LEARNING Swift™ 2 PROGRAMMING

JACOB SCHATZ

Learning Swift 2 Programming

Second Edition

The raw value of `Diamonds` is `3` because of the auto-increment. `Clubs` is `1`, `Hearts` is `2`, `Diamonds` is `3`, and `Spades` is `4`.

Play around with this and change `Clubs` to any integer you want. Try changing it to `100`, or `-10`, or `0`. It still auto-increments perfectly.

You can use the constructor `Suit(rawValue: n)` to do the opposite of `rawValue` by getting the raw value from an integer (or whatever type your enum is). `Suit(rawValue: n)` returns an optional of type `Suit`. Why is it an optional? You might try to grab the member with a raw value of `4000`, and that would not exist. However, because `Suit(rawValue: n)` gives a suit (in an optional), it's helpful to compare it to *something* rather than just printing it out. Here's what it looks like:

```
Suit(rawValue: 3) == chosenSuit // true
```

You can then use value binding to find the member for the raw value:

```
enum Suit:Int {
    case Clubs = 1, Hearts, Diamonds, Spades
}
var result = "Don't know yet."
if let theSuit = Suit(rawValue: 3) {
    switch theSuit {
    case .Clubs:
        result = "You chose clubs"
    case .Hearts:
        result = "You chose hearts"
    case .Diamonds:
        result = "You chose diamonds"
    case .Spades:
        result = "You chose spades"
    }
} else {
    result = "Nothing"
}
result // You chose diamonds
```

Here you have to do value binding for `Suit(rawValue: 3)` because it is an optional and could have been `nil`. After you get `theSuit` out of the value binding, assuming it's not `nil`, you can use your normal `switch` statement to find the chosen suit. Notice that the `result` variable was successfully changed even though you scoped it through the `if` and `switch` statements.

Structs

Structs (which is short for *structures*) are copied when they're passed around. Classes are passed around by reference. This means that you will never have the same instance of a struct. Conversely, you can have multiple instances of the same class.

Here is what classes and structs have in common:

- Both define properties to store values.
- Both define methods to provide functionality.
- Both provide subscripts to give access to their values.
- Both provide initializers to allow you to set up their initial state.
- Both can be extended to provide additional functionality beyond a default implementation. (This is different from inheritance.)
- Both have the capability to conform to protocols (which you will learn about in Chapter 8, "Expanding Your Reach: Protocols and Extensions").

> **Note**
>
> Do not worry too much if you don't understand everything in these lists. You will understand it all by the end of this chapter or in later chapters.

The following is the difference between classes and structs:

- Classes have inheritance.
- Classes have type checking.
- Structs have deinitializers so you can free up unused instances.
- Structs have reference counting. You can have more than one reference to a class instance.

Here's an example of a simple struct:

```
struct GeoPoint {
    var lat = 0.0
    var long = 0.0
}
```

This defines a new struct of type `GeoPoint`. You give the struct two properties and declare them as `doubles`. (Even though you don't see any explicit type declaration, it is happening because `0.0` is inferred as a `double`.)

Now you can use the new struct. If you want to interact with the `GeoPoint` struct, you must create a `GeoPoint` instance:

```
var somePlaceOnEarth = GeoPoint()
```

Now you can interact with the new `GeoPoint` struct, using the dot syntax:

```
somePlaceOnEarth.lat = 21.11111
somePlaceOnEarth.long = 24.232323
```

Notice that when you created a new `GeoPoint` struct, the code completion gives you the option to initialize it with properties (see Figure 4.1).

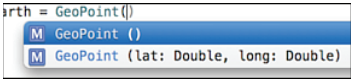

Figure 4.1 Code complete for GeoPoint shows multiple initializers

You can also write the last three lines as one line:

```
var somePlaceOnEarth = GeoPoint(lat: 21.1111, long: 24.23232)
```

Defining Methods in Structs

When we say *methods,* we are talking about the functions within structs. Methods are just functions that are going to be associated with the structs. By defining a method within the curly brackets of a struct, you are saying that this function belongs to this struct.

Here's an example of a struct with `Point`, `Size`, and `Rect`, which will be based on `CGRect`:

```
struct Point {
    var x:Int, y:Int
}

struct Size {
    var width:Int, height:Int
}

struct Rect {
    var origin:Point, size:Size

    func center() -> Point {
        let  x = origin.x + size.width/2
        let y = origin.y + size.height/2

        return Point(x: x, y: y)
    }
}
```

There are a couple of things to note here. The first thing you might notice is that you declare all the variables on one line. You can use this simplified version of declaring variables where it makes your code more readable. For example,

```
var one = 1,two = 2, three = 3
```

is the same as this:

```
var one = 1
```

```
var two  = 2
var three = 3
```

You might also notice that you set types for the properties explicitly (for example, `origin:Point, size:Size`). You did not give your properties any default values so Swift would be unable to determine the types of these properties.

However, because you did not give `Rect` any default value, Swift will complain. If you try to make a new `Rect` without any default values in the initializer, you will get an error:

```
var rect:Rect = Rect() // error: missing parameter for 'origin' in call
```

Swift does not like that you did not initialize the properties in the struct itself and did not initialize the properties upon making a new `Rect`.

The initializer included with every struct is called a *memberwise initializer*. Memberwise initializers are part of a much larger concept that we won't cover here. When creating a `Rect`, you can use the memberwise initializer to get rid of the error:

```
var point = Point(x: 0, y: 0)
var size = Size(width: 100, height: 100)
var rect:Rect = Rect(origin: point, size: size)
rect.size.height
rect.center()
```

That's better! Since you used the memberwise initializers when constructing `Point`, `Size`, and `Rect`, you no longer get errors. Here you also used the `center()` method of the `Rect`, and it told you that the center of the `Rect` is `{x 50 y 50}`.

Structs Are Always Copied

Earlier we talked about how structs are always copied when they are passed around. Let's take a look at an example that proves this, using the `Point` struct because it's supersimple:

```
var point1 = Point(x:10, y:10)
```

Now you can create `point2` and assign it to `point1`:

```
var point2 = point1
```

You modify `point2`:

```
point2.x = 20
```

Now `point1` and `point2` are different:

```
point1.x // 10
point2.x // 20
```

If `point1` and `point2` were classes, you would not get different values because classes are passed by reference.

Mutating Methods

If a method inside a struct will alter a property of the struct itself, it must be declared as `mutating`. This means that if the struct has some property that belongs to the struct itself (not a local variable inside a method) and you try to set that property, you will get an error unless you mark that method as `mutating`. Here's a struct that will throw an error:

```
struct someStruct {
    var property1 = "Hi there"
    func method1() {
        property1 = "Hello there"
        // property1 belongs to the class itself
        // so we can't change this with making some changes
    }
    // ERROR: cannot assign to 'property1' in 'self'
}
```

The fix for this error is simple. Just add the word `mutating` in front of the `func` keyword:

```
struct someStruct {
    var property1 = "Hi there"
    mutating func method1() {
        property1 = "Hello there"
    }
    // does not throw an error! YAY
}
```

Now that this is fixed, let's take a look at what this error means:

```
cannot assign 'property1' in 'self'
```

Well, it is `property1` that you are trying to modify. This error says that you cannot assign `property1` to `self`. What is `self`? `self` in this case is the struct's own instance. In this struct, `property1` belongs to an instance of the struct. You could rewrite the line with `property1` to be `self.property1`. However, `self` is always implied, so you don't need to write it. Also notice that the following code works without the `mutating` keyword:

```
struct someStruct {
    func method1() {
        var property2 = "Can be changed"
        property2 = "Go ahead and change me"
    }
}
```

The reason you can set `property2` is because it does not belong to `self` directly. You are not modifying a property of `self`. You are modifying a local variable within `method1`.

Classes

In the following example of creating a class, notice that it looks just like a struct but with the word `class`:

```
class FirstClass {
    // class implementation goes here
}
```

You create a class exactly the same way you create a struct, but instead of using the word `struct`, you use the word `class`. Adding properties to a class is very similar. For example, the following `Car` class has properties for the make, model, and year (and you will define a default value for each property):

```
class Car {
    let make = "Ford"
    let model = "Taurus"
    let year = 2014
}
```

In this example, there are three immutable properties of the `Car` class. Remember that when you make a struct, you are able to leave these properties blank. If you do the same for a class, you get an error:

```
class Car {
    let make:String
    let model:String
    let year:Int
}
// error: class 'Car' has no initializers
```

If you want to fix this error, you must create an initializer for the `Car` class and initialize all the uninitialized properties. Classes in Swift don't have automatic initialization (that is, memberwise initializers). If you leave the properties without default values, you must provide an initializer for the class. Each of the uninitialized properties must be initialized.

Swift provides a global function `init()` for this very purpose. Some languages call this a *constructor*.

Initialization

Initialization is the process of getting the instance of a class or structure ready for use. In initialization, you take all things that do not have values and give them values. You can also do things like call methods, and do other initializations. The big difference between Objective-C initializers and Swift initializers is that Swift initializers do not have to return `self`. The goal of Swift initializers is to give a value to everything that does not have a value. Structs can define initializers even though they have their own memberwise initializers. You can also define multiple initializers for a class or struct. The simplest type of initializer is one without any