# SWIFT for the Really Impatient

Matt Henderson
Dave Wood

Foreword by Jeff LaMarche

# Swift for the Really Impatient

```
let iceCreamCup = (scoops:2, flavor:"chocolate", style:"cup")

switch iceCreamCup {
case (0,_,_):
    println("I don't like ice cream")
case (1,"vanilla",_):
    println("One single scoop of vanilla for me")
case (1,_,_):
    println("I'd like one scoop of \(iceCreamCup.flavor) in a
➥\(iceCreamCup.style)")
case (_, "chocolate", _):
    println("I love chocolate and I want \(iceCreamCup.scoops)
➥scoops!")
default:
    println("I'd like \(iceCreamCup.scoops) scoop(s) of
➥\(iceCreamCup.flavor) in a \(iceCreamCup.style)")
}
```

Another specific use is to define a `where` clause to a `case` to provide an extra level of flexibility to `switch` statements. You can also bind data as part of the `case` matching, which helps create code that is more readable and also more compact.

The power and expressiveness of Swift `switch` statements goes well beyond the power and expressiveness of C-style switches. They're a good example of the philosophies of safety and expressiveness that embody Swift.

Here's an example of a `switch` statement that has data binding and `where` clauses:

```
let indexSizeAndValue = (9, 10, "")
switch indexSizeAndValue {
case (0,_, let value):
    println("The first value is \(value)")
case let (index,size, "") where index == (size-1):
    println("The last value is empty")
case let (index,size,value) where index == (size-1):
    println("The last value is \(value)")
case let (index,_,value):
    println("The value at index \(index) is \(value)")
}
```

**CAUTION**

When multiple patterns match on a `switch` statement, only the first `case` statement that is matched executes. Make sure to always have the more specific patterns come before general patterns when there is a possibility of matching multiple patterns.

If you look closely at each of these `case` statements, you can see how easy it is to write readable, flexible code. The first `case` statement, `case (0, _ , let value)`, matches any value where the first element of the tuple is `0`, and it also binds the third element to a new constant with the name `value` that can be used in the body of the `case` statement. Using this pattern of binding data in the `case` clause creates code that is much easier to read than assigning the values inside the `case` statement's body.

The second `case` statement, `case let (index, size, "") where index == (size-1)`, shows how to use value binding for multiple elements. Notice that the keyword `let` is before the whole tuple—and it adds an additional requirement for matching the `case` statement by using a `where` clause.

You may have noticed that there isn't a default `case` in this example. This is because the last `case`, `case let (index, _ ,value)`, matches any remaining values and fulfills the requirement of providing a comprehensive set of `case` statements. By requiring an exhaustive set of `case` statements, `switch` statements provide more safety than a series of `if`/`else` statements, and when combined with value binding and `where` clauses, `switch` statements provide a useful and convenient control flow mechanism.

## Exercises

1. Use `String`'s `toInt()` method to create a function that adds two strings together and returns an `Int?`. Then create a function that takes an array of strings and computes a sum. Once you have a working function, can you make the code more readable by using optional binding?
2. Write a function that takes a string and an array of strings and returns the index of the string if it is found in the array or `-1` if it is not found. Next, make this function return an optional `Int`. How do these two implementations compare in terms of code readability and safety?

3. The `String` type has a `debugDescription` property. Write a function that iterates over an array of strings and prints to the console the value in `debugDescription`. Next, extend this function by using generics. How much of the function's body did you have to change? Did you have to change how the function is called when using a `[String]`?

4. Some types can be implicitly converted to other types. Add a float type declaration to the variable `myNumber`:

   ```
   var myNumber = 2
   
   var myDivisionResult = 7 / myNumber
   ```

   How does the behavior of this variable change?

5. Several methods on standard objects in Swift take closures as parameters. Use the `sort` method on an array with a closure that sorts an array of `Int` from largest to smallest.

6. Nested functions can be returned and then executed in the calling scope. Make a function that takes a string and returns a function that doesn't take any parameters and returns a string. (This string should be a string that is created using the original argument and modified in the nested function.) Here's what the function signature should look like:

   ```
    func stringCreator(initialString:String) -> (() -> String)
   ```

7. Write a function that takes a tuple that represents an x and y coordinate and returns the two-dimensional Cartesian quadrant that contains the coordinate. Next, modify the function to return a tuple that contains named values for x, y, and the quadrant.

8. Write a generic function that takes an array of objects and returns an array of tuples that combine the elements into pairs. (Objects at indexes 0 and 1 would be the first tuple.) How could you handle an array with an odd number of elements?

9. Take a series of `if`/`else` statements from an existing project and convert it to a `switch` statement in Swift. Is the code more readable? Were you able to reduce the number of lines of code? Is the Swift code safer?

10. Given a tuple that represents a three-dimensional point (x, y, z), write a `switch` statement that prints out the point's octant (or the origin). Next, extend this `switch` statement to match cases where two of the values within the point are equal. Can you rewrite your `switch` to use value binding or underscores in the pattern matching to make the code more readable?

◆ ◆ ◆ CHAPTER 3

# Objects and Classes

## Topics in This Chapter

In Chapter 1, "Introducing Swift," you saw that structs and enums in Swift are more powerful than their counterparts in C-based languages, and they are more akin to classes. All three data structure types are first-class types in Swift and can have properties, type methods, and instance methods, and developers can extend them as needed. This chapter goes into detail about the various types, their differences, and why you would use one type over another.

These are the key points in this chapter:

- Enumerations and structures in Swift are very much like classes and can include type and instance properties and methods.
- Enumerations and structures are value types, which means they're copied as they're passed around in code; on the other hand, classes are reference types, which means a reference to an object is passed around in code.
- Enumerations, structures, and classes may have methods (including their `init()` methods) overloaded to accept different sets of arguments.
- Classes may be subclassed, with some rules set in place by the parent class, such as marking methods as `final` to prevent them from being overridden and marking `init()` methods as `required` to ensure that subclasses include them.
- When overriding a method in a subclass, you must confirm to the compiler that you intend to do so by using the `override` keyword.
- A subclass that provides its own designated `init()` methods does not inherit the `init()` methods from its parent class.
- Properties may be stored or computed and may have observers attached to them to execute a closure before and/or after a property is updated.
- Custom classes may make subscripting syntax available to developers to provide easy access to internal data.
- A protocol defines a set of properties and/or methods that a type must contain in order to conform to that protocol.
- Protocols are a type unto themselves and can be used as the type required for a parameter or as a return type.
- Enumerations, structures, and classes may be extended through extensions, with the ability to add computed properties and/or methods—even for types that do not make their source code available.
- You can use access control to hide implementation details in your classes or to make specific entities public and available to other modules.

## 3.1 Enumerations

You can use an enumeration to create a custom data type with a predefined set of possible values. For example, you could create a `NetworkConnection` type that could be either `NotConnected`, `WiFi`, or `Cellular`. Here's how you'd do it:

```
enum NetworkConnection {
    case NotConnected
    case WiFi
    case Cellular
}
```

Then you can create a variable in your app to monitor your current connection and update it as the connection changes:

```
var currentConnection = NetworkConnection.WiFi
```

Here `currentConnection` is inferred to be of type `NetworkConnection` because you're initializing it to a `NetworkConnection` type as you declare it. Because the compiler knows what type `currentConnection` is, you can update its value by using a shorter dot notation omitting the type:

```
currentConnection = .Cellular
```

You can use the variable in a `switch` construct in order to update the app's interface for the current connection:

```
switch currentConnection {
case .NotConnected:
    showOfflineIndicator()
case .WiFi:
    showWiFiIndicator()
case .Cellular:
    showCellularIndicator()
}
```

By default, an `enum` in Swift does not map its members to a basic `Int` type, as you might expect if you're used to that behavior in C-based languages. So members of an enumeration in Swift have no defined order unless explicitly set. `NotConnected` is not less than `WiFi`; they are only not equal to each other. Instead, the possible members are their own value and can only be checked for equality to other members in the same `enum`. Here's how it looks: