that the compiler is expecting the `Zombie`'s initializer to include a parameter for `town`. Yet, this expectation is strange given that the you did not provide an initializer to the `Zombie` class that required

`town`. In fact, you provided no initializer to this class whatsoever; instead you have been relying upon the empty initializer the compiler gives you for free when your properties have default values.

`Zombie` no longer gets the free, empty initializer that you were making use of earlier. The reason for this changes involves understanding *automatic initializer inheritance*.

Automatic Initializer Inheritance

Classes do not typically inherit their superclass's initializers. This feature of Swift is intended to prevent subclasses from inadvertently providing initializers that will not set values on all the properties of the subclass type, since subclasses frequently add additional properties that do not exist in the superclass. Such behavior prevents types from being partially initialized with incomplete initializers.

Nonetheless, there are circumstances in which a class will automatically inherit its superclass's initializers. If your subclass has provided default values for all new properties it adds, then the are two scenarios in which that subclass will inherit its superclass's initializers.

1. **Scenario 1:** A subclass that does not define any designated initializers will inherit its superclass's designated initializers.

2. **Scenario 2:** A subclass that implements all of its superclass's designated initializers -- either explicitly or via scenario 1 above -- will inherit all of its superclass's convenience initializers.

It turns out that your `Zombie` type falls within the first of these two scenarios. It is inheriting the `Monster` type's sole designated initializer because it does not define its own designated initializer. The signature for this initializer is: `init(town:monsterName:)`. Since the `Zombie` type is inheriting an initializer, the compiler will no longer provide the free initializer that you were using before. Thus, from the compiler's point of view, the `Zombie` class does not have an empty initializer available to use.

There are two places where you need to update the `Zombie`'s initializer to remove the error. The first one is for `fredTheZombie`, and the second is from a previous chapter on comparing reference types via `z1` and `z2`. You do not need those instances anymore, so go ahead and delete them.

```
~~let fredTheZombie = Zombie()~~
let fredTheZombie = Zombie(town: myTown, monsterName: "Fred")
~~fredTheZombie.town = myTown~~
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
~~let z1 = Zombie()~~
~~z1.walksWithLimp = false~~
~~let z2 = z1~~
~~z2.walksWithLimp = true~~
~~println("z1 walks with limp? \(z1.walksWithLimp), z2 walks with limp? \(z2.walksWithLimp)")~~
```

Now, when you create an instance of the `Monster` or `Zombie` type, you give the instance a value for its `town` and `name` properties. Build and run the application; the errors should be gone, and the results are the same.

Designated Initializers

Classes use designated initializers as the primary initializer for a class. As part of this role, designated initializers are responsible for ensuring that the class's properties are all given values before initialization is ended. If a class has a superclass, then its designated initializers must also call its superclass's designated initializer.

You have already written a designated initializer for the `Monster` class. Recall:

```
init(town: Town?, monsterName: String) {
    self.town = town
    name = monsterName
}
```

Designated initializers are unadorned, meaning that designated initializers are denoted by no special keywords placed before `init`. This syntax distinguishes designtated initializers from convenience initializers, which use the keyword `convenience`. This initializer ensures that all of the properties on `Monster` are given values before initialization completes. Currently, the `Zombie` type gives default values to all of its properties (except for the ones inherited from `Monster`). Thus, the initializer you defined for `Monster` works fine for the `Zombie`.

Remove the default values for the `Zombie`'s properties.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    ~~var walksWithLimp = false~~
    var walksWithLimp: Bool
    ~~private var isFallingApart = false~~
    private var isFallingApart: Bool

    override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }
}
```

Removing these default values triggers a compiler error: `Class 'Zombie' has no initializers`. As the error message indicates, the `Zombie` class needs an initializer to give its properties values. It currently has no means by which all of its properties receive values before initialization completes.

Add a new initializer to the `Zombie` class to solve this problem.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp: Bool
    private var isFallingApart: Bool
    init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
        walksWithLimp = limp
        isFallingApart = fallingApart
        super.init(town: town, monsterName: monsterName)
    }
    override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }
}
```

Your new initializer above takes care of the error because you are now ensuring that the `Zombie`'s properties have values by the end of initialization. This task is accomplished in two ways. First, the new initializer sets the values of the `walksWithLimp` and `isFallingApart` properties via the `limp` and `fallingApart` arguments. These properties are specific to the `Zombie` class, and so the designated initializer needs to initialize them with appropriate values. Thus, the values of these parameters are assigned to the properties.

Second, you call the designated initializer of the `Monster` type's superclass. `super` points to a subclass's superclass. Thus, the syntax: `super.init(town: town, monsterName: monsterName)` passes the values of the parameters `town` and `monsterName` from the initializer on the `Zombie` class to the designated initializer on the `Monster` class. Doing so calls this initializer on `Monster`, which will ensure that the `Zombie`'s properties for `town` and `name` will be set.

Yet, there is still one more error to deal with. The `Zombie` class is not getting initialized correctly in `main.swift`. Switch to this file, and you will see that there is an error from the compiler telling you that the initializer for `Zombie` is missing an argument. Fix this by updating this old initializer to the new one.

```
~~let fredTheZombie = Zombie(town: myTown, monsterName: "Fred")~~
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
```

`fredTheZombie` is now getting initialized with all of the information that it needs in order to be ready for use.

Convenience Initializers

In contrast to designated initializers, convenience initializers are not responsible for making sure all of a class's properties have a value. Instead, they do the work that they are defined to do, and then hand off that information

to either another convenience initializer or a designated initializer. All convenience initializers call across to another initializer on the same class. Eventually, a convenience initializer must call through to its class's designated initializer. The relationships between convenience and designated initializers on a given class thereby defines a path by which a class's stored properties received initial values.

Make a convenience initializer on the Zombie type. This initializer will only provide arguments for whether or not the `Zombie` instance should have a limp, and whether or not the instance is falling apart.

```
init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
    walksWithLimp = limp
    isFallingApart = fallingApart
    super.init(town: town, monsterName: monsterName)
}
convenience init(limp: Bool, fallingApart: Bool) {
    self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "")
    if walksWithLimp {
        println("This zombie has a bad knee.")
    }
}
override func terrorizeTown() {
    if !isFallingApart {
        town?.changePopulation(-10)
    }
}
```

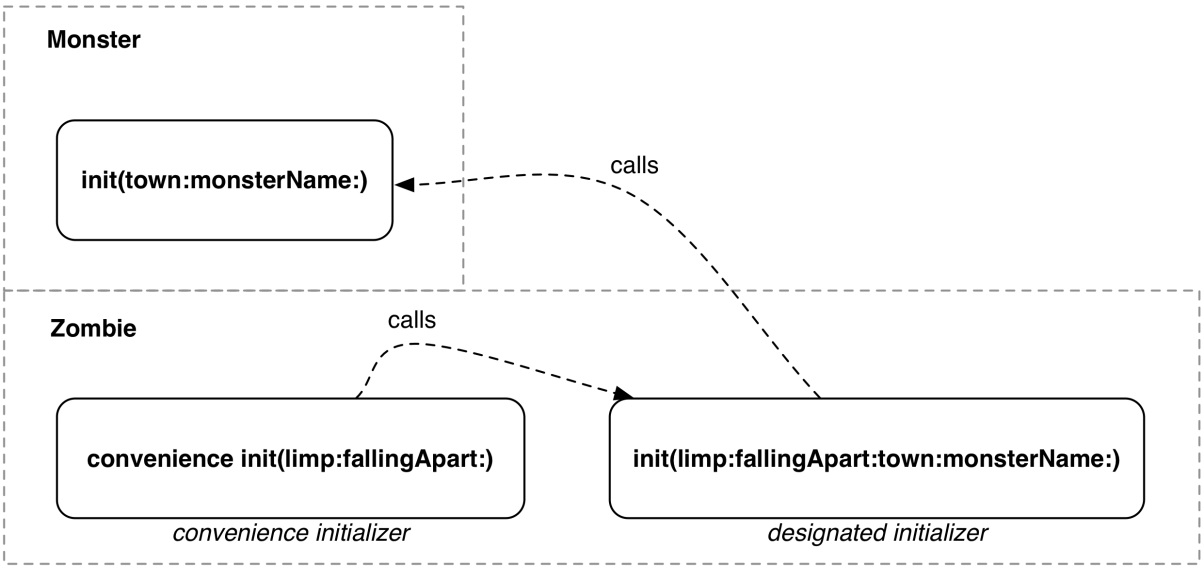
You mark an initializer as a convenience initializer with the `convenience` keyword. This keyword tells the compiler that the initializer will need to delegate to another initializer on the class, eventually calling to a designated initializer. After this call, an instance of the class is ready for use.

The convenience initializer above first calls the designated initializer on the `Zombie` class. It passes in the values for the parameters it received: `limp` and `fallingApart`. For the parameters that the convenience initializer did receive values for, `town` and `monsterName`, you passed `nil` and an empty string respectively.

The convenience initializer can consider the instance to be fully prepared for use after this point. Thus, you can check the value of the `walksWithLimp` property on the instance. If you had tried to do this check before calling across to the `Zombie`'s designated initializer, then the compiler would have issued an error: `Use of 'self' in delegating initializer before self.init is called`. This error is telling you that the delegating initializer is trying to use `self`, which is needed to access the `walksWithLimp` property, before it is ready for use.

Figure 17.1 graphically displays the relationships between the convenience and designated initializers discussed above.

Figure 17.1 Initializer Delegation



You can now create instances of the `Zombie` type with this convenience initializer. Switch to `main.swift`, and use it to create an instance. Remember, however, that instances of the `Zombie` created with this convenience initializer will have `nil` for the `town` property and an empty string (`""`) for the `name` property.

```
...
var fredTheZombie: Zombie? = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombie?.terrorizeTown()
fredTheZombie?.town?.printTownDescription()

var convenientZombie = Zombie(limp: false, fallingApart: false)
...
```

Required Initializers

A class can require subclasses to provide a specific initializer. For example, imagine that you want all subclasses of the `Monster` class to provide a basic initializer. To do so, you mark the initializer in question with the keyword `required` to indicate that all sublcasses of this type must provide the given initializer.

Switch to `Monster.swift` to make a required initializer.

```
class Monster {
    var town: Town? = nil
    let name: String

    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    ~~init(town: Town?, monsterName: String) {~~
    required init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }

    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

The sole designated initializer on the `Monster` class is now required. Subclasses must implement this initializer. This requirement makes sense given that two key components of a monster is its name and the town it haunts.

Unfortunately, the change triggers a compiler error: `'required' initializer 'init(town:monsterName:)' must be provided by subclass of 'Monster'`. The error is telling you that you are not yet implementing this newly required initializer on the `Zombie` class. Navigate to `Zombie.swift` to implement the intializer.

```
convenience init(limp: Bool, fallingApart: Bool) {
    self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "")
    if walksWithLimp {
        println("This zombie has a bad knee.")
    }
}
required init(town: Town?, monsterName: String) {
    walksWithLimp = false
    isFallingApart = false
    super.init(town: town, monsterName: monsterName)
}
override func terrorizeTown() {
    if !isFallingApart {
        town?.changePopulation(-10)
    }
}
```

To implement a superclass's required initializer, you must prefix the subclass's implementation of the initializer with the `required` keyword. Unlike other functions that you must override if you inherit them from your

superclass, you do not need to mark `required` initializers with the `override` keyword. It is implied by marking the intializer with `required`.

Your implementation of this `required` initializer makes it a designated initializer for the `Zombie` class. How did this initializer become a designated initializer?

Recall that designated initializers are responsible for initializing the type's properties and for delegating up to the superclass's initializer. This implementation does exactly those two things. You can therefore use this initializer to instantiate the `Zombie` class.

Deinitialization

Deinitialization refers to the process of removing instances of a class from memory when they are no longer needed. As a result, deinitialization is not available for use by value types. The details of memory management are covered in greater detail in Chapter 21.

In Swift, a *deinitializer* is called immediately prior to when the instance is removed from memory. It provides an opportunity to do any final maintenance before the instance is deallocated. Conceptually, it is the opposite of initialization.

Deinitializers are written with **`deinit`**.

```
...
required init(town: Town?, monsterName: String) {
    walksWithLimp = false
    isFallingApart = false
    super.init(town: town, monsterName: monsterName)
}

override func terrorizeTown() {
    if !isFallingApart {
        town?.isBeingTerrorized = true
        town?.changePopulation(-10)
    }
}
deinit {
    println("Zombie named \(name) is no longer with us.")
}
...
```

The new code above adds a deinitializer to the `Zombie` class. This deinitializer does not take any arguments, which is true for all deinitializers. Also, it is important to note that a class may only have one deinitializer.

Your new deinitializer simply logs a farewell to the `Zombie` instance that is about to be deallocated from memory. Notice that the deinitializer access's the `Zombie`'s name. Deinitializers have full access to a instance's properties and methods.

Open the `main.swift` to trigger the `Zombie`'s **`deinit`** method. You will set `fredTheZombie` to be `nil` at the end of the file. Doing so will incur the process of removing this instance from memory.

```
...
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
var fredTheZombie: Zombie? = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
fredTheZombie?.terrorizeTown()
fredTheZombie?.town?.printTownDescription()

println("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
println("Victim pool: \(fredTheZombie.victimPool)")

println("Victim pool: \(fredTheZombie?.victimPool)")
fredTheZombie?.victimPool = 500
println("Victim pool: \(fredTheZombie?.victimPool)")

println(Zombie.spookyNoise)
fredTheZombie = nil
```

Remember that only optional types can be or become `nil` in Swift. Therefore, you had to declare `fredTheZombie` as an optional: `Zombie?`. This change also meant that you now have to use optional chaining to unwrap the optional's value. You also needed to declare `fredTheZombie` with `var` instead of `let` because this instance can optionally change to become `nil`.

Build and run the program now. You will see that you now bid `fredTheZombie` farewell when the instance is deallocated.

Failable Initializers

Sometimes it is useful to define a type whose initialization can fail. In these cases, a fully prepared instance of the type with values for all of its properties is not the result of initialization; instead, you need a way to report to the caller that you were not able to initialize the instance. You use a *failable initializer* to handle these scenarios.

There are a number of reasons why initialization might fail. A type's initializer may be given invalid parameters. Perhaps a type's initialization depends upon an external resource that isn't available. For example, `let image = UIImage(named: "non-existing-image")` fails to create a `UIImage` instance because the image resource does not exist. When this happens, `UIImage`'s failable initializer will `return nil` to indicate that initialization has failed.

A Failable Town Initializer

Failable initializers return an optional instance of the type. You append a question mark after the keyword `init` to indicate that an initializer is failable (e.g., `init?`). You can also use an exclamation point after `init` to create a failable initializer that returns an implicitly unwrapped optional (e.g., `init!`).

Switch to `Town.swift` to give the `Town` struct a failable initializer. If an instance of `Town` is being created with a population of 0, then initialization will fail. This result makes sense because you cannot have a town without a population.

`Town` has two initializers. Remember that you delegated from the **`init(population:stopLights:)`** initializer to the **`init(region:population:stopLights:)`** initializer in this type. For now, just make the **`init(region:population:stopLights:)`** failable. Change its definition accordingly.

```
struct Town {
...
    ~~init(region: String, population: Int, stopLights: Int) {~~
        ~~self.region = region~~
        ~~self.population = population~~
        ~~numberOfStopLights = stopLights~~
    ~~}~~
    init?(region: String, population: Int, stopLights: Int) {
        if population <= 0 {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStopLights = stopLights
    }
...
}
```

Notice that you now use the failable initializer syntax: **`init?(region: String, population: Int, stopLights: Int)`**. After this declaration, you check to see if the given value for `population` is less than or equal to 0. If `population` is less than or equal to 0, then you `return nil`. The initializer fails in this case. It will create an optional instance of the `Town` type with the value of `nil`.

Open `main.swift` to see your new failable initializer in action. To do so, make sure that you initialize an instance of `Town` with a value of 0 for its `population` parameter.

```
~~var myTown = Town(population: 10000, stopLights: 4)~~
var myTown = Town(population: 0, stopLights: 4)
myTown.printTownDescription()
...
```

Take a moment to consider what is going on before you build and run the program.

The initializer **`init(population:stopLights:)`** currently delegates to a failable initializer. This process suggests that **`init(population:stopLights:)`** may get `nil` back from the designated initializer it delegates to. Receiving `nil` back from the designated initializer will be unexpected since **`init(population:stopLights:)`** is not failable itself.

Fix this problem by also making **`init(population:stopLights:)`** a failable initializer.

```
struct Town {
...
    init?(region: String, population: Int, stopLights: Int) {
        if population <= 0 {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStopLights = stopLights
    }

    ~~init(population: Int, stopLights: Int) {~~
        ~~self.init(region: "N/A", population: population, stopLights: stopLights)~~
    ~~}~~
    init?(population: Int, stopLights: Int) {
        self.init(region: "N/A", population: population, stopLights: stopLights)
    }
...
}
```

If you try to build the program now, then you will see that there are number of errors that you have to fix. You can find these errors in the `main.swift` file.

The line of code `myTown.printTownDescription()` has an error that reads: `'Town?' does not have a member named 'printTownDescription'`. Remember that changing the initializers on the `Town` struct to be failable means that they now return optionals. The initializers now return `Town?` and not `Town`. That means you have to unwrap the optionals before using them. Use optional chaining to fix the errors in `main.swift`.

```
var myTown = Town(population: 0, stopLights: 4)
~~myTown.printTownDescription()~~

~~let ts = myTown.townSize~~
~~println(ts.rawValue)~~

~~myTown?.changePopulation(1000000)~~
~~println("Size: \(myTown.townSize.rawValue); population: \(myTown.population)")~~
myTown?.printTownDescription()

let ts = myTown?.townSize
println(ts?.rawValue)

myTown?.changePopulation(1000000)
println("Size: \(myTown?.townSize.rawValue); population: \(myTown?.population)")
...
```

Representing `nil` in Swift tends to have a fairly extensive impact on your code. For example, using the failable initializer meant that you had to use optional chaining thoughout `main.swift`. These changes can add complexity and more code to your project. Both increase the chances of making a troublesome mistake. Accordingly, it is advisable to minimize your use of optionals to cases wherein you absolutely need them.

Build and run the program now. You removed the errors, and so the project runs fine.

Failable Initializers in Classes

Failable initializers work a bit differently in classes than in value types (like enumerations and structures). In value types, a failable initializer can fail at any point, and you would `return nil` at that time. Failable initializers in classes have to assign initial values in all of the class's properties before failing.

This requirement means that you cannot write the following code:

```
class MyClass {
    let myProperty: String
    init?(myProperty: String) {
        if myProperty.isEmpty {
            return nil
        }
        self.myProperty = myProperty
    }
}
```

Code like the above will trigger an error from the compiler telling you that all stored properties of a class must be initialized before returning nil. `myProperty` does not receive an initial value before the initializer fails. How can you avoid this error? One emerging pattern is to use an implicitly unwrapped optional.

```
class MyClass {
    ~~let myProperty: String~~
    let myProperty: String!
    init?(myProperty: String) {
        if myProperty.isEmpty {
            return nil
        }
        self.myProperty = myProperty
    }
}
```

The pseudo-code above makes `myProperty` an implicitly unwrapped optional. Note that the error has been removed. `myProperty`'s initial value is now `nil` because it is an implicitly unwrapped optional, which is a valid initial value. Therefore, the initializer can fail before giving `myProperty` a non-nil value.

Since `myProperty` is a constant, you will not be able to assign nil to it after initialization. For example, this code would trigger an error: `c?.myProperty = nil`. Thus, you can be sure that `myProperty` has a valid value if initialization succeeds. Moreover, the implicitly unwrapped optional syntax means that you can access `myProperty`'s value without having to use optional binding or chaining.

Nonetheless, the usual guidance about force unwrapping optionals is still important to keep in mind. If `MyClass` is initialized with a `String` instance that causes the initializer to fail (e.g., an empty `String`), then force unwrapping the resulting optional instance will cause a runtime crash. For example, this code is unsafe and should be avoided: `c!.myProperty`.

Conclusion

Initialization in Swift is a very defined process with a lot of rules. Thankfully, the compiler will remind you of what you need to do in order to comply and write a valid initializer. Rather than memorizing all of the rules to initialization, it is useful to think of Swift initialization in terms of value types and classes.

For value types, such as structs, initialization is principally responsible for ensuring that all of the instance's stored properties have been initialized and given appropriate values. This statement is true for classes as well, but initialization is a bit more complicated in this case.

Initialization for classes can be thought of as unfolding in two sequential phases.

In the first phase, a class's designated initializer is eventually called (either directly, or delegated to by a convenience initializer). At this point, all of the properties declared on this class are initialized with appropriate values inside of the designated initializer's definition. Next, a designated initializer delegates up to its superclass's designated initializer. The designated initializer on the superclass class then ensures that all of its own stored properties are initialized with appropriate values, which is a process that continues until the class at the top of the inheritance chain is reached. The first phase is now completed.

The second phase begins, and provides an opportunity for a class to further customize the values held by its stored properties. For example, a designated initializer can modify properties on `self` after it calls to the superclass's designated initializer. Designated initializers can also call instance methods on `self`. It is finally at this point that initialization reenters the convenience initializer, providing it with an opportunity to perform any customization on the instance that it would like to do.

The instance is fully initialized after these two phases, and all of its properties and methods are available for use.

The instinct behind this very definite initialization process is to guarantee the successful initialization of a class. The compiler secures this procedure, and will issue errors if any step along the process is not adhered to. In the end, it is not important that you remember each step in the process so long as you follow the compiler's guidance.

Challenges

Silver Challenge

Currently, the required initializer on `Monster` is implemented as a designated initializer on the `Zombie` subclass. Make this initializer a convenience initializer on the `Zombie` class. This change will involve delegating across the `Zombie` class to its designated initializer.

Gold Challenge

Currently, the `Monster` class can be initialized with any `String` instance for the `monsterName` parameter, even an empty `String`. Doing so would lead to an of instance of `Monster` with no name, and that does not make much sense. Fix this problem in the `Monster` class by ensuring that `monsterName` cannot be empty.

Your solution will involve giving `Monster` a failable initializer. Also note that this change will have an impact on initialization in the `Zombie` subclass. Make the necessary adjustments in this class as well.

For the More Curious

Initializer Parameters

Like functions and methods, initializers can provide explicit external parameter names. External parameter names distinguish between the parameter names available to callers, and the local parameter names used in the initializer's implementation. Since initializers follow different naming conventions than functions (i.e., initializer names are always **`init`**), the parameters names and types help to determine which initializer should be called. Thus, Swift provides external parameter names for all of the initializer's arguments by default.

You can provide your own external parameter names as needed. For example, imagine a `WeightRecordInLBS` struct that should be able to be initialized with kilograms.

```
struct WeightRecordInLBS {
    let weight: Double

    init(weightInKilos kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

The above initializer supplies `weightInKilos` as an explicit external parameter, and gives `kilos` as a local parameter. In its implementation, you simply convert `kilos` to pounds by multiplying with the correct conversion. You would then use this initializer like so: `let wr = WeightRecordInLBS(weightInKilos: 84)`.

You can even use _ as an explicit external parameter name if you do not want to expose a parameter name. For example, our fictitious `WeightRecordInLBS` struct obviously defines a weight record in terms of pounds. Thus, it would make sense for the initializer to default to taking pounds in its argument.

```
struct WeightRecordInLBS {
    let weight: Double

    init(_ pounds: Double) {
        weight = pounds
    }

    init(weightInKilos kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

The new initializer above can be used in the following manner: `let wr = WeightRecordInLBS(185)`. Since this type rather explicitly represents a weight record in pounds, there is no need for a named parameter in the argument list. Using an _ can make your code more concise, which is convenient when it is quite explicit what will be passed into the argument.

Part V

Advanced Swift

18
Protocols

In Chapter 16, you learned about using access controls to hide information. Hiding information is a form of encapsulation, which allows you to design your software in a way that allows you to change one part without affecting the rest of your program. Swift supports another form of encapsulation: a `protocol` allow you to specify and work with the *interface* of a type without knowing the type itself. An interface is a set of properties and a methods that a type provides.

Formatting a Table of Data

Protocols are a more abstract concept than many of the topics you have learned about so far. You will create a function that formats data into a table that looks like a simple spreadsheet. Next, you will use a protocol to make that function flexible enough to handle different *data sources*. Mac and iOS apps commonly separate the presentation of data from the source that provides the data. This separation is an extremely useful pattern which allows, for example, Apple to provide classes that handle presentation, while leaving it up to you to determine how data should be stored.

Create a new `Playground` called `Protocols.playground`. Begin with a function that takes an array whose individual elements are themselves arrays, an "array of arrays", and prints the numbers in a table. Each element of the `data` array is an array of integers that represent the columns of a single row, so the total number of rows is `data.count`.

```
func printTable(data: [[Int]]) {
    for row in data {
        // create an empty string
        var out = ""

        // append each item in this row to our string
        for item in row {
            out += " \(item) |"
        }

        // done - print it!
        println(out)
    }
}

let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]

printTable(data)
```

Open up the Timeline view, and you should see a simple table displaying the data. Add the ability to label each row. Labeling the rows is a little tricky because you want all the row labels to be aligned, so you will need to determine what the longest label is and pad the shorter rows with spaces:

```
~~func printTable(data: [[Int]]) {~~
func printTable(rowLabels: [String], data: [[Int]]) {
    // Determine length of longest row label
    let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })

    ~~for row in data {~~
        ~~// create an empty string~~
        ~~var out = ""~~
    for (i, row) in enumerate(data) {
        // Pad the row label out so they are all the same length
        let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])
        var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + " |"

        // append each item in this row to our string
        for item in row {
            out += " \(item) |"
        }

        // done - print it!
        println(out)
    }
}

let rowLabels = ["Joe", "Karen", "Fred"]
let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]

printTable(rowLabels, data)
```

You need to determine the width of the longest row label in order to align all the rows correctly. The `maxRowLabelWidth` instance stores this value; it is calculated by using `map` to convert each row label into an integer matching its length, and then passing that result to the `maxElement` function which returns the maximum (longest) length. Then, when you iterate over each row of the data, you have to insert some padding to rows that are shorter than the maximum width. The `paddingAmount` instance holds the amount of padding needed for a row, and `Repeat(count: paddingAmount, repeatedValue: " ")` creates a string containing `paddingAmount` spaces.

Finally, add the ability to label each column. Labeling columns is trickier still; you want all the columns to be lined up vertically, which may require padding all of the data items. You will have to keep track of the width of each column label, and pad all the items in that column out to that length.

```
~~func printTable(rowLabels: [String], data: [[Int]]) {~~
func printTable(rowLabels: [String], columnLabels: [String], data: [[Int]]) {
    // Determine length of longest row label
    let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })

    // Create first row containing column headers
    var firstRow: String = Repeat(count: maxRowLabelWidth, repeatedValue: " ") + " |"

    // Also keep track of the width of each column
    var columnWidths = [Int]()

    for columnLabel in columnLabels {
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(countElements(columnHeader))
    }
    println(firstRow)

    for (i, row) in enumerate(data) {
        // Pad the row label out so they are all the same length
        let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])
        var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + " |"

        // append each item in this row to our string
        ~~for item in row {~~
            ~~out += " \(item) |"~~
        for (j, item) in enumerate(row) {
            let itemString = " \(item) |"
            let paddingAmount = columnWidths[j] - countElements(itemString)
            out += Repeat(count: paddingAmount, repeatedValue: " ") + itemString
        }

        // done - print it!
        println(out)
    }
}

let rowLabels = ["Joe", "Karen", "Fred"]
let columnLabels = ["Age", "Years of Experience"]
let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]

~~printTable(rowLabels, data)~~
printTable(rowLabels, columnLabels, data)
```

You created and printed `firstRow`, which contains all of the column headers. It uses the same trick with `Repeat()` to add spaces over the first column (which contains the row headers). As you are constructing the first row, you also recorded the width of each column header in the `columnWidths` array. Later, when you append each data item to the output row, you use the `columnWidths` array and `Repeat()` to pad each item so that it is the same width as its column header.

Check your Timeline output again. You should see a nicely-formatted table of data:

```
      | Age | Years of Experience |
Joe   |  30 |                   6 |
Karen |  40 |                  18 |
Fred  |  50 |                  20 |
```

However, there is at least one major problem with the `printTable` function: it is very difficult to use! You have to have separate arrays for row and column labels and the data, and you have to manually make sure the number of row labels and column labels matches the number of elements in the data array. You are much more likely to want to represent information like this using structures and classes. Replace the part of the code where you call `printTable` with some *model objects*, which are types that represent the data your app works with:

```
~~let rowLabels = ["Joe", "Karen", "Fred"]~~
~~let columnLabels = ["Age", "Years of Experience"]~~
~~let data = [~~
~~    [30, 6],~~
~~    [40, 18],~~
~~    [50, 20],~~
~~]~~

~~printTable(rowLabels, columnLabels, data)~~
struct Person {
    let name: String
    let age: Int
    let yearsOfExperience: Int
}

struct Department {
    let name: String
    var people = [Person]()

    mutating func addPerson(person: Person) {
        people.append(person)
    }
}

var department = Department(name: "Engineering", people: [])
department.addPerson(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.addPerson(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.addPerson(Person(name: "Fred", age: 50, yearsOfExperience: 20))
```

You now have a `Department`, and you would like to be able to print out the details of its people using the `printTable` function. You could modify `printTable` to take a `Department` instead of the three arguments it takes now. However, the current implementation of `printTable` could be used to print any kind of tabular data, and it would be nice to keep that functionality. A protcol can help to preserve this functionality.

Protocols

A *protocol* allows you to define the interface you want a type to satisfy. A type that satisfies a protocol is said to *conform* to the protocol. Define a protocol that specifies the interface we need for the `printTable` function. The `printTable` function needs to know how many rows and columns there are, what the label for each row and column is, and what the item of data to display in each cell should be. It does not matter to the Swift compiler where in your Playground file you put this protocol, but it probably makes the most sense to put it at the top of the file, just before `printTable`.

```
protocol TabularDataSource {
    var numberOfRows: Int { get }
    var numberOfColumns: Int { get }

    func labelForRow(row: Int) -> String
    func labelForColumn(column: Int) -> String

    func itemForRow(row: Int, column: Int) -> Int
}
```

The syntax for a protocol should look familiar to you; it is very similar to defining a structure or a class, except that all the computed property and function definitions are omitted. The `TabularDataSource` protocol states that any conforming type must have two properties, `numberOfRows` and `numberOfColumns`. The syntax `{ get }` signifies that the property can be read, but not written. If the property were intended to be readable and writable, then you would use `{ get set }`. Note that a protocol property marked with `{ get }` does not exclude the possibility that a conforming type might have a property that is both readable and writable, only that the protocol only requires it to be readable. Additionally, `TabularDataSource` specifies that a conforming type must have the three methods listed with the exact types that are listed.

A protocol defines the minimum of properties and methods a type must have. The type can have more than what the protocol lists, but the protocol defines the baseline. Extra properties and methods are fine as long as all the requirements of the protocol are present.

You will make `Department` conform to the `TabularDataSource` protocol. Begin by declaring that `Department` conforms to `TabularDataSource`:

```
~~struct Department {~~
struct Department : TabularDataSource {
    let name: String
    var people = [Person]()

    mutating func addPerson(person: Person) {
        people.append(person)
    }
}
```

The syntax for conforming to a protocol is to add `: ProtocolName` after the name of the type. (This may look very familiar to how you declare a superclass. We will cover how protocols and superclasses can be used together later.) Your playground file now has an error, and if you open up the Timeline, you will see the details. You have claimed that `Department` conforms to `TabularDataSource`, but `Department` is missing all the properties and methods that `TabularDataSource` requires. Add implementations of them all:

```
struct Department : TabularDataSource {
    let name: String
    var people = [Person]()

    mutating func addPerson(person: Person) {
        people.append(person)
    }

    var numberOfRows: Int {
        return people.count
    }

    var numberOfColumns: Int {
        return 2
    }

    func labelForRow(row: Int) -> String {
        return people[row].name
    }

    func labelForColumn(column: Int) -> String {
        switch column {
        case 0: return "Age"
        case 1: return "Years of Experience"
        default: fatalError("Invalid column!")
        }
    }

    func itemForRow(row: Int, column: Int) -> Int {
        let person = people[row]
        switch column {
        case 0: return person.age
        case 1: return person.yearsOfExperience
        default: fatalError("Invalid column!")
        }
    }
}
```

A `Department` has a row for each person, so its `numberOfRows` property returns the number of people in the department. Each person has two properties that should be displayed, so `numberOfColumns` returns two. The label of each row is the name of the person to be shown on that row. `labelForColumn` and `itemForRow` are a little more interesting: you used the `switch` statement to return one of the two column headers. (Why is there a `default` case? Refer back to Chapter 6.)

You still need to go back and modify `printTable` to accept and work with a `TabularDataSource`. Protocols do not just define the properties and methods a conforming type must supply. They can also be used as types themselves: you can have variables, function arguments, and return values that have the type of a protocol. Change `printTable` to take a data source of type `TabularDataSource`, now that that protocol provides all the same data as the old arguments did (including all the column and row headers and the amount of data available):

```
~~func printTable(rowLabels: [String], columnLabels: [String], data: [[Int]]) {~~
func printTable(dataSource: TabularDataSource) {
    // Determine length of longest row label
    ~~let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })~~
    let maxRowLabelWidth = maxElement(map(0 ..< dataSource.numberOfRows, {
        countElements(dataSource.labelForRow($0))
    }))

    // Create first row containing column headers
    var firstRow: String = Repeat(count: maxRowLabelWidth, repeatedValue: " ") + " |"

    // Also keep track of the width of each column
    var columnWidths = [Int]()

    ~~for columnLabel in columnLabels {~~
    for c in 0 ..< dataSource.numberOfColumns {
        let columnLabel = dataSource.labelForColumn(c)
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(countElements(columnHeader))
    }
    println(firstRow)

    ~~for (i, row) in enumerate(data) {~~
    for r in 0 ..< dataSource.numberOfRows {
        // Pad the row label out so they are all the same length
        ~~let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])~~
        ~~var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + " |"~~
        let rowLabel = dataSource.labelForRow(r)
        let paddingAmount = maxRowLabelWidth - countElements(rowLabel)
        var out = rowLabel + Repeat(count: paddingAmount, repeatedValue: " ") + " |"

        // append each item in this row to our string
        ~~for (j, item) in enumerate(row) {~~
            ~~let itemString = " \(item) |"~~
            ~~let paddingAmount = columnWidths[j] - countElements(itemString)~~
        for c in 0 ..< dataSource.numberOfColumns {
            let item = dataSource.itemForRow(r, column: c)
            let itemString = " \(item) |"
            let paddingAmount = columnWidths[c] - countElements(itemString)
            out += Repeat(count: paddingAmount, repeatedValue: " ") + itemString
        }

        // done - print it!
        println(out)
    }
}
```

The `Department` type now conforms to `TabularDataSource`, and `printTable` has been modified to accept a `TabularDataSource`. Therefore, you can print your department:

```
var department = Department(name: "Engineering", people: [])
department.addPerson(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.addPerson(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.addPerson(Person(name: "Fred", age: 50, yearsOfExperience: 20))

printTable(department)
```

Protocol Conformance

As noted earlier, the syntax for protocol conformance looks exactly the same as the syntax you use to declare a class's superclass, as seen in Chapter 15. This brings up a few questions:

1. What types can conform to protocols?
2. Can a type conform to multiple protocols?
3. Can a class have a parent class and still conform to protocols?

All types can conform to protocols. You made a structure (`Department`) conform to a protocol; enums and classes can also conform to protocols. The syntax for declaring that an enum conforms to a protocol is exactly the same as it is for a struct: a colon and the protocol name follows the declaration of the type. (Classes can be a little more complicated; we will get to them momentarily).

It is also possible for a type to conform to multiple protocols. A protocol that is defined by Swift is `Printable`, which is used by functions like `println` to print instances. `Printable` has a single requirement: the type must have a gettable property named `description` that returns a `String`. Modify `Department` so that it conforms to both `TabularDataSource` and `Printable` using a comma to separate the protocols:

```
~~struct Department : TabularDataSource {~~
struct Department : TabularDataSource, Printable {
    let name: String
    var people = [Person]()

    var description: String {
        return "Department(\(name))"
    }

    // ... rest of Department stays the same
}
```

You implemented `description` as a read-only, computed property. You can now see the name of your department when you print it:

```
printTable(department)
println(department)
```

Finally, classes can also conform to protocols. If the class does not have a parent class, the syntax is the same as for structs and enums. For example,

```
class ClassName : ProtocolOne, ProtocolTwo {
    // ...
}
```

If the class does have a parent class, the name of the parent class comes first, and any protocols the class conforms to follow it; e.g.,

```
class ClassName : ParentClass, ProtocolOne, ProtocolTwo {
    // ...
}
```

Protocol Inheritance

Swift supports *protocol inheritance*. A protocol that inherits from another protocol requires conforming types to provide implementations for all the properties and methods required by both itself and the protocol it inherits from. This is different from class inheritance, which defines a close relationship between the superclass and subclass. Protocol inheritance merely adds any requirements from the parent protocol to the child protocol. For example, modify `TabularDataSource` so that it inherits from the `Printable` protocol:

```
~~protocol TabularDataSource {~~
protocol TabularDataSource : Printable {
    var numberOfRows: Int { get }
    var numberOfColumns: Int { get }

    func labelForRow(row: Int) -> String
    func labelForColumn(column: Int) -> String

    func itemForRow(row: Int, column: Int) -> Int
}
```

Now, any type that conforms to `TabularDataSource` must also conform to `Printable`, meaning it has to supply all the properties and methods listed in `TabularDataSource` as well as the `description` property required by `Printable`. Make use of this in `printTable` to print a heading on the table:

```
func printTable(dataSource: TabularDataSource) {
    println("Table: \(dataSource.description)")

    // ... rest of function remains unchanged
}
```

Protocols are allowed to inherit from multiple other protocols, just as types can conform to multiple protocols. The syntax for multiple protocol inheritance is what you probably expect: separate additional parent protocols with commas, like so:

```
protocol MyProtocol : MyOtherProtocol, Printable {
    // ... requirements of MyProtocol
}
```

Protocol Composition

Protocol inheritance is a powerful tool that lets you easily create a new protocol that adds requirements to an existing protocol or set of protocols. Nevertheless, using protocol inheritance can potentially lead you to make poor decisions in creating your types. In fact, that is exactly what has happened with `TabularDataSource`. You made `TabularDataSource` inherit from `Printable` because you wanted to be able to print a description of the data source, but there is not anything inherently `Printable` about a tabular data source. Go back and fix that misguided attempt to print data sources:

```
~~protocol TabularDataSource : Printable {~~
protocol TabularDataSource {
...
```

The compiler now rightfully complains when you try to get the `description` of the data source passed to `printTable()`. You can use *protocol composition* to state that the argument to `printTable` must conform to both `TabularDataSource` and `Printable`:

```
~~func printTable(dataSource: TabularDataSource) {~~
func printTable(dataSource: protocol<TabularDataSource, Printable>) {
    println("Table: \(dataSource.description)")

    // ... rest of function remains unchanged
}
```

The syntax for protocol composition uses the keyword `protocol` to signal to the compiler that you are combining multiple protocols into a single requirement. You can use protocol composition with more than two protocols by adding additional protocols, separated by commas, inside the <> brackets. The example above requires that `dataSource` is both a `TabularDataSource` and is also `Printable`: `dataSource: protocol<TabularDataSource, Printable>`.

Consider another possibility. You could create a new protocol that inherits from both `TabularDataSource` and `Printable`, like so:

```
protocol PrintableTabularDataSource: TabularDataSource, Printable {
}
```

You could then use that protocol as the type of the argument to `printTable()`. Both `PrintableTabularDataSource` and `protocol<TabularDataSource, Printable>` require conforming types to implement all properties and methods required by `TabularDataSource` and `Printable`. What is the difference between them?

The difference is that `PrintableTabularDataSource` is a distinct type: you would have to go back and modify `Department` to state that it conforms to `PrintableTabularDataSource` even though it already fulfills all the requirements. On the other hand, you can think of `protocol<TabularDataSource, Printable>` as a "temporary" protocol type. You did not have to go back and annotate `Department`; it already conforms to both `TabularDataSource` and `Printable`, so it also conforms to `protocol<TabularDataSource, Printable>`.

Mutating Methods

Recall from Chapter 14 and Chapter 15 that methods on value types, structs and enums, cannot modify self unless the method is marked as `mutating`. Methods in protocols default to non-mutating. Recall the `LightBulb` type from Chapter 14:

```
enum LightBulb {
    case On
    case Off

    mutating func toggle() {
        switch self {
        case .On:
            self = .Off

        case .Off:
            self = .On
        }
    }
}
```

Suppose you want to define a protocol that an instance is "toggleable":

```
protocol Toggleable {
    func toggle()
}
```

Declaring that `LightBulb` conforms to `Toggleable` will result in a compiler error. The message you get includes a note that explains the problem:

```
error: type 'LightBulb' does not conform to protocol 'Toggleable'

note: candidate is marked 'mutating' but protocol does not allow it
mutating func toggle() {
              ^
```

The note points out that in `LightBulb`, the `toggle` method is marked as mutating, but the `Toggleable` protocol expects a non-mutating function. You can fix this problem by marking `toggle` as mutating in the protocol definition:

```
protocol Toggleable {
    mutating func toggle()
}
```

A class that conformed to the `Toggleable` protocol would not need to mark its `toggle` method as mutating. Methods on classes are always allowed to change properties of `self` because they are reference types.

Conclusion

You have learned quite a bit about Swift protocols on this chapter:

- How to declare a protocol with property and method requirements
- How to modify an existing type to conform to a protocol
- Using protocol inheritance to create a new protocol that adds additional requirements to existing protocols
- When to use `mutating` on a method in a protocol

Additionally, you have now had an introduction to the pattern of a *data source* protocol, which is a very common pattern in iOS and Mac OS X development. There is more to the story - protocols have some additional powerful features related to Swift generics, the subject of Chapter 20.

Challenges

Silver Challenge

The `printTable` function has a bug - it crashes if any of the data items are longer than the label of the column they are in. Try changing Joe's age to 1000 to see this in action. Fix the bug. (For the easier version of this challenge, just make the function not crash. For the harder version of this challenge, make sure all the rows and columns of the table are still aligned correctly.)

Gold Challenge

Create a new type, `BookCollection`, that conforms to `TabularDataSource`. Calling `printTable` on a book collection should show a table of books, with columns for the book's title, author, and average review on Amazon. (Unless all the books you use have very short titles and author names, you will need to have completed the previous challenge!)

19
Extensions

Imagine that you are working with a type in the Swift standard library, say the `Double` type. For whatever reason, your application uses this type frequently, and it would be great if the `Double` type supported some additional methods. Unfortunately, you do not have `Double`'s implementation available, and so you cannot add that functionality directly to it yourself. What can you do?

Swift provides a feature called *Extensions* that designed for just these cases. Extensions allow you to add functionality to an existing type. You can extend structs, enums, and classes. Extensions can add computed properties, methods, protocol conformance, and a number of other features to an existing type.

In this chapter you will use extensions to add functionality to an existing type whose definition and implementation details are not available to you. You will also use extensions to add functionality to a custom type of your own creation. In both cases you will add functionality to these types in a modular fashion, meaning that you will group like functionality in a single extension.

Extending an Existing Type

Create a new playground named `Extensions.playground`. Save it where you like and open it.

You will be modeling the behavior of a car in this playground. Velocity is an important characteristic of any vehicle. Since velocity can have decimal values, it is reasonable to represent it as a `Double`.

Given that you will be using the `Double` type frequently, it would be useful to refer to it in a way that is contextually relevant. Swift's `typealias` keyword provides a way to give another name to an existing type. Give the `Double` type alternate name in the playground.

```
typealias Velocity = Double
```

The `typealias` keyword allows you to define `Velocity` as an alternative type name for `Double`. `Velocity` is a good name for `Double` in this example since you will be using this type to model a vehicle's speed. You can now refer to `Double` with the `Velocity` type name. This interchangeability helps to contextualize `Double` by making it more relevant to your use.

Now that you have a `typealias` set up, it is time to extend the type to support conversion between commonly used units for speed. Add the following to your playground.

```
typealias Velocity = Double
extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
```

The `extension` keyword signifies that you are extending the `Velocity` type. Swift's extensions do not allow you to add stored properties to a type. In this case you are adding two computed properties to `Velocity`: `kph` and `mph`. These properties on `Velocity` convert a vehicle's speed from kilometers per hour (`kph`) to miles per hour (`mph`), and vice versa. Note, that the extension treates `mph` as the default unit; that computed property simply returns `self`, while `kph` performs the conversion.

While the interchangeability provided by a `typealias` can be a benefit, it can also be a bit tricky. You may encounter a case in this file where you will want to use the `Double` type and not the `Velocity` type. Since

`Velocity` can be used interchangeably with `Double`, the extension you defined on `Velocity` is also available for the `Double` type. This reality can be confusing, but adding the extension to `Double` using the `Velocity` `typealias` gives helpful context. It documents that the computed properties are only meaningful when used with the `Velocity` typealias, even though they are available to all `Doubles`.

Recall that one of the goals of this chapter is to define the behavior of a vehicle. Protocols are one of Swift's most helpful features for helping to define the interace for a type. As you will see, you can add protocol conformance to a type with an extension.

Create a new protocol called `VehicleType` to describe some of the basic characteristics of a vehicle.

```
typealias Velocity = Double

extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
protocol VehicleType {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}
```

`VehicleType` declares three properties: 1) `topSpeed`, 2) `numberOfDoors`, and 3) `hasFlatbed`. Each property only requires the conforming type to implement a "getter" for the property. A type conforming to this protocol will be required to provide these properties that describe some general characteristics of a vehicle.

Extending Your Own Type

You will need to create a new type before you can add protocol conformance to it through an extension. Make a new struct to represent a `Car` type. You will later use an extension on `Car` to add conformance to the `VehicleType` protocol.

```
typealias Velocity = Double

extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}

protocol VehicleType {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}
struct Car {
    let make: String
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue >= 0.0, "New value must be between 0 and 1.")
        }
    }
}
```

You defined above a new struct called `Car`. The `Car` type defined a number of stored properties that will be specific to a given instance. All of the properties are constants, with one exception: `gasLevel`.

`gasLevel` is a mutable stored property with a property observer on it. The `willSet` observer will be called very time you are going to set a new value for `gasLevel`. You used an `precondition()` inside of this implementation

to ensure that the `newValue` being assigned to the `gasLevel` property is between 0 and 1. These values indicate how full an instance's gas tank is in terms of percentage points.

Use Extensions to Add Protocol Conformance

Extensions can provide a great mechanism to group related chunks of functionality together. Grouping related pieces of functionality together in a single extension can help to make your code more readable and maintainable. This pattern also helps a type to keep its interface uncluttered.

Extend the `Car` type to conform to the `VehicleType` protocol.

```
...
struct Car {
    let make: String
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue > 0.0, "New value must be between 0 and 1.")
        }
    }
}
extension Car: VehicleType {
    var topSpeed: Velocity { return 180 }
    var numberOfDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
```

Your new extension above extends `Car` to conform to `VehicleType`. The syntax for conforming to a protocol is the same as you have seen before. What is different is that you use an extension to accomplish this protocol conformance: `extension Car: VehicleType`.

You implement the protocol's required properties inside of the extension's body. Each property was given a simple "getter". For the sake of convenience, you simply returned some default values for each of the protocol's properties.

Adding an Initializer with an Extension

Recall that structs give you a free memberwise initializer if you do not provide your own. If you want to write a new initializer for your struct, but do not want to lose the memberwise initializer, then add the initializer to a given type with an extension. Add an initializer to `Car` in a new extension on the type.

```
...
extension Car: VehicleType {
    var topSpeed: Velocity { return 180 }
    var numberOfDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
extension Car {
    init(carMake: String, carModel: String, carYear: Int) {
        self.init(make: carMake,
            model: carModel,
            year: carYear,
            color: "Black",
            nickname: "N/A",
            gasLevel: 1.0)
    }
}
```

The new extension on the `Car` type adds an initializer that accepts arguments only for an instance's `make`, `model`, and `year`. This new initializer utilizes the free memberwise initializer on the `Car` struct. Your new initializer's arguments are passed into the memberwise initializer, and you also provided default values for the missing

arguments. The combiniation of these two initializers ensures that an instance of the `Car` type will values for all of its properties.

As you can see, the memberwise initializer is preserved on `Car` because this new initializer is actually defined and implemented on an extension. This pattern can be quite helpful.

To see the new initializer defined in the extension work, go ahead and create an instance of `Car`.

```
...
extension Car {
    init(carMake: String, carModel: String, carYear: Int) {
        self.init(make: carMake,
            model: carModel,
            year: carYear,
            color: "Black",
            nickname: "N/A",
            gasLevel: 1.0)
    }
}
var c = Car(carMake: "Ford", carModel: "Fusion", carYear: 2013)
```

The code above creates a new instance `c`. This instance is created with the initializer defined in an extension on `Car`. Take a look in the results sidebar. You should see that `c`'s properties have the values you gave to this new initializer; the default values you gave to the memberwise initializer should also be visible in the sidebar as well.

Nested Types and Extensions

Swift's extensions can also add nested types to an existing type. Say, for example, that you want to add an enumeration to your `Car` struct to help classify the sort of car an instance might be. Create a new extension on the `Car` type to add a nested type.

```
...
var c = Car(carMake: "Ford", carModel: "Fusion", carYear: 2013)
extension Car {
    enum CarKind: Printable {
        case Coupe, Sedan
        var description: String {
            switch self {
            case .Coupe:
                return "Coupe"

            case .Sedan:
                return "Sedan"
            }
        }
    }
    var kind: CarKind {
        if numberOfDoors == 2 {
            return .Coupe
        } else {
            return .Sedan
        }
    }
}
```

The extension on `Car` adds a nested type called `CarKind`. `CarKind` is an enumeration that has two cases: one for a `Coupe`, and one for a `Sedan`. The extension also adds a computed property on `Car` called `kind`. This property will represent what kind of car an instance of `Car` is. The nested type also conforms to the `Printable` protocol to facilitate logging to the console.

`kind` returns values of the nested enumeration based upon how many doors the instance has. If the instance has two doors, then it is a coupe. If it has more, then it is a sedan.

Exercise the new nested type in the extension by accessing the computed property `kind` on the instance you created before.

```
...
extension Car {
    enum CarKind: Printable {
        case Coupe, Sedan
        var description: String {
            switch self {
            case .Coupe:
                return "Coupe"

            case .Sedan:
                return "Sedan"
            }
        }
    }
    var kind: CarKind {
        if numberOfDoors == 2 {
            return .Coupe
        } else {
            return .Sedan
        }
    }
}
c.kind.description
```

You should see `"Sedan"` logged to the console.

Extensions with Functions

You can use an extension to give an existing type a function. For example, you may have noticed that the `Car` doesn't have a function to fill its gas. Make an extension to add this functionality to `Car`.

```
...
c.kind.description
extension Car {
    mutating func emptyGas(amount: Double) {
        precondition(amount <= 1 && amount > 0, "Amount to remove must be between 0 and 1.")
        gasLevel -= amount
    }

    mutating func fillGas() {
        gasLevel = 1.0
    }
}
```

Your new extension adds two functions, **`emptyGas()`** and **`fillGas()`**, to the `Car`. Note that both functions are marked with the `mutating` keyword. Why? Remember that the `Car` type is a struct. If a function wants to change the value of any of the struct's properties, then it must be declared with the `mutating` keyword.

The **`emptyGas()`** function takes one argument: the amount of gas to remove from the tank. You used an **`precondition`** inside of the **`emptyGas()`** to ensure that the amount removed from the tank is between 0 and 1. The implementation of the **`fillGas()`** simply sets the `gasLevel` property on the `Car` to be full, or **`1.0`**.

Exercise these new functions on your existing type.

```
extension Car {
    mutating func emptyGas(amount: Double) {
        precondition(amount <= 1 && amount > 0, "Amount to remove must be between 0 and 1.")
        gasLevel -= amount
    }

    mutating func fillGas() {
        gasLevel = 1.0
    }
}
c.emptyGas(0.3)
c.gasLevel
c.fillGas()
c.gasLevel
```

After you use the **`emptyGas()`** function, you should see that the gas level is `0.7` in the sidebar. After you fill the gas level, you see that the gas level is now `1.0`.

Conclusion

Extensions allow you to add functionality to an existing type. You can use them to add functionality to a type that you did not create. This feature is especially helpful when you do not have the source file for that type. You can also add functionality to a type that you created yourself. Extending your own types in this way allows you to group related functionality together in a single extension.

In general, you can use extensions to extend types with:

- Computed properties
- New initializers
- Protocol conformance
- New methods
- Embedded types

You saw each of these in the chapter.

Challenges

Bronze Challenge

Extend the `Int` type to have a `timesFive` computed property. The computed property should return the result of multiplying the integer by 5. You should be able to use it like so:

```
5.timesFives // 25
```

Silver Challenge

The **`emptyGas()`** method has some bugs. For example, if the current `gasLevel` is less than the amount to remove, then the new value for this property will be negative. A negative value does not make sense, and will actually stop the program from running (recall the `precondition` in `gasLevel`'s property observer). Revise **`emptyGas`**'s implementation to ensure that `gasLevel` is not decremented to be a negative value.

20
Generics

So far, all the types you have created and all the functions you have written have worked on concrete types like `Int`, `String`, and `Monster`. You may have noticed, however, that Swift allows you to create arrays that contain any given type, including `Int`, `String`, and `Monster`. How is array implemented? How can you write code that can work with a variety of types in the same way? The answer to both of these questions is "generics".

Swift generics allow you to write types and functions that make use of types that are not yet known to you or the compiler. Many of the built-in types you have used throughout this book, including Optionals, Arrays, and Dictionaries, are implemented using generics. In this chapter, you will investigate how to write generic types (much like Array), and additionally, how you can use generics to write flexible functions, as well as how generics are related to protocols.

Generic Data Structures

You will create a generic *stack*, which is a venerable data structure in computer science. A stack is a last-in, first-out (LIFO) data structure. It supports two basic operations. You can *push* an item onto the stack, which adds the item to the stack, and you can *pop* to get the most-recently-pushed item off of the stack, which removes the item from the stack.

To begin, create a new playground called `Generics.playground`, and make a `Stack` structure that only stores integers:

```
struct Stack {
    var items = [Int]()

    mutating func push(newItem: Int) {
        items.append(newItem)
    }

    mutating func pop() -> Int? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }
}
```

This struct has three elements of interest. The `items` stored property is an array which you are using to hold on to the items currently in a stack. The `push()` method pushes a new item onto the stack by appending it to the end of the `items` array. Finally, the `pop()` method pops the top item off of the stack by calling the `removeLast()` method of an array, which simultaneously removes the last item and returns it. Note that `pop()` returns an optional `Int` because the stack might be empty (in which case there is nothing to pop).

Create a `Stack` instance to see it in action:

```
var intStack = Stack()
intStack.push(1)
intStack.push(2)

println(intStack.pop()) // prints Optional(2)
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil
```

You created a new `Stack` instance, pushed two values on, then tried to pop three values off. As expected, the pop calls return the integers you pushed in reverse order, and pop returns nil when the stack no longer has any items left.

Now, modify `Stack` to be a generic data structure that can hold any type, not just `Int`:

```
~~struct Stack {~~
struct Stack<T> {
    ~~var items = [Int]()~~
    var items = [T]()

    ~~mutating func push(newItem: Int) {~~
    mutating func push(newItem: T) {
        items.append(newItem)
    }

    ~~mutating func pop() -> Int? {~~
    mutating func pop() -> T? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }
}
```

You added a *placeholder type*, named `T`, to the declaration of `Stack`. Swift's syntax for declaring a generic uses angle brackets (`<>`) and immediately follows the name of the type. The letter between the angle brackets represents the placeholder type: `<T>`. The use of the letter `T` is not required; however, using single uppercase letters (like `T` and `U`) is a common convention. The placeholder type `T` can be used inside the `Stack` structure anywhere a concrete type could be used. You can see this usage as you replaced all the occurrences of `Int` with `T`, including in a property declaration, the type of the argument in `push()`, and the type of the return value of `pop()`.

There is now a compiler error where you create a `Stack` because you have not specified what type should be substituted for the placeholder type `T`. The process of the compiler substituting a concrete type for a placeholder is called *specialization*. Fix the error by specifying that `intStack` should be an instance of `Stack` specialized for `Int` using the same angle bracket syntax:

```
~~var intStack = Stack()~~
var intStack = Stack<Int>()
```

You can now create `Stacks` of any kind of type. Create a `Stack` of `Strings`:

```
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil

var stringStack = Stack<String>()
stringStack.push("this is a string")
stringStack.push("another string")

println(stringStack.pop()) // prints Optional("another string")
```

It is important to note that even though `intStack` and `stringStack` are both `Stack` instances, they do not have the same type. `intStack` is a `Stack<Int>`; it would be a compile-time error to pass anything other than an `Int` to `intStack.push()`. Likewise, `stringStack` is a `Stack<String>`, which is distinct from `Stack<Int>`.

Generic data structures are both common and extremely useful. Classes and enumerations can also be made generic using the same syntax as you used here for structures. In addition, types are not the only element of Swift that can be generic; functions and methods can also be generic.

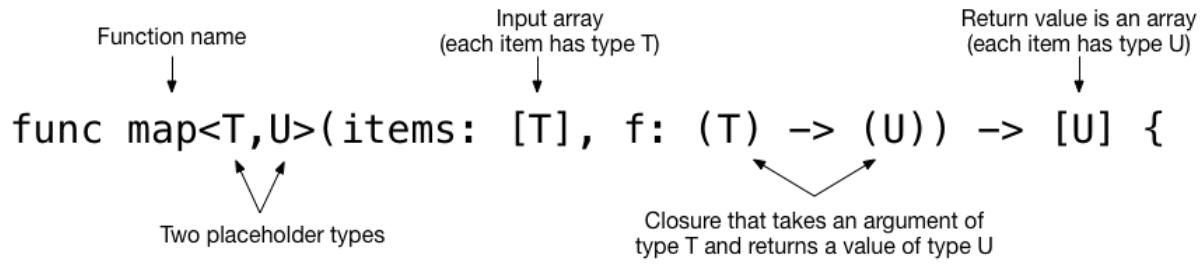
Generic Functions and Methods

Recall the `map` function from Chapter 13. `map` takes a sequence of items and applies a closure to each element in the sequence, and returns an array of the results. Given what you just learned about generics, you can now implement this function yourself. Add the following code to your playground:

```
func myMap<T,U>(items: [T], f: (T) -> (U)) -> [U] {
    var result = [U]()
    for item in items {
        result.append(f(item))
    }
    return result
}
```

The declaration of `myMap` may look pretty ugly if you have not been exposed to generics in other languages. The only thing that is new is that it declares two placeholder types, `T` and `U`, not just one. Here is a breakdown of that line:

Figure 20.1 myMap Declaration



`myMap` can be used the same way `map` is used. Create an array of strings, then map it to an array of their lengths:

```
func myMap<T,U>(items: [T], f: (T) -> (U)) -> [U] {
    // ... unchanged
}

let strings = ["one", "two", "three"]
let stringLengths = myMap(strings, { countElements($0) })
println(stringLengths) // prints [3, 3, 5]
```

The closure passed to `myMap` must take a single argument that matches the type contained in the `items` array, but the type of its return value can be anything at all. In this call to `myMap`, `T` is replaced by `String` and `U` is replaced by `Int`. Note that in real projects there is no need to declare your own mapping function - just use the built-in `map`.

Methods can also be generic, even inside of types that are already themselves generic. The `myMap` function you wrote only works on arrays, but it seems reasonable to want to map a `Stack`. Create a `map` method on `Stack`:

```
struct Stack<T> {
    var items = [T]()

    mutating func push(newItem: T) {
        items.append(newItem)
    }

    mutating func pop() -> T? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }

    func map<U>(f: (T) -> (U)) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }
}
```

The `map` method only declares one placeholder type, `U`, but it uses both `T` and `U`. `T` is available because `map` is inside of the `Stack` structure, which makes the placeholder type `T` available. The body of `map` is almost identical to `myMap`, differing only in that it returns a new `Stack` instead of an array. Try out your new method:

```
var intStack = Stack<Int>()
intStack.push(1)
intStack.push(2)
var doubledStack = intStack.map({ 2 * $0 })

println(intStack.pop()) // prints Optional(2)
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil

println(doubledStack.pop()) // prints Optional(4)
println(doubledStack.pop()) // prints Optional(2)
```

Type Constraints

One of the most important things to keep in mind when writing generic functions and data types is that, by default, you do not know anything about the concrete type that is going to be used. You created stacks of `Int` and `String`, but you could also create stacks of any other type at all. The practical impact of this lack of knowledge is that there is very little you can do with a value of a placeholder type. For example, you cannot even check if two of them are equal; this code would not compile:

```
func checkIfEqual<T>(first: T, second: T) -> Bool {
    return first == second
}
```

This function could be called with any type at all, including types for which equality does not make sense, such as closures. (It is hard to describe what it would mean for two closures to be "equal". Swift does not allow the comparison.) Generic functions would be pretty uncommon if you were never able to assume anything about the placeholder types. To solve this problem, Swift allows the use of *type constraints*, which place restrictions on the concrete types that can be passed to generic functions. There are two kinds of type constraints: a constraint that a type be a subclass of a given class, or a constraint that a type conforms to a protocol (or a protocol composition). The `Equatable` protocol is a Swift-provided protocol that states that two values can be checked for equality. You can write `checkIfEqual` by including a constraint that `T` must be `Equatable`:

```
func checkIfEqual<T: Equatable>(first: T, second: T) -> Bool {
    return first == second
}

println(checkIfEqual(1, 1))
println(checkIfEqual("a string", "a string"))
println(checkIfEqual("a string", "a different string"))
```

Every placeholder type can have a type constraint. For example, write a function that checks if two `Printable` values have the same description:

```
func checkIfDescriptionsMatch<T: Printable, U: Printable>(first: T, second: U) -> Bool {
    return first.description == second.description
}

println(checkIfDescriptionsMatch(Int(1), UInt(1)))
println(checkIfDescriptionsMatch(1, 1.0))
println(checkIfDescriptionsMatch(Float(1.0), Double(1.0)))
```

The constraint that both `T` and `U` are `Printable` guarantees that both `first` and `second` have a property named `description` that returns a `String`. Even though the two arguments may have different types, you can still compare their descriptions.

Associated Type Protocols

Now that you know that types, functions, and methods can all be made generic, it is natural to ask if protocols can be made generic. The answer is no; however, protocols support a similar and related feature: *associated types*. You will explore protocols with associated types by examining a couple of protocols defined by the Swift standard library. First, the `GeneratorType` protocol:

```
protocol GeneratorType {
    typealias Element

    mutating func next() -> Element?
}
```

The `GeneratorType` protocol requires a single mutating method, `next()`, which returns a value of type `Element?`. The purpose of `GeneratorType` is that you can call `next` repeatedly, and it generates new values each time. If the generator is no longer able to generate new values, `next` returns nil. The new syntax present present in this protocol is `typealias Element`. This syntax indicates that a type that conforms to `GeneratorType` must provide an associated, concrete type to be used as the `Element` type. At the top of your playground, create a new a new struct called `StackGenerator` that conforms to `GeneratorType`:

```
struct StackGenerator<T> : GeneratorType {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element? {
        return stack.pop()
    }
}

struct Stack<T> {
    // ...
}
```

`StackGenerator` wraps up a `Stack`, and generates values by popping items off of the stack. The type of the `Element` that `next` returns is `T`, so you set the typealias appropriately. Create a new stack, add some items, then create a generator and loop over its values to see `StackGenerator` in action:

```
var myStack = Stack<Int>()
myStack.push(10)
myStack.push(20)
myStack.push(30)

var myStackGenerator = StackGenerator(stack: myStack)
while let value = myStackGenerator.next() {
    println("got \(value)")
}
```

`StackGenerator` is a little more verbose than it needs to be. Swift can infer the type of a protocol's associated types, so you can remove the explicit typealias by indicating that `next` returns a `T?`:

```
struct StackGenerator<T> : GeneratorType {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element? {
    mutating func next() -> T? {
        return stack.pop()
    }
}
```

The next associated type protocol you will examine is `SequenceType`. The definition of `SequenceType` is:

```
protocol SequenceType {
    typealias Generator : GeneratorType
    func generate() -> Generator
}
```

`SequenceType` has an associated type named `Generator`. The `: GeneratorType` syntax is a type constraint on the associated type. It has the same meaning as type constraints on generics: for a type to conform to `SequenceType`, it must have an associated type `Generator` which conforms to the protocol `GeneratorType`. `SequenceType` also requires conforming types to implement a single method, `generate()`, which returns a value of the associated type `GeneratorType`. Since you already have a suitable generator for stacks, modify `Stack` to conform to `SequenceType`:

```
~~struct Stack<T> {~~
struct Stack<T> : SequenceType {
    var items = [T]()

    mutating func push(newItem: T) {
        items.append(newItem)
    }

    mutating func pop() -> T? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }

    func map<U>(f: (T) -> (U)) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }

    func generate() -> StackGenerator<T> {
        return StackGenerator(stack: self)
    }
}
```

You again made use of Swift's type inference to avoid having to explicitly state `typealias Generator = StackGenerator<T>`, although it would not be an error to do so.

The `SequenceType` protocol is what Swift uses internally for its `for ... in` loops. Now that `Stack` conforms to `SequenceType`, you can loop over its contents:

```
while let value = myStackGenerator.next() {
    println("got \(value)")
}

for value in myStack {
    println("for-in loop: got \(value)")
}
```

`StackGenerator` pops values off of its stack every time `next()` is called, which is a fairly destructive operation. When a `StackGenerator` returns nil from `next()`, its stack property is now empty. However, you were able to create a generator by hand from `myStack`, and then use `myStack` again in a `for ... in` loop. This reuse is possible because `Stack` is a value type, which means every time a `StackGenerator` is created, it gets a copy of the stack, leaving the original untouched.

A final note: if a protocol has an associated type, you cannot use that protocol as a concrete type. For example, you cannot declare a variable with the type `GeneratorType` or declare a function that accepts an argument of type `GeneratorType`, because `GeneratorType` has an associated type. However, protocols with associated types are fundamental to using *where clauses* in generic declarations.

Type Constraint where Clauses

Write a new function that takes every element of an array and pushes it onto a stack:

```
func pushItemsOntoStack<T>(inout stack: Stack<T>, fromArray array: [T]) {
    for item in array {
        stack.push(item)
    }
}

pushItemsOntoStack(&myStack, fromArray: [1, 2, 3])
for value in myStack {
    println("after pushing: got \(value)")
}
```

`pushItemsOntoStack` takes its first argument, a `Stack`, as an `inout` argument so that it can call the mutating method `push()`. This function is useful, but it is not as general as it could be. Now you know that any type that conforms to `SequenceType` can be used in a `for ... in` loop, so why should this function require an array? It should be able to accept any kind of sequence; even another `Stack`, now that `Stack` conforms to `SequenceType`. A first attempt at this will produce a compiler error:

```
~~func pushItemsOntoStack<T>(inout stack: Stack<T>, fromArray array: [T]) {~~
func pushItemsOntoStack<T, S: SequenceType>(inout stack: Stack<T>, fromSequence sequence: S) {
    ~~for item in array {~~
    for item in sequence {
        stack.push(item)
    }
}
```

You made `pushItemsOntoStack` generic with two placeholder types: `T`, which is the type of the stack's elements, and `S`, which is some type that conforms to the `SequenceType` protocol. The constraint on `S` guarantees that you can loop over it with the `for ... in` syntax; however, this is not sufficient. In order to push the items you get from `sequence` onto the stack, you need to guarantee that the type of the items coming from the sequence matches the type of the stack's elements. That is, you need to add an additional constraint that the elements produced by `S` are themselves of type `T`. Swift supports constraints of this kind using a `where` clause:

```
~~func pushItemsOntoStack<T, S: SequenceType>(inout stack: Stack<T>, fromSequence sequence: S) {~~
func pushItemsOntoStack<T, S: SequenceType where S.Generator.Element == T>(
        inout stack: Stack<T>, fromSequence sequence: S) {
    for item in sequence {
        stack.push(item)
    }
}
```

`pushItemsOntoStack` is a generic function with two placeholder types. The first placeholder type, `T`, has no constraints - it can be anything. The second placeholder type, `S`, has a constraint that the concrete type used must conform to the `SequenceType` protocol. Following the placeholder types, the where clause imposes further restrictions. `S.Generator.Element` refers to the `Element` type associated to the `Generator` type associated to `S`. The constraint `S.Generator.Element == T` requires that the concrete type used for the `Element` associated type must match the concrete type used for `T`.

The syntax for generic where clauses is often difficult to read at first glance, but an example should make it more clear. If you pass a stack of `Int`s as the first argument to `pushItemsOntoStack`, the second argument must be a sequence that produces `Int`s. Two types you already know that are `Int`-producing sequences are `Stack<Int>` and `[Int]`; try them out:

```
var myOtherStack = Stack<Int>()
pushItemsOntoStack(&myOtherStack, fromSequence: [1, 2, 3])
pushItemsOntoStack(&myStack, fromSequence: myOtherStack)
for value in myStack {
    println("after pushing items onto stack, got \(value)")
}
```

You created a new, empty stack of integers: `myOtherStack`. Next, you pushed all the integers from an array onto `myOtherStack`. Finally, you pushed all the integers from `myOtherStack` onto `myStack`. You were able to use the same generic function in both cases because arrays and stacks both conform to `SequenceType`.

Conclusion

Generics are an extermely powerful feature of Swift. You now understand how the Swift standard library includes data structures like arrays that can hold values of any type, and how to create your own generic data structures. You also know how to write generic functions and methods. Finally, you learned about associated type protocols and how you can use type constraints to write generic functions that accept types that provide specific behavior. If generics have not sunk in, don't fret - they are a simultaneously complex and abstract concept. Take your time and understand the `Stack` class you wrote throughout this chapter, and try your hand at the challenges.

Challenges

Bronze Challenge

Add a `filter` method to your `Stack` structure. It should take a single argument, a closure that takes a `T` and returns a `Bool`, and return a new `Stack<T>` that contains any elements for which the closure returns true.

Silver Challenge

Write a generic function called `findAll` that takes an array of any type `T` that conforms to the `Equatable` protocol and a single element (also of type `T`). `findAll` should return an array of integers corresponding to every location where the element was found in the array. For example, `findAll([5,3,7,3,9], 3)` should return `[1,3]` because the item 3 exists at indices 1 and 3 in the array. Try your function with both integers and strings.

Gold Challenge

Modify the `findAll` function you wrote for the Silver challenge to accept a generic `CollectionType` instead of an array. Hint: You will need to change the return type from `[Int]` to an array of an associated type of the `CollectionType` protocol.

For the More Curious: Parametric Polymorphism

In Chapter 15, you learned about class inheritance. Any function that expects an argument of a class can also accept arguments that are subclasses of that class. This ability to accept either a class or any subclass of it is often referred to as *polymorphism*, but is more accurately known as *run-time polymorphism* or *subclass polymorphism*. Polymorphism, meaning many forms, means you have written a single function that can accept different types.

Run-time polymorphism is a very powerful tool, and the frameworks Apple provides for iOS and Mac OS X development use it very frequently. Unfortunately, it has drawbacks. Classes that are related by inheritance are tied together tightly; it can be difficult to change one without affecting the others. There is also a small but observable performance penalty to run-time polymorphism due to how the compiler must implement functions that accept class arguments.

Swift's feature to add constraints to generics allow you to use another form of polymorphism: *compile-time polymorphism*, also known as *parametric polymorphism*. Generic functions with constraints are still true to the definition of polymorphism: you can write a single function that accepts different types. Compile-time polymorphic functions address both of the issues listed above that plague run-time polymorphism. Many different types can conform to a protocol, allowing them to be used in any generic function that requires a type conforming to that protocol, but the types can be otherwise unrelated, making it easy to change any one of them without affecting the others. Additionally, compile-time polymorphism generally does not have a performance penalty. In the playground, you called `pushItemsOntoStack` once with an array and once with a stack. The compiler actually produced two different version of `pushItemsOntoStack` in the executable, meaning the function itself doesn't have to do anything at runtime to handle the different argument types.

It is too early in the lifetime of Swift to make many generalizations about good Swift style and idioms. We will go out on a limb and suggest that based on the development of the Swift standard library, we expect generics and compile-time polymorphism to play a large role in the future. The next time you start to write a class hierarchy, consider whether whatever problem you are trying to solve might be better served with a solution featuring protocols and generics.

21

Memory Management and ARC

All computer programs use memory. Most computer programs use memory dynamically: as the program runs, it will allocate and deallocate memory as needed. Swift's stance on memory management is relatively unique. Most issues are handled for you automatically, but the language does not use a garbage collector (a common tool for automatic memory management in programming languages). Instead, Swift uses a system of reference counting. In this chapter, you will investigate how that system works and learn what you need to be aware of to avoid memory leaks.

Memory Allocation

The memory allocation and management for value types - enumerations and structures - is very simple. When you create a new instance of a value type, an appropriate amount of memory is automatically set aside for your instance. Anything you do to pass the instance around, including passing it to a function and storing it in a property, creates a copy of the instance. Swift reclaims the memory when the instance no longer exists. You do not have to do anything to manage the memory of value types.

The remainder of this chapter is about managing the memory for reference types - specifically, class instances. When you create a new class instance, memory is allocated for the instance to use, just like for value types. However, the difference comes in what happens when you pass the class instance around. Passing a class instance to a function or storing it in a property creates an additional reference to the same memory, rather than copying the instance itself. Having multiple references to the same memory means that when any one of them changes the class instance, that change is apparent to all of the references.

Swift does not require you to manually manage memory as in a language like C. Instead, every class instance has a *reference count*, which is the number of references to the memory making up the class instance. The instance remains alive as long as the reference count is greater than 0. Once the reference count becomes 0, the instance is deallocated and your `deinit` method will run.

In the not too distant past, apps that were developed for iOS and Mac OS X in Objective-C used "manual reference counting". Manual reference counting required you, the programmer, to manage the reference counts of all your class instances. Every class had a method to *retain* the object (incrementing its reference count) and a method to *release* the instance (decrementing its reference count). As you can probably imagine, manual reference counting was the source of many bugs: if you retain an instance too many times, it never gets deallocated (causing what is called a memory leak), but if you release an instance too many times, that would usually result in a crash.

In 2011, Apple introduced *Automatic Reference Counting* (ARC) for Objective-C. Under ARC, the compiler is responsible for analyzing your code and inserting retain and release calls in all of the appropriate places. Swift is also built on top of ARC. You do not have to do anything to manage the reference count of class instances - the compiler does that for you. However, it is still important for you to understand how the system works. There are some common mistakes that can cause memory management problems.

Strong Reference Cycles

Create a new Command Line Tool named CyclicalAssets. Add a new file to your project named `Person.swift`, and insert the following definition of the `Person` class:

```
import Foundation

class Person : Printable {
    let name: String

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        println("\(self) is being deallocated")
    }
}
```

The `Person` class has a single property that you set in its initializer. It conforms to the `Printable` protocol by implementing the `description` computed property. You added an implementation of `deinit` so you can see when a person is being deallocated; i.e., its memory is being reclaimed because the reference count has dropped to 0. Modify `main.swift` to create an optional `Person`:

```
import Foundation

~~println("Hello, world!")~~
var bob: Person? = Person(name: "Bob")
println("created \(bob)")

bob = nil
println("the bob variable is now \(bob)")
```

You created a new `Person?`, printed out its name, then set it to `nil`. (You made this an optional so that you could set it to nil and therefore see the `deinit` execute.) Build and run your program. You should see the following output:

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
```

The `bob` variable is an Optional that contains a class instance - a reference type. By default, all references that you create are *strong references*, which means they increment the reference count of the instance they refer to. Therefore, the `Person` whose name is Bob has a reference count of 1 after it is created and assigned to the `bob` variable. When you set `bob` to nil, Bob's reference count is decremented. You then see the `Person Bob is being deallocated` message, because the reference count has dropped to 0.

Next, create a new Swift file called `Asset.swift`, and insert an `Asset` class:

```
import Foundation

class Asset : Printable {
    let name: String
    let value: Double
    var owner: Person?

    var description: String {
        if let actualOwner = owner {
            return "Asset(\(name), worth \(value), owned by \(actualOwner))"
        } else {
            return "Asset(\(name), worth \(value), not owned by anyone)"
        }
    }

    init(name: String, value: Double) {
        self.name = name
        self.value = value
    }

    deinit {
        println("\(self) is being deallocated")
    }
}
```

The `Asset` class is very similar to the `Person` class. `Asset` has `name` and `value` properties, conforms to `Printable`, and prints a message when it is deallocated. It also has a variable stored property, `owner`, which will refer to the `Person` who owns this asset. `owner` is optional because it is reasonable for an asset to exist without someone owning it. Create a few assets in `main.swift`:

```
import Foundation

var bob: Person? = Person(name: "Bob")
println("created \(bob)")

var laptop: Asset? = Asset(name: "Shiny Laptop", value: 1500.0)
var hat: Asset? = Asset(name: "Cowboy Hat", value: 175.0)
var backpack: Asset? = Asset(name: "Blue Backpack", value: 45.0)

bob = nil
println("the bob variable is now \(bob)")

laptop = nil
hat = nil
backpack = nil
```

You again used optionals so that you could set the instances to be nil, which would fire off the `deinit` methods. As expected, all of the assets are deallocated and do not have owners:

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
the bob variable is now nil
```

People can own things; your person class will model this quality by having a property for assets. Go back to `Person.swift`, and add a property and method for people to gain assets:

```
import Foundation

class Person : Printable {
    let name: String
    var assets = [Asset]()

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        println("\(self) is being deallocated")
    }

    func takeOwnershipOfAsset(asset: Asset) {
        asset.owner = self
        assets.append(asset)
    }
}
```

You added `assets`, an array of `Assets` that the person owns, and `takeOwnershipOfAsset`, a method to give an asset to a person. Taking ownership of an asset means the person adds it to their `assets` array and sets the asset's `owner` property to refer back to this person. In `main.swift`, give Bob ownership of a couple of assets:

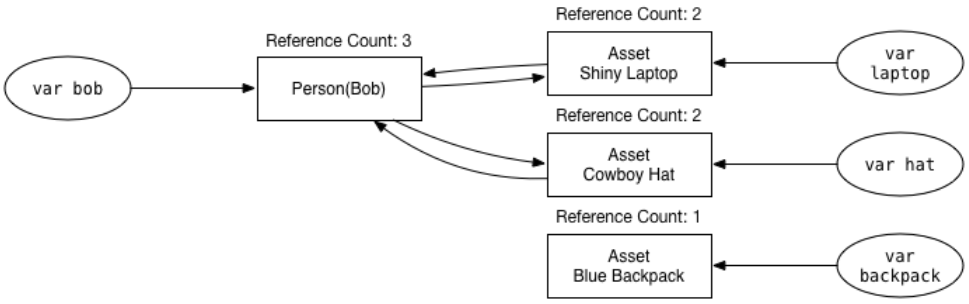
```
var laptop: Asset? = Asset(name: "Shiny Laptop")
var hat: Asset? = Asset(name: "Cowboy Hat")
var backpack: Asset? = Asset(name: "Blue Backpack")

bob?.takeOwnershipOfAsset(laptop!)
bob?.takeOwnershipOfAsset(hat!)

bob = nil
```

Build and run again. The output may be surprising:

```
created Optional(Person(Bob))
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
the bob variable is now nil
```

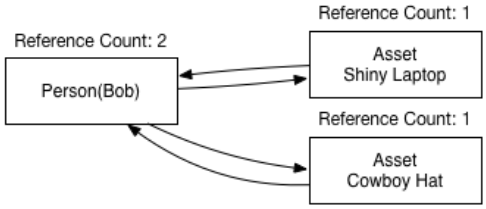
The only instance being deallocated now is the backpack - its reference count dropped to 0 when you set `backpack = nil`. The laptop, hat, and Bob himself are no longer being deallocated. Why not? Consider this diagram of who has a reference to whom, before any of the variables are set to nil in `main.swift`:

Figure 21.1 CyclicalAssets Before



Each instance is labeled with its current reference count. The reference count is exactly the number of arrows pointing to the instance; i.e., the number of references to the instance. After you set all the variables in `main.swift` to nil, those references go away, leaving this:

Figure 21.2 CyclicalAssets After



You have created two *strong reference cycles*, which is the term used when two instances have strong references to each other. Bob has a reference to the laptop (via his `assets` property), and the laptop has a reference to Bob (via its `owner` property). The same cycle exists between Bob and the hat. The memory for these instances is no longer reachable - all the variables pointing to them are gone - but the memory will never be reclaimed because each instance has a reference count greater than 0.

The solution to a strong reference cycle is to break the cycle. You could break the cycle in `Person`'s `deinit` method by looping over each asset and setting its owner to nil. However, Swift provides a word to get the same effect automatically. Modify `Asset` to make the `owner property` a *weak reference* instead of a strong reference:

```
class Asset : Printable {
    let name: String
    let value: Double
    ~~var owner: Person?~~
    weak var owner: Person?

    // ... rest of Asset remains unchanged
}
```

A weak reference is a reference that does not increase the reference count of the instance it is referring to. In this case, making `owner` a weak reference means when you assigned Bob as the owner of the laptop and the hat, Bob's reference count did not increase. The only strong reference to Bob is the `bob` variable in `main.swift`. When you set the `bob` variable to nil, the reference count on Bob drops to 0, so it is deallocated. When Bob

is deallocated, he no longer holds a strong reference to his assets, so their reference counts drop to 0 as well. Running your program again confirms that all the objects are deallocated:

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
```

What happens to a weak reference if the instance it is referring to is deallocated? The weak reference is set to nil. You can see this in action by editing `main.swift`:

```
bob?.takeOwnershipOfAsset(laptop!)
bob?.takeOwnershipOfAsset(hat!)

println("While Bob is alive, hat's owner is \(hat!.owner)")
bob = nil
println("the bob variable is now \(bob)")
println("After Bob is deallocated, hat's owner is \(hat!.owner)")
```

There are two requirements on weak references:

- Weak references must always be declared as `var`, not `let`.
- Weak references must always be declared as optional

Both of these requirements are the result of weak references being changed to nil if the instance they point to is deallocated. The only types that can become nil are optionals, so weak references must be optional. The instances declared with `let` cannot change, so weak references must be declared with `var`.

In most cases, strong reference cycles like the one you just resolved are easy to avoid. `Person` is a class that owns assets, so it makes sense that it would keep strong references to the assets. `Asset` is a class that is owned by a `Person`; if it wants a reference to its owner, that reference should be weak. After all, a person owns an asset; an asset does not own a person!

There is another way to create reference cycles that is much more subtle: capturing `self` in a closure.

Reference Cycles in Closures

Time to add an accountant class that will keep track of a `Person`'s net worth. Create a new Swift file called `Accountant.swift` and insert the following:

```
import Foundation

class Accountant {
    typealias NetWorthChanged = (Double) -> ()

    var netWorthChangedHandler: NetWorthChanged? = nil
    var netWorth: Double = 0.0 {
        didSet {
            netWorthChangedHandler?(netWorth)
        }
    }

    func gainedNewAsset(asset: Asset) {
        netWorth += asset.value
    }
}
```

`Accountant` defines a typealias, `NetWorthChanged`, which is a closure that takes a `Double` (the new net worth value) and returns nothing. It has two properties: `netWorthChangedHandler`, which is an optional closure to call when the net worth changes, and `netWorth`, the current net worth of a person. `netWorth` has a `didSet` property observer that calls the `netWorthChangedHandler` closure if it is non-nil. Finally, the `gainedNewAsset` should be called to tell the accountant that the value of a new asset should be added to the net worth value.

Update `Person.swift` to have an accountant to track their net worth:

```
import Foundation

class Person : Printable {
    let name: String
    let acccountant = Accountant()
    var assets = [Asset]()

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            netWorth in

            self.netWorthDidChange(netWorth)
            return
        }
    }

    deinit {
        println("\(self) is being deallocated")
    }

    func takeOwnershipOfAsset(asset: Asset) {
        asset.owner = self
        assets.append(asset)
        accountant.gainedNewAsset(asset)
    }

    func netWorthDidChange(netWorth: Double) {
        println("The net worth of \(self) is now \(netWorth)")
    }
}
```

You added an `accountant` property that has a default value of a new `Accountant`. `Person` has a strong reference to their `Accountant`, which is perfectly reasonable. In `init`, you set the `netWorthChangedHandler` on the accountant to call your new `netWorthDidChange` method, which just logs the person's new net worth. Finally, you updated `takeOwnershipOfAsset` to notify the accountant of new assets. Build and run your program; you should see the following:

```
created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
the bob variable is now nil
After Bob is deallocated, hat's owner is Optional(Person(Bob))
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
```

You got the log messages of the net worth changing, so all of the accountant code you added appears to be working correctly. However, it looks like the memory leak is back: Bob, the laptop, and the hat are not being deallocated. Why aren't these instances being removed from memory?

It turns out that your new code includes a not-so-obvious strong reference cycle. `Person` has a strong reference to `Accountant`, but `Accountant` does not have a strong reference back to `Person`, at least at first glance. To get a hint and what is going on, try modifying the `init` method of `Person` (this will cause a compiler error):

```
    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            netWorth in

            ~~self.netWorthDidChange(netWorth)~~
            netWorthDidChange(netWorth)
            return
        }
    }
```

Try to build your program now. The error message you receive states, `Call to method 'netWorthDidChange' in closure requires explicit 'self.' to make capture semantics explicit`. What are the "capture semantics" of a closure?

A closure has its own scope within its definition. By default, a closure takes a strong reference to any variables that it used inside the scope. `netWorthDidChange` is a method on `self`, so calling it would give the closure a strong reference to `self`. This explains why you are leaking memory: `Accountant` actually does have a strong reference back to `Person`! `Accountant`'s `netWorthChangedHandler` is holding a strong reference to its owning `Person` via that `Person`'s `self`.

Take another look at the error message: "to make capture semantics explicit". Swift could allow you to use `self` implicitly in closures, but doing so would make it very easy to accidentally create strong reference cycles just like you have done here. Instead, the language requires you to be *explicit* about your use of `self`, forcing you to consider whether a reference cycle is a possibility.

To change the capture semantics of a closure to capture references weakly, you can use a *capture list*. Modify `Person.swift` to use a capture list when creating the closure:

```
    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            ~~netWorth in~~
            [weak self] netWorth in

            ~~netWorthDidChange(netWorth)~~
            self?.netWorthDidChange(netWorth)
            return
        }
    }
```

The capture list syntax is a list of variables inside square brackets (`[]`) immediately before the list of the closure arguments. The capture list you wrote here tells Swift to capture `self` weakly instead of strongly. Now that the `Accountant`'s closure no longer strongly references the `Person`, the strong reference cycle is broken.

Note the use of `self?` in the body of the closure. Because `self` is captured weakly and all weak instances must be optional, `self` inside the closure is optional. Run your program again, and confirm that all the instances are being deallocated appropriately:

```
created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
After Bob is deallocated, hat's owner is nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
```

Conclusion

You do not have to manually manage memory in Swift. ARC does the right thing for you the vast majority of the time. Sadly, it does not solve the problem of strong reference cycles. You learned how to think about owning relationships between class instances. You also learned why closures require you to be explicit about references to `self`, and how to use a capture list to capture `self` weakly in a closure.

Challenges

Bronze Challenge

The idea of asset ownership by a `Person` is incomplete. `Person` has a way to take ownership of an asset, but no way to give up ownership of an asset. Update `Person` so that an instance can relinquish an asset. (Hint: You will probably need to update `Accountant` too if you want a valid net worth value.)

Silver Challenge

Create another `Person` in `main.swift`. Immediately after you give Bob ownership of the laptop, try giving your new `Person` ownership of the same laptop. Now both people own the laptop! Fix this bug.

22

Equatable and Comparable

Much of programming depends upon comparing values. It is important to be able to know if one value is equal to another value. Moreover, it is often important to be able know *how* one value compares to another. Is it less than or greater than a given value? Swift provides two protocols for testing equality and comparability: `Equatable` and `Comparable`.

This chapter will show you how you can make a custom type conform to these protocols. Doing so will involve implementing a few functions that will teach instances of your type how to compare themselves to other instances of the same type. Create a new Playground called `Comparison.playground` to begin.

Conforming to Equatable

Begin by creating a new type that does not yet conform to the `Equatable` protocol.

```
struct Point {
    let x: Int
    let y: Int
}
```

The struct above defines a `Point` type. `Point` uses its `x` and `y` properties to describe a specific location on a two-dimensional plane.

At the moment, `Point` does not know how to determine if an instance is equal to another instance. Create two instances of this type to see what happens when you check if the two are equal.

```
struct Point {
    let x: Int
    let y: Int
}
let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
```

You created two new points, `a` and `b`, using the free memberwise initializer provided by the compiler. Notice that the points were given the same values for the `x` and `y` properties. Try to use the `==` operator to test for equality between these two points.

```
struct Point {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
```

You should see that this check for equality does not work. In fact, it generates an error from the compiler. This error stems from the fact that you have not yet taught your `Point` struct how to test for equality between two instances.

Doing so will involve making `Point` conform to the `Equatable` protocol. Add the following code to your struct's declaration.

```
struct Point {
    let x: Int
    let y: Int
}

struct Point: Equatable {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
```

You should now see a new error; this time, the error is on the line where you declare the `Point` struct. The error says: `Type 'Point' does not conform to protocol 'Equatable'`. In order to figure out how to conform to the `Equatable` protocol, open the documentation by clicking on the "Help" menu item, and select "Documentation and API Reference".

The documentation tells you that you are required to overload the `==` operator at global scope in order to conform to this protocol. Why do you have to conform at global scope? If you think about it, this requirement makes sense because `==` is not a method that you call on a specific type. Instead, `==` appears between and operates on two targets at once: it checks to see if the value on the left hand side is equal to the value on the right hand side. This sort of operator is called an *infix operator*.

Provide an implementation of `==` that compares two instances of the `Point` type and tests for their equality.

```
struct Point: Equatable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
```

You have now written a new implementation of the `==` operator (note: an operator is a just a function with a special name; for example, `==`). This definition has two arguments: a `lhs` arguments for the left hand side, and a `rhs` for the right hand side of the equality check. Both of these arguments are expected to be of type `Point`.

The function's implementation is fairly straightforward. It compares the `x` and `y` values for both instances of the `Point` type that are passed into the function's arguments. Last, the function will return a `Bool`, indicating whether or not the instances are equal.

Take a look at your Playground's results sidebar. You should notice that the errors are gone and that your test for equality between the two points `a` and `b` succeeds. The two points are equal because their `x` and `y` values are the same.

But there is more. Swift's standard library provides a default implementation of the `!=` function that depends upon the definition of `==`. This feature means that if your type conforms to `Equatable` by implementing its own version of `==`, then it also has a working implementation of the `!=` function.

Try it out by adding the following test.

```
struct Point: Equatable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
```

The results sidebar should update to show that this test for inequality yields `false`. In other words, the two points are *equal*, which means that they are not *unequal*. You can interpret `false` as answering "*No!*" to the question: "*Are points A and B unequal?*".

Conforming to Comparable

Now that your `Point` type conforms to the `Equatable` protocol, you maybe interested in more nuanced forms of comparison. For example, perhaps you are interested in knowing if a point is less than another point. You accomplish this functionality by conforming to the `Comparable` protocol.

Open up the documentation and search for "Comparable" to determine what is needed. You will find that you need to overload one operator: the < infix operator. Add the following code your struct to make it conform to `Comparable`.

```
~~struct Point: Equatable {~~
    ~~let x: Int~~
    ~~let y: Int~~
~~}~~
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
```

You have added a new declaration to your `Point` that says it now conforms to the `Comparable` protocol. A bit below that, you added an implementation of the **<** operator. This implementation works similar to what you saw above for your implementation of **==**. It checks to see if the point passed in on the left hand side is less than the point passed in on the right hand side. If the `x` and `y` values for the point on the left hand side are both smaller than the values on the right hand side, then the function will return `true`. Otherwise, the function will return `false`, indicating that the point on the left hand side is greater than the point on the right hand side.

Create two new points to test this function.

```
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == b)
let cLessThanD = (c < d)
```

Notice that these two points are given different values for `x` and `y`. You also check to see if `c` and `d` are equal, which returns `false`; the two points are not the same. Last, you exercise your overload of the **<** operator to determine if `c` is less than `d`. In this case, the comparison evaluates to `true`. The point `c` is less than the point `d` because both its `x` and `y` values are smaller than `d`'s.

As before with conforming to the `Equatable` protocol, implementing one function actually gives you much more functionality. This result is because the Swift standard library defines the **>**, **>=**, and **<=** operators in terms of the **<** and **==** operators. Accordingly, if your type conforms to `Comparable`, then it will get implementations of these operators for free.

Test this functionality by adding a series of new comparisons.

```
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == b)
let cLessThanD = (c < d)

let cLessThanEqualD = (c <= d)
let cGreaterThanD = (c > d)
let cGreaterThanEqualD = (c >= d)
```

These last three comparisons check to see if:

- a) `c` is less than or equal to `d`,
- b) `c` is greater than `d`,
- c) `c` is greater than or equal to `d`.

As anticipated, these comparisons evaluate to `true`, `false`, and `false` respectively.

Comparable's Inheritance

`Comparable` actually inherits from `Equatable`. You may be able to guess what the implication of this inheritance is. In order to conform to the `Comparable` protocol, you must also conform to the `Equatable` protocol.

Since `Equatable` requires the **`==`** operator, any conformance to `Comparable` must supply an implementation of this function. Note that this relationship between the two protocols means that your type does not have to explicitly declare conformance to `Equatable`. Remove the explicit declaration of conformance to `Equatable` from your `Point` struct.

```
struct Point: ~~Equatable,~~ Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

...
```

You should notice that the playground works just as it did before.

One final note on style. While it is extremely explicit to declare conformance to both `Equatable` and `Comparable`, it is unnecessary. If your type conforms to `Comparable`, then it must conform to `Equatable` as well. This point is a detail listed in the documentation, which makes it an expected consequence of conforming to `Comparable`. Adding the explicit conformance to `Equatable` does not add that much more information. On the other hand, since it is not listed in the official documentation, it may make sense to have a type explicitly conform to all protocols involved when conforming to a custom protocol that inherits from another protocol.

Conclusion

This chapter introduced conforming to two system provided protocols: `Equatable` and `Comparable`. `Equatable` is used to determine if two instances of a given type are equal. `Comparable` is used to get a nuanced comparison between to instances of a type. For example, it helps you to determine if an instance is less than or greater than another instance. It is good to get comfortable with these protocols; you will use them frequently.

Challenges

Bronze Challenge

It is reasonable that you would want to add two points together in the future. The addition of two points should return a new `Point` that adds the given points' x-values and y-values respectively. Make it possible to add two points together by overloading the **`+`** operator for the `Point` struct.

Gold Challenge

Create a new `Person` class. It should have two properties: `name` and `age`. For convenience, create an initializer that provides arguments for both of these properties.

Next, create two new instances of the `Person` class. Assign those instances to two constants named `p1` and `p2`. Also create an array named `people` to hold these instances, and put them inside of it.

You will occasionally need to find the index of a instance of a custom type within an array. Try doing so with the **`find(_:,_:)`**. The first argument expects the collection you will be searching within, and the second argument is the value whose index you would like to find. Use the function to find the index of `p1` inside of the `people` array.

You will get an error. Take some time to understand the error, and then resolve it. You should be able to assign the result of the **`find(_:,_:)`** to a constant named `p1Index`. Its value should be 0.

For the More Curious: Custom Operators

Swift allows the developer to create their own custom operators. This feature means that you can create your own operator to signify that one instance of the `Person` type has married another instance. Say, for example, you want to create the **`+++`** to marry one instance to another.

Create a new `Person` class like so:

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}
```

The class has two properties: one for a name, and another for a spouse. It also has an initializer that will give values to those properties. Note that the `spouse` property is an optional to indicate that a person may not necessarily have a spouse.

Next, create two instances of this class.

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)
```

Now, declare your new infix operator. It has to be declared at global scope. Also define how the new operator function will work.

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++ {}

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}
```

The new operator, **`+++`**, will be used to marry two instances of the `Person` class. As an infix operator, it will be used in between two instances. Thus, the implementation of **`+++`** will simply assign each instance to the other's `spouse` property.

Exercise this new operator like so:

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++ {}

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}

matt +++ drew
matt.spouse?.name
drew.spouse?.name
```

The code `matt +++ drew` serves to marry the two instances. You can check that this process worked by examining the Playground's results sidebar.

While this operator "works", and it is not too difficult to determine what is going on by looking at it, it is generally a good idea to avoid declaring your own custom operators. It is good practice to only create custom operators for your own types when the operator will be recognizable to anyone that may read your code. That typically means restricting your own custom operators to the realm of well-known mathematical operators. Indeed, Swift only allows you to use a well-defined collection of mathematical symbols to create custom operators. For example, you cannot refactor the **`+++`** operator to be the Emoji "face throwing throwing a kiss" (i.e., `U+1F61A`).

The thinking behind this recommendation is driven by the fear that somebody reviewing your code in the future may not know exactly what you meant by **`+++`**. After all, it is fairly ambiguous in this case. Moreover, it is not as though this operator accomplishes something more elegantly or efficiently than a **`marry(_:)`** could not accomplish in its own right.

For example, a **`marry(_:)`** function might look like this:

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }

    func marry(spouse: Person) {
        self.spouse = spouse
        spouse.spouse = self
    }
}

~~let matt = Person(name: "Matt", spouse: nil)~~
~~let drew = Person(name: "Drew", spouse: nil)~~

~~infix operator +++ {}~~

~~func +++(lhs: Person, rhs: Person) {~~
    ~~lhs.spouse = rhs~~
    ~~rhs.spouse = lhs~~
~~}~~

~~matt +++ drew~~
matt.marry(drew)
matt.spouse?.name
drew.spouse?.name
```

The code `matt.marry(drew)` is far more readable, and it is quite clear about what the code is doing. These qualities will make it easier to maintain this code in the future.

Index

T

U

V

W

X