Protocol-Oriented Programming with Swift

At the heart of Swift's design is an incredibly powerful idea: protocol-oriented programming. Its many benefits include better code maintainability, increased developer productivity and superior application performance. The book will teach the reader how to apply the ideas behind the protocol-oriented programming paradigm to improve the code they write.

This book will introduce the readers to the world of protocol-oriented programming in Swift and will demonstrate the ideas behind this new programming paradigm with real world examples. In addition to learning the concepts of protocol-oriented programming, it also shows the reader how to reduce the complexity of their codebase using protocol extensions. Beginning with how to create simple protocols, readers will learn how to extend protocols and also to assign behaviors to them.

By the end of this book readers will be able to harness the power of protocol-oriented programming to build real world applications.

Who this book is written for

This book is for Swift developers who want to learn and implement protocol-oriented programming in their real-world applications.

What you will learn from this book

- The difference between object-oriented programming and protocol-oriented programming
- The difference between reference and value types and when to use each
- How we can leverage tuples to reduce the complexity of our code
- What protocols are and how to use them
- How to implement protocol extensions to create a very flexible code base
- How to implement several design patterns with a protocol-oriented approach
- How to solve real-world design issues with protocol-oriented programming


Jon Hoffman

$ 39.99 US
£ 25.99 UK

Community Experience Distilled

Protocol-Oriented Programming with Swift

Build fast and powerful applications by exploiting the power of protocol-oriented programming in Swift
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Object-Oriented and Protocol-Oriented Programming'
- A synopsis of the book’s content
- More information on Protocol-Oriented Programming with Swift
Jon Hoffman has over 20 years of experience in the field of information technology. Over these 20 years, Jon has worked in the areas of system administration, network administration, network security, application development, and architecture. Currently, Jon works as a senior software engineer for Syn-Tech Systems.

Jon has been developing extensively for the iOS platform since 2008. This includes several apps that he has published in the app store—apps that he wrote for third parties and numerous enterprise applications.


What really drives Jon is the challenges in the Information Technology field and there is nothing more exhilarating to him than overcoming a challenge. Some of Jon’s other interests are watching baseball (go Sox) and basketball (go Celtics). Jon also really enjoys tae kwon do where he and his oldest daughter Kailey earned their black belts together early in 2014. Kim (his wife) earned her black belt in December 2014 and his youngest daughter is currently working towards her black belt.
This book is about protocol-oriented programming. When Apple announced Swift 2 at the World Wide Developers Conference (WWDC) in 2015, they also declared that Swift was the world's first protocol-oriented programming language. By its name, we may assume that protocol-oriented programming is all about the protocol; however, that would be wrong assumption. Protocol-oriented programming is about so much more than just the protocol; it is actually a new way of not only writing applications, but thinking about programming as well.

One of the biggest misconceptions about protocol-oriented programming is that it is just another name for object-oriented programming. Chapter 1, *Object-Oriented and Protocol-Oriented Programming*, immediately takes on this myth by comparing protocol-oriented programming to object-oriented programming to see what is similar and what the differences are. We will also discuss the advantages and disadvantages of both programming paradigms.

From Chapter 2, *Our Type Choices*, to Chapter 5, *Let’s Extend Some Types*, we will take an in-depth look at each of the components of the protocol-oriented programming paradigm. These chapters are designed to give the reader a solid understanding of the different components of protocol-oriented programming, so users will understand how they can use these components in their applications.

The last two chapters are written to help the reader understand how they can design their application in a protocol-oriented programming way. Chapter 6, *Adopting Design Patterns in Swift*, looks at how we would implement several design patterns in a protocol-oriented way and Chapter 7, *Case Studies*, looks at three real-world case studies to reinforce everything previously discussed in the book.
What this book covers

Chapter 1, Object-Oriented and Protocol-Oriented Programming, will show the differences between object-oriented programming and protocol-oriented programming, giving the reader a good understanding of the different paradigms. By the end of the chapter, the reader should have a basic understanding of protocol-oriented programming and what it offers.

Chapter 2, Our Type Choices, will look at the different types that Swift offers (structs, classes, enums, and tuples). We will show several examples of when to use the various types and when not to.

Chapter 3, Catching Our Errors, will look at the various ways we can catch and report errors. This chapter will not only cover the new Swift 2 do-try-catch blocks, but also the older error handling patterns.

Chapter 4, All about the Protocol, will be all about the protocol. We will show the advantages of using protocols and why, in our design, we should start with the protocol.

Chapter 5, Let’s Extend Some Types, will cover extensions in great detail. We will look at how to extend standard Swift types, such as structures and classes. We will also look at protocol extensions and discuss why they are so important in protocol-oriented programming.

Chapter 6, Adopting Design Patterns in Swift, will be implementing several design patterns using protocol-oriented programming. For each of the design patterns, we will look at the problem they are designed to solve and how to implement the pattern.

Chapter 7, Case Studies, will explore three case studies. This chapter is designed to pull everything from the first six chapters together to show the reader how to use protocol-oriented programming with design patterns in real-world situations.
This book is about protocol-oriented programming. When Apple announced Swift 2 at the World Wide Developers Conference (WWDC) in 2015, they also declared that Swift was the world's first protocol-oriented programming language. By its name, we may assume that protocol-oriented programming is all about the protocol; however, this would be a wrong assumption. Protocol-oriented programming is about so much more than just the protocol; it is actually a new way of not only writing applications, but also how we think about programming.

In this chapter, you will learn:

- How Swift can be used as an object-oriented programming language
- How Swift can be used as a protocol-oriented programming language
- The differences between object-oriented programming and protocol-oriented programming
- The advantages that protocol-oriented programming offers as compared to object-oriented programming

While this book is about protocol-oriented programming, we are going to start off by discussing how Swift can be used as an object-oriented programming language. Having a good understanding of object-oriented programming will help us understand protocol-oriented programming and also give us some insight into the issues protocol-oriented programming is designed to solve.
Swift as an object-oriented programming language

Object-oriented programming is a design philosophy. Writing applications with an object-oriented programming language is fundamentally different than writing applications with older procedural languages, such as C and Pascal. Procedural languages use a list of instructions to tell the computer what to do step by step by relying on procedures (or routines). Object-oriented programming, however, is all about the object. This may seem like a pretty obvious statement given the name. But essentially, when we think about object-oriented programming, we need to think about the object.

The object is a data structure that contains information about the attributes of the object in the form of properties and the actions performed by or to the object in the form of methods. Objects can be considered a thing, and in the English language, they would normally be considered nouns. These objects can be real-world or virtual objects. If you take a look around, you will see many real-world objects and, virtually, all of them can be modeled in an object-oriented way with attributes and actions.

As I am writing this chapter, I look outside and see a lake, numerous trees, grass, my dog, and the fence in our backyard. All of these items can be modeled as objects with both properties and actions.

While I am writing this chapter, I am also thinking about one of my all-time favorite energy drink. That energy drink is called Jolt. Not sure how many people remember Jolt soda or Jolt energy drink, but I would not have made it through college without them. A can of Jolt can be modeled as an object with attributes (volume, caffeine amount, temperature, and size) and actions (drinking and temperature change).

We could keep the cans of Jolt in a cooler to keep them cold. This cooler could also be modeled as an object because it has attributes (temperature, cans of Jolt, and maximum number of cans) and actions (adding and removing cans).

The object is what makes object-oriented programming so powerful. With an object, we can model real-world objects, such as a can of Jolt, or virtual objects like characters in a video game. These objects can then interact within our application to model real-world behavior or the behavior we want in our virtual world.

Within a computer application, we cannot create an object without a blueprint that tells the application what properties and actions to expect from the object. In most object-oriented languages, this blueprint comes in the form of a class. A class is a construct that allows us to encapsulate the properties and actions of an object into a single type that models the object we are trying to represent in our code.
We use initializers within our classes to create instances of the class. We usually use these initializers to set the initial values of the properties for the object or perform any other initialization that our class needs. Once we create the instance of a class, we can then use it within our code.

All of this explanation about object-oriented programming is fine, but nothing demonstrates the concepts better than the actual code. Let's see how we would use classes in Swift to model cans of Jolt and a cooler to keep our Jolt cold. We will begin by modeling the cans of Jolt as follows:

```swift
class Jolt {
    var volume: Double
    var caffeine: Double
    var temperature: Double
    var canSize: Double
    var description: String

    init(volume: Double, caffeine: Double, temperature: Double) {
        self.volume = volume
        self.caffeine = caffeine
        self.temperature = temperature
        self.description = "Jolt energy drink"
        self.canSize = 24
    }

    func drinking(amount: Double) {
        volume -= amount
    }

    func temperatureChange(change: Double) {
        temperature += change
    }
}
```

In this `Jolt` class, we defined five properties. These properties are `volume` (the amount of Jolt in the can), `caffeine` (how much caffeine comes in a can), `temperature` (the present temperature of the can), `description` (the description of the product), and `canSize` (the size of the can itself). We then define one initializer that will be used to initiate the properties of the object when we create an instance of the class. This initializer will ensure that all of the properties are properly initialized when the instance is created. Finally, we defined two actions for the can. These two actions are `drinking` (called when someone drinks from the can) and `temperatureChange` (called when the temperature of the can changes).
Now, let's see how we would model a cooler that we can use to keep our cans of Jolt cold because no one likes warm cans of Jolt:

class Cooler {
    var temperature: Double
    var cansOfJolt = [Jolt]()
    var maxCans: Int

    init(temperature: Double, maxCans: Int) {
        self.temperature = temperature
        self.maxCans = maxCans
    }

    func addJolt(jolt: Jolt) -> Bool {
        if cansOfJolt.count < maxCans {
            cansOfJolt.append(jolt)
            return true
        } else {
            return false
        }
    }

    func removeJolt() -> Jolt? {
        if cansOfJolt.count > 0 {
            return cansOfJolt.removeFirst()
        } else {
            return nil
        }
    }
}

We modeled the cooler in a similar fashion to how we modeled the cans of Jolt. We began by defining the three properties of the cooler. The three properties are temperature (the present temperature in the cooler), cansOfJolt (the cans of Jolt in the cooler), and maxCans (the maximum number of cans the cooler can hold). We then used an initializer to initiate the properties when we create the instances of the Cooler class. Finally, we defined the two actions for the cooler. They are addJolt (used to add a can of Jolt to the cooler) or removeJolt (used to remove a can of Jolt from the cooler). Now that we have our Jolt and Cooler classes, let's see how we would use these two classes together:

    var cooler = Cooler(temperature: 38.0, maxCans: 12)

    for _ in 0...5 {
        let can = Jolt(volume: 23.5, caffeine: 280,
temperature: 45)
    let _ = cooler.addJolt(can)
}

let jolt = cooler.removeJolt()
jolt?.drinking(5)
print("Jolt Left in can: \(jolt?.volume)")

In this example, we created an instance of the Cooler class using the initializer to set the default properties. We then created six instances of the Jolt class and added them to the cooler using a for-in loop. Finally, we took a can of Jolt from the cooler and drank some of it. A refreshing drink of Jolt and the jolt of caffeine. What could be better?

This design seems to work well for our simplistic example; however, it really is not that flexible. While I really like caffeine, my wife doesn't; she prefers Caffeine Free Diet Coke. With our present cooler design, when she goes to add some of her Caffeine Free Diet Coke to the cooler, we would have to tell her that it is not possible because our cooler only accepts instances of Jolt. This would not be good, because this is not the way coolers work in the real world and also because I would not want to tell my wife that she can't have her Diet Coke (trust me no one wants to tell her she can't have her Diet Coke). So, how could we make this design more flexible?

The answer to this question is polymorphism. Polymorphism comes from the Greek words Poly (for many) and Morph (forms). In computer science, we use polymorphism when we want to use a single interface to represent multiple types within our code. Polymorphism gives us the ability to interact with multiple types in a uniform manner. When we interact with multiple objects through a uniform interface, we are able to add additional object types that conform to that interface at any time. We can then use these additional types in our code with little to no changes.

With object-oriented programming languages, we can achieve polymorphism and code reuse with subclassing. Subclassing is when one class is derived from another superclass. For example, if we had a Person class that modeled a typical person, we could then subclass the Person class to create a Student class. The Student class would then inherit all of the properties and methods of the Person class. The Student class could override any of the properties and methods that it inherited and/or add its own additional properties and methods. We could then add additional classes that are also derived from the Person superclass, and we could interact with all of these subclasses using the interface presented by the Person class.
When one class is derived from another class, the original class, the one we are deriving the new class from, is known as the super or the parent class and the new class is known as the child or subclass. In our person-student example, the **Person** was the super or parent class and the **Student** was the sub or child class. In this book, we will be using the terms superclass and subclass.

Polymorphism is achieved with subclassing because we can interact with the instances of all the child classes through the interface that is presented by the superclass. As an example, if we had three child classes (**Student**, **Programmer**, and **Fireman**) that were all subclasses of the **Person** class, then we could interact with all three of the subclasses through the interface that is presented by the **Person** class. If the **Person** class had a method named `running()`, then we can be assured that all the subclasses of the **Person** class has a method named `running()` (either the method from the **Person** class or one from the subclass that overrides the method from the **Person** class). Therefore, we can interact with all the subclasses using the `running()` method.

Let's see how polymorphism can help us add drinks other than Jolt to our cooler. In our original example, we were able to hard code the can size in our **Jolt** class because Jolt energy drinks were only sold in 24 oz cans (the sodas had different sizes, but the energy drink was only sold in 24 oz cans). The following enumeration defines the can sizes that our cooler will accept:

```
enum DrinkSize {
    case Can12
    case Can16
    case Can24
    case Can32
}
```

This `DrinkSize` enumeration lets us use 12, 16, 24, and 32 oz drink sizes in our cooler.

Now, let's look at our base or superclass that all of our drink types will derive from. We will name this superclass `Drink`:

```
class Drink {
    var volume: Double
    var caffeine: Double
    var temperature: Double
    var drinkSize: DrinkSize
    var description: String

    init(volume: Double, caffeine: Double,
         temperature: Double, drinkSize: DrinkSize) {
```
This Drink class is very similar to our original Jolt class. We defined the same five properties that we had in our original Jolt class; however, drinkSize is now defined to be of the DrinkSize type rather than Double. We defined a single initializer for our Drink class that initializes all the five properties of the class. Finally, we have the same two methods that were in the original Jolt class, which are drinking() and temperatureChange(). One thing to take note of is, in the Drink class, our description is set to Drink base class.

Now, let's create the Jolt class that will be a subclass of the Drink class. This class will inherit all the property and methods from the Drink class:

```
class Jolt: Drink {
    init(temperature: Double) {
        super.init(volume: 23.5, caffeine: 280,
                    temperature: temperature, drinkSize: DrinkSize.Can24)
        self.description = "Jolt energy drink"
    }
}
```

As we can see in the Jolt class, we do not need to redefine the properties and the methods from the Drink superclass. We will add an initializer for our Jolt class. This initializer only requires that the temperature of the can of Jolt be provided. All the other values are set to their default values for a can of Jolt.

Now, let's see how we would create the Cooler class that will accept other drink types besides Jolt:

```
class Cooler {
    var temperature: Double
    var cansOfDrinks = [Drink]()
```
Object-Oriented and Protocol-Oriented Programming

```swift
var maxCans: Int

init(temperature: Double, maxCans: Int) {
    self.temperature = temperature
    self.maxCans = maxCans
}

func addDrink(drink: Drink) -> Bool {
    if cansOfDrinks.count < maxCans {
        cansOfDrinks.append(drink)
        return true
    } else {
        return false
    }
}

func removeDrink() -> Drink? {
    if cansOfDrinks.count > 0 {
        return cansOfDrinks.removeFirst()
    } else {
        return nil
    }
}

This new `Cooler` class is exactly like the original `Cooler` class, except that we replace all the references to the `Jolt` class with the references to the `Drink` class. Since the `Jolt` class is a subclass of the `Drink` class, we can use it any place where an instance of the `Drink` class is required. Let's see how this would work. The following code will create an instance of the `Cooler` class. Add six cans of Jolt to the cooler, remove one of the cans from the cooler, and then take a drink of Jolt:

```swift
var cooler = Cooler(temperature: 38.0, maxCans: 24)
for _ in 0...5 {
    let can = Jolt(temperature: 45.1)
    let _ = cooler.addDrink(can)
}

let jolt = cooler.removeDrink()
cooler.cansOfDrinks.count
jolt?.drinking(5)
print("Jolt Left in can: \(jolt?.volume")")
```
Notice that in this example, we used instances of the \texttt{Jolt} class where instances of the \texttt{Drink} class are required. This is polymorphism in action. Now that we have a cooler with our Jolt in it, we are ready to go on a trip. My wife of course wants to bring her Caffeine Free Diet Coke so she asks if she can put some in the cooler to keep it cold. Knowing that we do not want to deprive her of Diet Coke, we quickly create a \texttt{CaffeineFreeDietCoke} class that we can use with the cooler. The code for this class is:

```swift
class CaffeineFreeDietCoke: Drink {
    init(volume: Double, temperature: Double, drinkSize: DrinkSize) {
        super.init(volume: volume, caffeine: 0, temperature: temperature, drinkSize: drinkSize)
        self.description = "Caffeine Free Diet Coke"
    }
}
```

The \texttt{CaffeineFreeDietCoke} class is very similar to the \texttt{Jolt} class. They both are subclasses of the \texttt{Drink} class, and they each define an initializer that initializes the class. The key is that they both are subclasses of the \texttt{Drink} class, which means we can use instances of both classes in our cooler. Therefore, when my wife brings her six Caffeine Free Diet Cokes, we can put them in the cooler just like the cans of Jolt. The following code demonstrates this:

```swift
var cooler = Cooler(temperature: 38.0, maxCans: 24)
for _ in 0...5 {
    let can = Jolt(temperature: 45.1)
    let _ = cooler.addDrink(can)
}
for _ in 0...5 {
    let can = CaffeineFreeDietCoke(volume: 15.5, temperature: 45, drinkSize: DrinkSize.Can16)
    let _ = cooler.addDrink(can)
}
```

In this example, we created an instance of the cooler; we added six cans of Jolt and six cans of Caffeine Free Diet Coke to it. Using polymorphism, as shown here, allows us to create as many subclasses of the \texttt{Drink} class as we need, and all of them can be used with the \texttt{Cooler} class without changing the code of the \texttt{Cooler} class. This allows our code to be extremely flexible.
So, what happens when we grab a can from the cooler? Obviously, if my wife grabs a can of Jolt, she will want to put it back and get a different can. But how will she know which can she grabbed?

To check whether an instance is of a particular type, we use the type check operator (is). The type check operator will return true if the instance is of the type or false if it isn't. In the following code, we use the type check operator to continuously remove cans from the cooler until we find a can of Caffeine Free Diet Coke:

```swift
var foundCan = false
var wifeDrink: Drink?

while !foundCan {
    if let can = cooler.removeDrink() {
        if can is CaffeineFreeDietCoke {
            foundCan = true
            wifeDrink = can
        } else {
            cooler.addDrink(can)
        }
    }
}

if let drink = wifeDrink {
    print("Got: " + drink.description)
}
```

In this code, we have a while loop that continuously loops until the foundCan Boolean is set to true. Within the while loop, we remove a drink from the cooler and then use the type check operator (is) to see whether the can that we removed is an instance of the CaffeineFreeDietCoke class. If it is an instance of the CaffeineFreeDietCoke class, then we will set the foundCan Boolean to true and set the wifeDrink variable to the instance of the drink we just removed from the cooler. If the drink is not an instance of the CaffeineFreeDietCoke class, then we will put the can back in the cooler and loop back to grab another drink.

In the previous example, we showed how Swift can be used as an object-oriented programming language. We also used polymorphism to make our code very flexible and easy to expand; however, there are several drawbacks to this design. Before we move on to protocol-oriented programming, let's take a look at two of these drawbacks. Then, we will see how protocol-oriented programming can be used to make this design better.
The first drawback of our design is the initializers of the drink (Jolt, CaffeineFreeDietCoke, and DietCoke) classes. When we initialize a subclass, we need to call the initializer of the superclass. This is a double-edged sword. While calling the initializer of our superclass gives us consistent initialization, it can also give us improper initialization if we are not careful. For example, let's say that we created another Drink class named DietCoke with the following code:

```swift
class DietCoke: Drink {
    init(volume: Double, temperature: Double, drinkSize: DrinkSize) {
        super.init(volume: volume, caffeine: 45,
                   temperature: temperature, drinkSize: drinkSize)
    }
}
```

If we look carefully, we will see that in the initializer of the DietCoke class, we never set the description property. Therefore, the description of this class will end up being Drink base class, which is not what we want.

We need to be careful when we create subclasses like this to ensure that all of the properties are properly set and we don't just assume that the initializer of the superclass will properly set all of the properties for us.

The second drawback to our design is we are using reference types. While those who are familiar with object-oriented programming may not see this as a drawback and reference types are preferred in a lot of cases, in our design, it makes more sense to define the drink types as value types. If you are not familiar with how reference and value types work, we will be looking at them in depth in Chapter 2, Our Type Choices.

When we pass an instance of a reference type (that is, we pass to a function or a set in a collection like an array), we are passing a reference to the original instance. When we pass an instance of a value type, we are passing a new copy of the original instance. Let's see the issue that using reference types can cause if we are not careful by examining the following code:

```swift
var jolts = [Drink]()

var myJolt = Jolt(temperature: 48)
for _ in 0...5 {
    jolts.append(myJolt)
}

jolts[0].drinking(10)

for (index,can) in jolts.enumerate(){
    print("Can \(index) amount Left: \(can.volume)")
}
```
In this example, we created an array that will contain instances of the `Drink` class or instances of a type that is a subclass of the `Drink` class. We then created an instance of the `Jolt` class and used it to populate our array with six cans of Jolt. Next, we took a drink from the first can in our array and printed out the remaining volume of each can in our array. If we run this code, we would see the following results:

- Can 0 amount Left: 13.5
- Can 1 amount Left: 13.5
- Can 2 amount Left: 13.5
- Can 3 amount Left: 13.5
- Can 4 amount Left: 13.5
- Can 5 amount Left: 13.5

As we can see from the results, all of the cans in the array have the same amount of Jolt remaining. This is because we created a single instance of the `Jolt` class and then, to the `jolts` array, we added six references to this single instance. Therefore, when we took a drink from the first can in the array, we actually took a drink from all of the cans in the array.

A mistake like this to an experienced object-oriented programmer may seem out of the question; however, it's amazing how often it occurs with junior developers or developers who are not familiar with object-oriented programming. This error occurs more often with classes that have complex initializers. We can avoid this issue by using the Builder pattern that we will see in Chapter 6, Adopting Design Patterns in Swift or by implementing a copy method in our custom class that will make a copy of an instance.

One other thing to note about object-oriented programming and subclassing, as shown in the previous example, is that a class can only have one superclass. For example, the superclass for our `Jolt` class is the `Drink` class. This can lead to a single superclass that is very bloated and contains code that is not needed or wanted by all the subclasses. This is a very common problem in game development.

Now, let's look at how we would implement our drinks and cooler example using protocol-oriented programming.
Chapter 1

Swift as a protocol-oriented programming language

With object-oriented programming, we usually begin our design by thinking about the objects and the class hierarchy. Protocol-oriented programming is a little different. Here, we begin our design by thinking about the protocols. However, as we stated at the beginning of this chapter, protocol-orientated programming is about so much more than just the protocol.

As we go through this section, we will briefly discuss the different items that make up protocol-oriented programming with regards to our current example. We will then discuss these items in depth over the next couple of chapters to give you a better understanding of how to use protocol-oriented programming as a whole in our applications.

In the previous section, when we looked at Swift as an object-oriented programming language, we designed our solution with a class hierarchy, as shown in the following diagram:

![Diagram of class hierarchy](image)

To redesign this solution with protocol-oriented programming, we would need to rethink a couple areas of this design. The first area that we would want to rethink is the Drink class. Protocol-oriented programming states that we should begin with a protocol rather than a superclass. This means that our Drink class would become a Drink protocol. We would then use protocol extensions to add common code for our drink types that will conform to this protocol. We will go over the protocols in Chapter 4, All about the Protocol, and we will cover the protocol extensions in Chapter 5, Let’s Extend Some Types.
The second area that we would want to rethink is the use of reference (class) types. With Swift, Apple has stated that it is preferable to use value types over reference types where appropriate. There is a lot to consider when we decide whether to use reference or value types, and we will go over this in depth in Chapter 2, Our Type Choices. In this example, we will use value (structure) types for our drink types (Jolt and CaffeineFreeDietCoke) and a reference (class) type for our Cooler type.

The decision to use value types for our drink types and a reference type for our Cooler type, in this example, is based on how we would use the instances of these types. The instance of our drink types will only have one owner. For example, when a drink is in the cooler, the cooler owns it. But then, when a person takes the drink out, the drink is removed from the cooler and given to a person who would then own it.

The Cooler type is a little different from the drink types. While the drink types will have only one owner interacting with it at a time, instances of the Cooler type may have several parts of our code interacting with it. For example, we may have one part of our code adding drinks to the cooler while we have instances of several people taking drinks from the cooler.

To summarize it, we use a value type (structure) to model our drink types because only one part of our code should be interacting with an instance of the drinks type at any one time. However, we use a reference type (class) to model our cooler because multiple parts of our code will be interacting with the same instance of the Cooler type.

We are going to stress this many times in this book: one of the main differences between reference and values types is how we pass the instances of the type. When we pass an instance of a reference type, we are passing a reference to the original instance. This means that the changes made are reflected in both the references. When we pass an instance of a value type, we are passing a new copy of the original instance. This means that the changes made in one instance are not reflected in the other.

Before we examine protocol-oriented programming further, let's take a look at how we would rewrite our example in a protocol-oriented programming manner. We will start by creating our Drink Protocol:

```swift
protocol Drink {
    var volume: Double {get set}
    var caffeine: Double {get set}
    var temperature: Double {get set}
    var drinkSize: DrinkSize {get set}
    var description: String {get set}
}
```
Within our `Drink` protocol, we defined the five properties every type that conforms to this protocol must provide. The `DrinkSize` type is the same `DrinkSize` type that we defined in the object-oriented section of this chapter.

Before we add any types that conform to our `Drink` protocol, we want to extend the protocol. Protocol extensions were added to the Swift language in version 2, and they allow us to provide functionality to conforming types. This lets us define the behavior for all types that conform to a protocol rather than adding the behavior to each individual conforming type. Within the extension for our `Drink` protocol, we will define two methods: `drinking()` and `temperatureChange()`. These are the same two methods that were in our `Drink` superclass in the object-oriented programming section of this chapter. Following is the code for our `Drink` extension:

```swift
extension Drink {
    mutating func drinking(amount: Double) {
        volume -= amount
    }
    mutating func temperatureChange(change: Double) {
        temperature += change
    }
}
```

Now, any type that conforms to the `Drink` protocol will automatically receive the `drinking()` and the `temperatureChange()` methods. Protocol extensions are perfect for adding common functionality to all the types that conform to a protocol. This is similar to adding functionality to a superclass where all subclasses receive the functionality from the superclass. The individual types that conform to a protocol can also shadow any functionality provided by an extension similar to overriding functionality from a superclass.

Now let's create our `Jolt` and `CaffeineFreeDietCoke` types:

```swift
struct Jolt: Drink {
    var volume: Double
    var caffeine: Double
    var temperature: Double
    var drinkSize: DrinkSize
    var description: String

    init(temperature: Double) {
        self.volume = 23.5
        self.caffeine = 280
        self.temperature = temperature
        self.description = "Jolt Energy Drink"
        self.drinkSize = DrinkSize.Can24
    }
}
```
As we can see, both the Jolt and CaffeineFreeDietCoke types are structures rather than classes. This means that they are both value types rather than reference types, as they were in the object-oriented design. Both of the types implement the five properties that are defined in the Drink protocol as well as an initializer that will be used to initialize the instances of the types.

There is more code needed in these types as compared to the drink classes in the object-oriented example. However, it is easier to understand what is going on in these drink types because everything is being initialized within the type itself rather than in a superclass.

Finally, let's look at the cooler type:

```swift
class Cooler {
    var temperature: Double
    var cansOfDrinks = [Drink]()
    var maxCans: Int

    init(temperature: Double, maxCans: Int) {
        self.temperature = temperature
        self.maxCans = maxCans
    }
}
```
As we can see, the Cooler class is the same class that we created in the Object-oriented programming section of this chapter. There could be a very viable argument for creating the Cooler type as a structure rather than a class, but it really depends on how we plan to use it in our code. Earlier, we stated that various parts of our code will need to interact with a single instance of our cooler. Therefore, in our example, it is better to implement our cooler as a reference type rather than a value type.

Apple’s recommendation is to prefer value types over reference types where appropriate. Therefore, when in doubt, it is recommended that we go with a value type rather than a reference type.

The following diagram shows how the new design looks:
Now that we have finished redesigning, let's summarize what protocol-oriented programming is and how it is different from object-oriented programming.

**Summarizing protocol-oriented programming and object-oriented programming**

We just saw how Swift can be used as both an object-oriented programming language and a protocol-oriented programming language, and what is the real difference between the two. In the example presented in this chapter, there were two major differences between the two designs.

The first difference that we saw with protocol-oriented programming is that we should start with a protocol rather than a superclass. We can then use protocol extensions to add functionality to the types that conform to that protocol. With object-oriented programming, we started with a superclass. When we redesigned our example, we converted the `Drink` superclass to a `Drink` protocol and then used a protocol extension to add the `drinking()` and `temperatureChange()` methods.

The second real difference that we saw was the use of value types (structures) rather than reference types (class) for our drink types. Apple has said that we should prefer value types over reference types where appropriate. In our example, it was appropriate to use value types when we implemented our drink types; however, we still implemented the `Cooler` type as a reference type.

Mixing and matching value and reference types may not be the best approach for the long-term maintainability of our code. We used it in our example to stress the difference between the value and reference types. In Chapter 2, *Our Type Choices*, we will discuss this in detail.

Both the object-oriented design and the protocol-oriented design used polymorphism to let us interact with different types using the same interface. With the object-oriented design, we used the interface provided by the superclass to interact with all the subclasses. In the protocol-oriented design, we used the interface provided by the protocol and the protocol extension to interact with the types that conform to the protocol.

Now that we have summarized the differences between the object-oriented programming design and the protocol-oriented programming design, let's take a closer look at these differences.
Object-oriented programming versus protocol-oriented programming

I mentioned at the beginning of the chapter that protocol-oriented programming is about so much more than just the protocol and that it is a new way of not only writing applications but also thinking about programming. In this section, we will examine the difference between our two designs to see what that statement actually means.

As a developer, our primary goal is always to develop an application that works properly, but we should be focused on writing clean and safe code as well. In this section, we will be talking about clean and safe code a lot, so let's look at what we mean by these terms.

Clean code is code that is very easy to read and understand. It is important to write clean code because any code that we write will need to be maintained by someone and that someone is usually the person who wrote it. There is nothing worse than looking back at code you wrote and not being able to understand what it does. It is also a lot easier to find errors in the code that is clean and easy to understand.

By safe code we mean code that is hard to break. There is nothing more frustrating as a developer than to make a small change in our code and have errors pop up throughout the code base or to have numerous bugs pop up within our application. By writing clean code, our code will be inherently safer because other developers will be able to look at the code and understand exactly what it does.

Now, let's briefly look at the difference between protocols / protocol extensions and superclasses. We will be covering this a lot more in Chapter 4, All about the Protocol, and Chapter 5, Let's Extend Some Types.

Protocol and protocol extensions versus superclasses

In the object-oriented programming example, we created a Drink superclass from which all of the drink classes were derived. In the protocol-oriented programming example, we used a combination of a protocol and a protocol extension to achieve the same results; however, there are several advantages of using protocols.

To refresh our memory of the two solutions, let's look at the code for both the Drink superclass and the Drink protocol and protocol extension. The following code shows the Drink superclass:

```swift
class Drink {
    var volume: Double
```
The `Drink` superclass is a complete type that we can create instances of. This can be a good or a bad thing. There are times, like in this example, when we should not be creating instances of the superclass; we should only be creating instances of the subclasses. For this, we can still use protocols with object-oriented programming; however, we will need to use protocol extensions to add the common functionality that will then lead us down the protocol-oriented programming path.

Now, let's look at how we would use protocol-oriented programming with the `Drink` protocol and the `Drink` protocol extension:

```swift
protocol Drink {
    var volume: Double {get set}
    var caffeine: Double {get set}
    var temperature: Double {get set}
    var drinkSize: DrinkSize {get set}
    var description: String {get set}
}

extension Drink {
    mutating func drinking(amount: Double) {
        volume -= amount
    }
    func temperatureChange(change: Double) {
        temperature += change
    }
}
```

---

[20]
mutating func temperatureChange(change: Double) {
    temperature += change
}

The code in both the solutions is pretty safe and easy to understand. As a personal preference, I like separating the implementation from the definition. Therefore, to me, the protocol / protocol extension code is better, but this really is a matter of preference. However, we will see in the next few pages that the protocol / protocol extension solution as a whole is cleaner and easier to understand.

There are three other advantages that protocols / protocol extensions have over superclasses. The first advantage is that types can conform to multiple protocols; however, they can only have one superclass. What this means is that we can create numerous protocols that add very specific functionality rather than creating a single monolithic superclass. For example, with our Drinks protocol, we could also create the DietDrink, SodaDrink, and EnergyDrink protocols that contain specific requirements and functionality for these types of drinks. Then, the DietCoke and CaffeineFreeDietCoke types would conform to the Drink, DietDrink, and SodaDrink protocols, while the Jolt structure would conform to the Drink and EnergyDrink protocols. With a superclass, we would need to combine the functionality defined in the DietDrink, SodaDrink, and EnergyDrink protocols into the single monolithic superclass.

The second advantage that protocol / protocol extensions have is that we can use protocol extensions to add functionality without needing the original code. What this means is that we can extend any protocol, even the protocols that are a part of the Swift language itself. To add functionality to our superclass, we need to have the original code. We could use extensions to add functionality to a superclass, which means that all the subclass will also inherit that functionality. However, generally we use extensions to add functionality to a specific class rather than adding functionality to a class hierarchy.

The third advantage that protocols / protocol extensions have is that protocols can be adopted by classes, structures, and enumerations, while class hierarchies are restricted to class types. Protocols / protocol extensions give us the option to use value types where appropriate.
Implementing drink types

The implementation of drink types (the Jolt and CaffeineFreeDietCoke types) was significantly different between the object-oriented example and the protocol-oriented example. We will look at the differences between these two examples, but first let's take a look at the code again to remind us about how we implemented the drink types. We will look at how we implemented the drink types in the object-oriented example first:

```swift
class Jolt: Drink {
    init(temperature: Double) {
        super.init(volume: 23.5, caffeine: 280,
                   temperature: temperature, drinkSize: DrinkSize.Can24)
        self.description = "Jolt energy drink"
    }
}

class CaffeineFreeDietCoke: Drink {
    init(volume: Double, temperature: Double, drinkSize: DrinkSize) {
        super.init(volume: volume, caffeine: 0,
                   temperature: temperature, drinkSize: drinkSize)
        self.description = "Caffeine Free Diet Coke"
    }
}
```

Both of these classes are subclasses of the Drink superclass and both implement a single initializer. While these are pretty simple and straightforward implementations, we really need to fully understand what the superclass expects to implement them properly. For example, if we do not fully understand the Drink superclass, we may forget to set the description properly. In our example, forgetting to set the description may not be that big of an issue, but in more complex types, forgetting to set a property may cause very unexpected behavior. We could prevent these mistakes by setting all the properties in the superclass's initializer; however, this may not be possible in some situations.

Now, let's look at how we implemented the drink types in the protocol-oriented programming example:

```swift
struct Jolt: Drink {
    var volume: Double
    var caffeine: Double
```
var temperature: Double
var drinkSize: DrinkSize
var description: String

init(temperature: Double) {
    self.volume = 23.5
    self.caffeine = 280
    self.temperature = temperature
    self.description = "Jolt Energy Drink"
    self.drinkSize = DrinkSize.Can24
}

}

struct CaffeineFreeDietCoke: Drink {
    var volume: Double
    var caffeine: Double
    var temperature: Double
    var drinkSize: DrinkSize
    var description: String

    init(volume: Double, temperature: Double,
         drinkSize: DrinkSize) {
        self.volume = volume
        self.caffeine = 0
        self.temperature = temperature
        self.description = "Caffiene Free Diet Coke"
        self.drinkSize = drinkSize
    }
}

Implementing the types in the protocol-oriented programming example takes significantly more code than the object-oriented programming example; however, the code in the protocol-oriented example is a lot safer and easier to understand. The reason we say the protocol-oriented example is safer and easier to understand is how the properties and initializer are implemented in both the examples.

In the object-oriented programming example, all of the properties are defined in the superclass. We will need to look at the code or the documentation for the superclass to see what properties are defined and how they are defined. With protocols, we also need to look at the protocol itself or the documentation for the protocol to see what properties to implement, but the implementation is done in the type itself. This allows us to see how everything is implemented for the type without having to look back at the code for the superclass or having to dig through a complete class hierarchy to see how things are implemented and initialized.
The initializer in the subclass must also call the initializer in the superclass to ensure that all the properties of the superclass are set properly. While this does ensure that we have a consistent initialization between the subclasses, it also hides how the class is initialized. With the protocol example, all the initialization is done within the type itself. Therefore, we do not have to dig through a class hierarchy to see how everything is initialized.

Superclasses in Swift provide an implementation of our requirements. Protocols in Swift are simply a contract that says any type that conforms to a given protocol must fulfill the requirements specified by the protocol. Therefore, with protocols, all of the properties, methods, and initializers are defined in the conforming types themselves. This allows us to very easily see how everything is defined and initialized.

Value types vs reference types

There are several fundamental differences between reference and value types, and we will discuss these in greater detail in Chapter 2, Our Type Choices. Right now, we will focus on one of the main differences between the two types and that is how the types are passed. When we pass an instance of a reference type (class), we are passing a reference to the original instance. This means that any changes made are reflected back to the original instance. When we pass an instance of a value type, we are passing a new copy of the original instance. This means any changes made are not reflected back to the original instance.

As we mentioned earlier, in our example, an instance of the drink types should only have one owner at a time. There should never be a need for multiple parts of our code to interact with a single instance of a drink type. As an example, when we create an instance of a drink type, we will put it in an instance of the cooler type. Then, if a person comes along and removes that instance from the cooler, the person will own that drink instance. If one person gives the drink to another person, then that second person will own the drink.

Using value types ensures that we always get a unique instance since we pass a copy of the original instances rather than a reference to the original instance. Therefore, we can trust that no other part of our code is going to unexpectedly change that instance on us. This is especially helpful with multithreaded environments, where a different thread can alter the data and create unexpected behavior.

We need to make sure that we use value types and reference types appropriately. In this example, the drink types illustrated when value types should be preferred and the Cooler type illustrated when a reference type should be preferred.
In most object-oriented languages, we do not have the option of implementing our custom types as value types. In Swift, classes and structures are much closer in functionality than other languages and we have the option of creating custom types as value types. We just need to make sure we use the appropriate type when we create our custom types. We will discuss these options in greater detail in Chapter 2, Our Type Choices.

**The winner is...**

As we were reading through this chapter and seeing all of the advantages that protocol-oriented programming has over object-oriented programming, we may think that protocol-oriented programming is clearly superior to object-oriented programming. However, this assumption may not be totally correct.

Object-oriented programming has been around since the 1970s and is a tried and true programming paradigm. Protocol-oriented programming is the new kid on the block and was designed to correct some of the issues with object-oriented programming. I have personally used the protocol-oriented programming paradigm in a couple of projects and I am very excited about its possibilities.

Object-oriented programming and protocol-oriented programming have similar philosophies like creating custom types that model real-world objects and polymorphism to use a single interface to interact with multiple types. The difference is how these philosophies are implemented.

To me, the code base in a project that uses protocol-oriented programming is much safer and easier to read as compared to a project that uses object-oriented programming. This does not mean that I am going to stop using object-oriented programming altogether. I can still see plenty of need for class hierarchy and inheritance.

Remember, when we are designing our application, we should always use the right tool for the right job. We would not want to use a chainsaw to cut a piece of 2 x 4 lumber, but we also would not want to use a skilsaw to cut down a tree. Therefore, the winner is the programmer, where we have the choice of using different programming paradigms rather than being limited to only one.
Summary

In this chapter, we saw how Swift can be used as an object-oriented programming language and also saw how it can be used as a protocol-oriented programming language. While these two programming paradigms have similar philosophies, they implement these philosophies differently.

With object-oriented programming, we would use classes as our blueprints when we create objects. With protocol-oriented programming, we have the choice of using classes, structures, and enumerations. We can even use other types, as we will see in Chapter 2, Our Type Choices.

With object-oriented programming, we can implement polymorphism using class hierarchies. With protocol-oriented programming, we use a combination of protocols and protocol extensions to implement polymorphism. We will look at protocols in depth in Chapter 4, All about the Protocol.

With object-oriented programming, we are able to implement functionality in our superclasses that is inherited by the subclasses. The subclasses do have the ability to override the functionality provided by the superclass. With protocol-oriented programming, we use protocol extensions to add functionality to types that conform to our protocols. These types can also shadow this functionality if they choose to. We will look at protocol extensions in depth in Chapter 5, Let’s Extend Some Types.

While object-oriented programming has been around since the 1970s and it is a tried and true programming paradigm, it is also beginning to show some wear and tear. In this chapter, we looked at the problems and design issues that protocol-oriented programming was designed to solve.

Now that we have seen an overview of protocol-oriented programming, it is time to look at each area that makes up protocol-oriented programming in greater detail. By getting a better understanding of the different areas, we will be able to better implement protocol-oriented programming in our applications. We will start off by looking at the various type choices that we have with the Swift programming language and how we should use each of them.
Where to buy this book

You can buy Protocol-Oriented Programming with Swift from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.