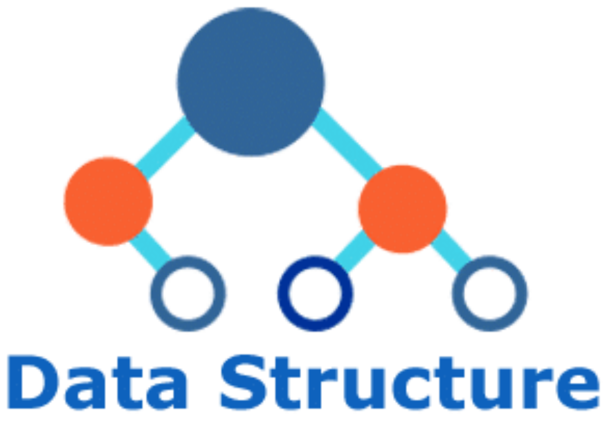
CHAPTER SEVEN : OOP AND DATA STRUCTURES IN SWIFT



I.INTRODUCTION

As is demonstrated in the previous chapters, **Object Oriented Programming(OOP)** is widely used in iOS development in Swift. When you work on a large project, using OOP is a great way to make your code easier to implement and manage. Besides, the framework for iOS applications are provided based on the OOP way.

Data Structure is another very important concept in all software development. You have already used two very commonly used data structures: **arrays** and **maps**. In this chapter , you will see other common data structures like **enumerations** and **structures**.

We will also learn protocols, extensions, and generics. These are great features that go alongside OOP. And we will learn briefly the error handling mechanism.

II. OBJECTS AND CLASSES

Use `class` followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class. Likewise, method and function declarations are written the same way.

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

EXPERIMENT 1

Add a constant property with `let`, and add another method that takes an argument.

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

This version of the Shape class is missing something important: an initializer to set up the class when an instance is created. Use init to create one.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Notice how self is used to distinguish the name property from the name argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned—either in its declaration (as with numberOfSides) or in the initializer (as with name).

Use **deinit** to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There is no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's implementation are marked with override—overriding a method by accident, without override, is detected by the compiler as an error. The compiler also detects methods with override that don't actually override any method in the superclass.

```
class Square: NamedShape {
    var sideLength: Double
```

```
        init(sideLength: Double, name: String) {
            self.sideLength = sideLength
            super.init(name: name)
            numberOfSides = 4
        }

        func area() -> Double {
            return sideLength * sideLength
        }

        override func simpleDescription() -> String {
            return "A square with sides of length \(sideLength)."
        }
    }
    let test = Square(sideLength: 5.2, name: "my test square")
    test.area()
    test.simpleDescription()
```

EXPERIMENT 2

Make another subclass of `NamedShape` called `Circle` that takes a radius and a name as arguments to its initializer. Implement an `area()` and a `simpleDescription()` method on the `Circle` class.

In addition to simple properties that are stored, properties can have a getter and a setter.

```
    class EquilateralTriangle: NamedShape {
        var sideLength: Double = 0.0

        init(sideLength: Double, name: String) {
            self.sideLength = sideLength
            super.init(name: name)
            numberOfSides = 3
        }

        var perimeter: Double {
            get {
                return 3.0 * sideLength
            }
            set {
                sideLength = newValue / 3.0
            }
        }

        override func simpleDescription() -> String {
            return "An equilateral triangle with sides of length \
(sideLength)."
        }
    }
    var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
    print(triangle.perimeter)
    // Prints "9.3"
    triangle.perimeter = 9.9
    print(triangle.sideLength)
    // Prints “3.3000000000000003”
```

In the setter for **`perimeter`**, the new value has the implicit name **`newValue`**. You can provide an explicit name in parentheses after `set`.

Notice that the initializer for the **`EquilateralTriangle`** class has three different steps:

1. Setting the value of properties that the subclass declares.

2. Calling the superclass's initializer.

3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that is run before and after setting a new value, use **`willSet`** and **`didSet`**. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
    class TriangleAndSquare {
        var triangle: EquilateralTriangle {
            willSet {
                square.sideLength = newValue.sideLength
            }
        }
        var square: Square {
            willSet {
                triangle.sideLength = newValue.sideLength
            }
        }
        init(size: Double, name: String) {
            square = Square(sideLength: size, name: name)
            triangle = EquilateralTriangle(sideLength: size, name: name)
        }
    }
    var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test
shape")
    print(triangleAndSquare.square.sideLength)
    // Prints "10.0"
    print(triangleAndSquare.triangle.sideLength)
    // Prints "10.0"
    triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
    print(triangleAndSquare.triangle.sideLength)
    // Prints “50.0"
```

When working with optional values, you can write `?` before operations like methods, properties, and subscripting. If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

```
    let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
square")
    let sideLength = optionalSquare?.sideLength
```

III. ENUMERATIONS AND STRUCTURES

Use **enum** to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
    enum Rank: Int {
        case ace = 1
        case two, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king

        func simpleDescription() -> String {
            switch self {
            case .ace:
                return "ace"
            case .jack:
                return "jack"
            case .queen:
                return "queen"
            case .king:
                return "king"
            default:
                return String(self.rawValue)
            }
        }
    }
    let ace = Rank.ace
    let aceRawValue = ace.rawValue
```

EXPERIMENT
Write a function that compares two `Rank` values by comparing their raw values.

By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, `Ace` is explicitly given a raw value of `1`, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the `rawValue` property to access the raw value of an enumeration case.

Use the `init?(rawValue:)` initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or `nil` if there is no matching `Rank`.

```
    if let convertedRank = Rank(rawValue: 3) {
```

```
    let threeDescription = convertedRank.simpleDescription()
}
```

The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

EXPERIMENT

Add a `color()` method to `Suit` that returns "black" for spades and clubs, and returns "red" for hearts and diamonds.

Notice the two ways that the hearts case of the enumeration is referred to above: When assigning a value to the hearts constant, the enumeration case Suit.hearts is referred to by its full name because the constant doesn't have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form .hearts because the value of self is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

```
enum ServerResponse {
    case result(String, String)
```

```
        case failure(String)
    }

    let success = ServerResponse.result("6:00 am", "8:09 pm")
    let failure = ServerResponse.failure("Out of cheese.")

    switch success {
    case let .result(sunrise, sunset):
        print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
    case let .failure(message):
        print("Failure...  \(message)")
    }
    // Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm.”
```

EXPERIMENT
Add a third case to `ServerResponse` and to the switch.

Notice how the sunrise and sunset times are extracted from the ServerResponse value as part of matching the value against the switch cases.

Use struct to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they are passed around in your code, but classes are passed by reference.

```
    struct Card {
        var rank: Rank
        var suit: Suit
        func simpleDescription() -> String {
            return "The \(rank.simpleDescription()) of \
(suit.simpleDescription())"
        }
    }
    let threeOfSpades = Card(rank: .three, suit: .spades)
    let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

EXPERIMENT
Write a function that returns an array containing a full deck of cards, with one card of each combination of rank and suit.

IV. PROTOCOLS AND EXTENSIONS

Use `protocol` to declare a protocol.

```
    protocol ExampleProtocol {
        var simpleDescription: String { get }
        mutating func adjust()
    }
```

Classes, enumerations, and structs can all adopt protocols.

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

EXPERIMENT

Add another requirement to `ExampleProtocol`. What changes do you need to make to `SimpleClass` and `SimpleStructure` so that they still conform to the protocol?

Notice the use of the **mutating** keyword in the declaration of **SimpleStructure** to mark a method that modifies the structure. The declaration of **SimpleClass** doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Use **extension** to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that is declared elsewhere, or even to a type that you imported from a library or framework.

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
print(7.simpleDescription)
// Prints "The number 7"
```

EXPERIMENT

Write an extension for the `Double` type that adds an `absoluteValue` property.

You can use a protocol name just like any other named type—for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition are not available.

```
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// Prints "A very simple class.  Now 100% adjusted."
// print(protocolValue.anotherProperty)  // Uncomment to see the error
```

Even though the variable **protocolValue** has a runtime type of **SimpleClass**, the compiler treats it as the given type of **ExampleProtocol**. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.

V. ERROR HANDLING

You represent errors using any type that adopts the **Error** protocol.

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

Use **throw** to throw an error and **throws** to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use **do-catch**. Inside the do block, you mark code that can throw an error by writing **try** in front of it. Inside the **catch** block, the error is automatically given the name **error** unless you give it a different name.

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
// Prints "Job sent”
```

EXPERIMENT

Change the printer name to `"Never Has Toner"`, so that the `send(job:toPrinter:)` function throws an error.

You can provide multiple **catch** blocks that handle specific errors. You write a pattern after **catch** just as you do after case in a switch.

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
// Prints "Job sent”
```

EXPERIMENT

Add code to throw an error inside the `do` block. What kind of error do you need to throw so that the error is handled by the first `catch` block? What about the second and third blocks?

Another way to handle errors is to use **try?** to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is **nil**. Otherwise, the result is an optional containing the value that the function returned.

```
let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner”)
```

Use **defer** to write a block of code that is executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use defer to write setup and cleanup code next to each other, even though they need to be executed at different times.

```
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
```

```
    print(fridgeIsOpen)
    // Prints “false"
```

VI. GENERICS

Write a name inside angle brackets to make a generic function or type.

```
    func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {
        var result = [Item]()
        for _ in 0..<numberOfTimes {
            result.append(item)
        }
        return result
    }
    makeArray(repeating: "knock", numberOfTimes: 4)
```

You can make generic forms of functions and methods, as well as classes, enumerations, and structures.

```
    // Reimplement the Swift standard library's optional type
    enum OptionalValue<Wrapped> {
        case none
        case some(Wrapped)
    }
    var possibleInteger: OptionalValue<Int> = .none
    possibleInteger = .some(100)
```

Use **where** right before the body to specify a list of requirements—for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.

```
    func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> 
Bool
        where T.Element: Equatable, T.Element == U.Element
    {
        for lhsItem in lhs {
            for rhsItem in rhs {
                if lhsItem == rhsItem {
                    return true
                }
            }
        }
        return false
    }
    anyCommonElements([1, 2, 3], [3])
```

EXPERIMENT

Modify the `anyCommonElements(_:_:)` function to make a function that returns an array of the elements that any two sequences have in common.

Writing **<T: Equatable>** is the same as writing **<T> ... where T: Equatable**.

SUMMARY

In this chapter, we have worked on these topics:

- define classes and create objects
- define and use enumerations and structures
- define protocols to organize classes
- add extensions to classes
- handle errors
- define generic classes

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/MyPlayground_completed.playground.zip