CHAPTER FIVE : DOCUMENTATION AND BASIC SWIFT

I.OVERVIEW OF VIEWCONTROLLER.SWIFT

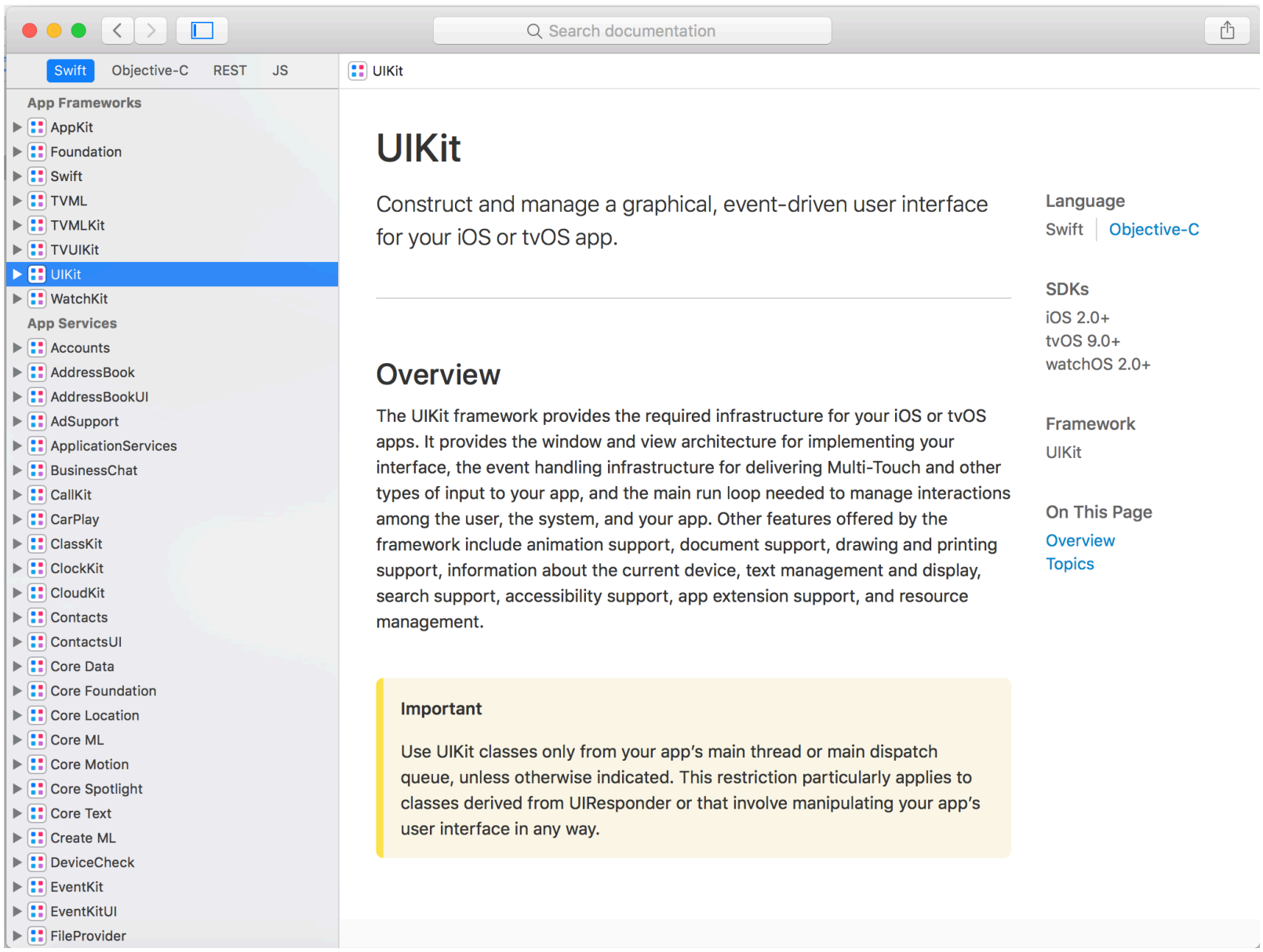
The code that we added to the `ViewController.swift` might be a little bit intimidating. Let's take a closer look at it.

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    @IBAction func showMessage(_ sender: UIButton) {
        // Create an alert controller object to display an alert
        let alertController = UIAlertController(title: "Welcome to my hello world app", message:
"Hello World!", preferredStyle: .alert)
        // Add an action OK to the alert controller
        alertController.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
        // Present the alert controller with animation
        present(alertController, animated: true, completion: nil)
    }

}
```

While learning Swift(or any programming language), reading the official document can be really helpful. In the first line, we have:

```
    import UIKit
```

The `import` keyword imports the external framework. The UIKit framework provides a set of built-in functions for graphics operations that we need in our `ViewController.swift`. If you press option and hover your mouse over the name **UIKit**, you will see a question mark instead of a mouse arrow. Click it and choose **Search Documentation**, you will go to the **Documentation Viewer**.

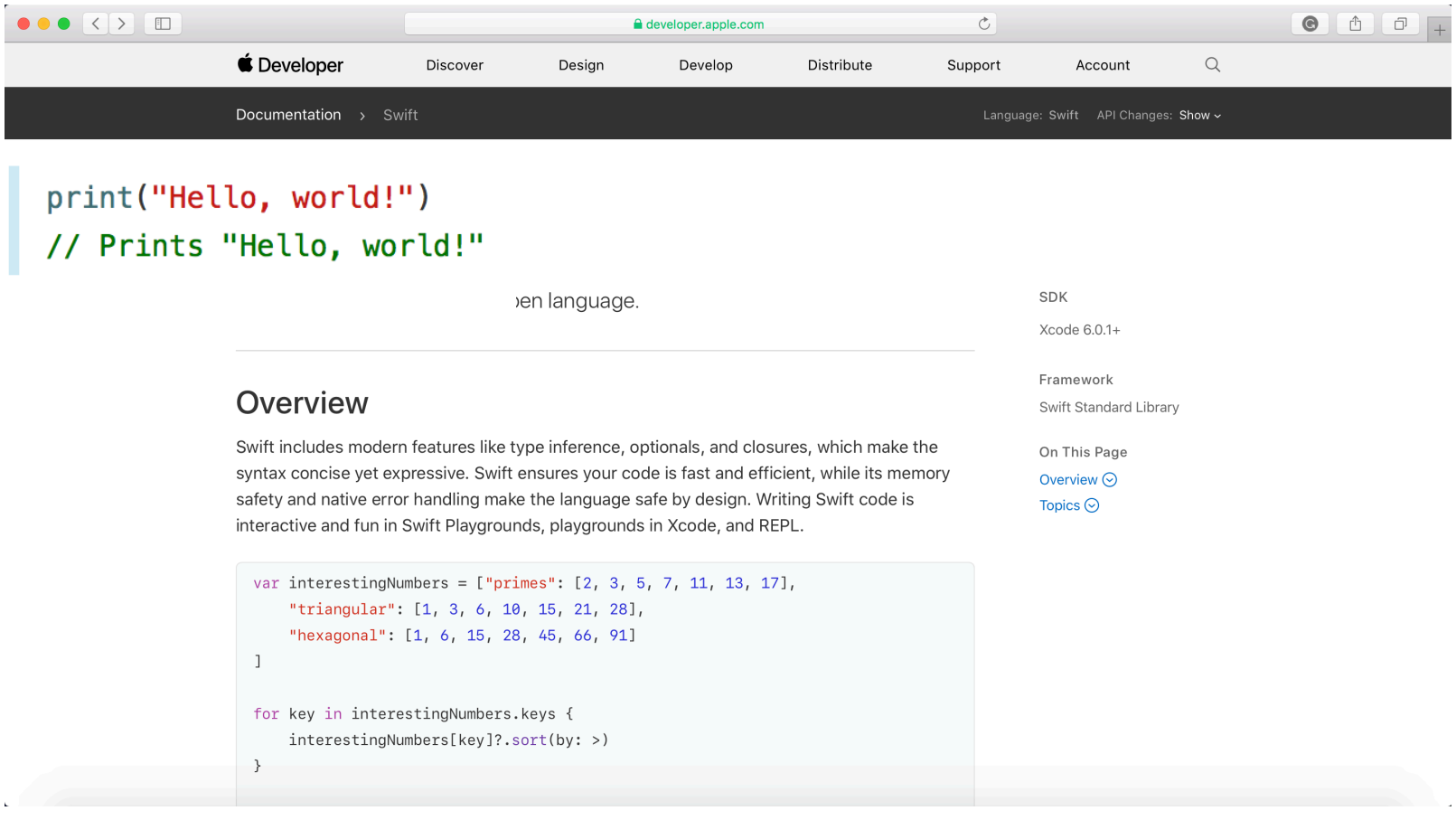


This feature of Xcode allows you to gives you a well-written overview as well as ALL the information you need. This UIKit is used in most of the ViewControllers and it is normally automatically imported when you choose to create a .swift file.

You can also check the official documentation in the website of Apple. Go to

https://developer.apple.com/documentation/swift

and you will see the following page. It shows you more detailed information and allows you to navigate more easily between different sections.



Exploit by yourself in Apple Developer website. Go to other sections like App Services, Media, and Developer Tools. These are the built-in features that you need when you are building a complex app. Take some time and have a look at the features that interest you.

II. A SWIFT TOUR AND PLAYGROUND

You can download **The Swift Programming Language** in iTunes Store. Click the following link to download it to your iBooks library.

https://itunes.apple.com/us/book/the-swift-programming-language-swift-4-2/id881256329?mt=11

I highly recommend you to read the section "**A Swift Tour**" section. It gives you a general idea of Swift syntax. If you don't have iBooks app, use this online version in docs.swift.org:

https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html

You can use Xcode playground to practice the syntax. Download the playground file and open it in your Xcode.

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

In the playground file, you will see these following lines of code:

```
print("Hello, world!")
// Prints "Hello, world!"
```

This allows you to display a message in your console. However, the user will not see this message in the UI. If you don't like using debugger, this is really helpful during the debugging. You can see the state and the value of a certain object in a certain line.

If you have written code in C or Objective-C, this syntax looks familiar to you—in Swift, this line of code is a complete program. You don't need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don't need a main() function. You also don't need to write semicolons at the end of every statement.

This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don't worry if you don't understand something—everything introduced in this tour is explained in detail in the rest of this book.

III. SIMPLE VALUES

Use let to make a constant and var to make a variable. The value of a constant doesn't need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

A constant or variable must have the same type as the value you want to assign to it. However, you don't always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that myVariable is an integer because its initial value is an integer.

If the initial value doesn't provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

> EXPERIMENT
>
> Create a constant with an explicit type of `Float` and a value of `4`.

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

> EXPERIMENT
>
> Try removing the conversion to `String` from the last line. What error do you get?

There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (\) before the parentheses. For example:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

> EXPERIMENT
>
> Use `\()` to include a floating-point calculation in a string and to include someone's name in a greeting.

Use three double quotation marks (""") for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quotation marks. For example:

```
let quotation = """
Even though there's whitespace to the left,
the actual lines aren't indented.
Except for this line.
Double quotes (") can appear without being escaped.

I still have \(apples + oranges) pieces of fruit.
    """
```

```
var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

Create arrays and dictionaries using brackets ([]), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

```
var shoppingList = ["catfish", "water", "tulips"]
shoppingList[1] = "bottle of water"

```

Create arrays and dictionaries using brackets ([]), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

```
var shoppingList = ["catfish", "water", "tulips"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

Arrays automatically grow as you add elements.

```
shoppingList.append("blue paint")
print(shoppingList)
```

To create an empty array or dictionary, use the initializer syntax.

```
let emptyArray = [String]()
let emptyDictionary = [String: Float]()
```

If type information can be inferred, you can write an empty array as [] and an empty dictionary as [:]—for example, when you set a new value for a variable or pass an argument to a function.

```
shoppingList = []
occupations = [:]
```

IV. CONTROL FLOW

Use **if** and **switch** to make conditionals, and use **for-in, while,** and **repeat-while** to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
// Prints "11"
```

In an **if** statement, the conditional must be a Boolean expression—this means that code such as **if score { ... }** is an error, not an implicit comparison to zero.

You can use **if** and **let** together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains **nil** to indicate that a value is missing. Write a question mark (**?**) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
print(optionalString == nil)
// Prints "false"

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

> EXPERIMENT
>
> Change `optionalName` to `nil`. What greeting do you get? Add an `else` clause that sets a different greeting if `optionalName` is `nil`.

If the optional value is **nil**, the conditional is **false** and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after **let**, which makes the unwrapped value available inside the block of code.

Another way to handle optional values is to provide a default value using the **??** operator. If the optional value is missing, the default value is used instead.

```
let nickName: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickName ?? fullName)"
```

Switches support any kind of data and a wide variety of comparison operations—they aren't limited to integers and tests for equality.

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    print("Add some raisins and make ants on a log.")
case "cucumber", "watercress":
    print("That would make a good tea sandwich.")
case let x where x.hasSuffix("pepper"):
    print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}
// Prints "Is it a spicy red pepper?"
```

> EXPERIMENT
>
> Try removing the default case. What error do you get?

Notice how **let** can be used in a pattern to assign the value that matched the pattern to a constant.

After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn't continue to the next case, so there is no need to explicitly break out of the switch at the end of each case's code.

You use **for-in** to iterate over items in a dictionary by providing a pair of names to use for each key-value pair. Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
```

```
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
// Prints "25"
```

> EXPERIMENT
>
> Add another variable to keep track of which kind of number was the largest, as well as what that largest number was.

Use **while** to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
var n = 2
while n < 100 {
    n *= 2
}
print(n)
// Prints "128"

var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
// Prints "128"
```

You can keep an index in a loop by using **..<** to make a range of indexes.

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
// Prints "6"
```

Use **..<** to make a range that omits its upper value, and use **...** to make a range that includes both values.

V. FUNCTIONS AND CLOSURES

Use **func** to declare a function. Call a function by following its name with a list of arguments in parentheses. Use **->** to separate the parameter names and types from the function's return type.

```
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

> EXPERIMENT
>
> Remove the day parameter. Add a parameter to include today's lunch special in the greeting.

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write _ to use no argument label.

```
func greet(_ person: String, on day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value—for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
// Prints "120"
print(statistics.2)
// Prints "120"
```

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

A function can take another function as one of its arguments.

```
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is

executed—you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces ({}). Use in to separate the arguments and return type from the body.

```
numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

> EXPERIMENT
>
> Rewrite the closure to return zero for all odd numbers.

You have several options for writing closures more concisely. When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```
let mappedNumbers = numbers.map({ number in 3 * number })
print(mappedNumbers)
// Prints "[60, 57, 21, 36]"
```

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

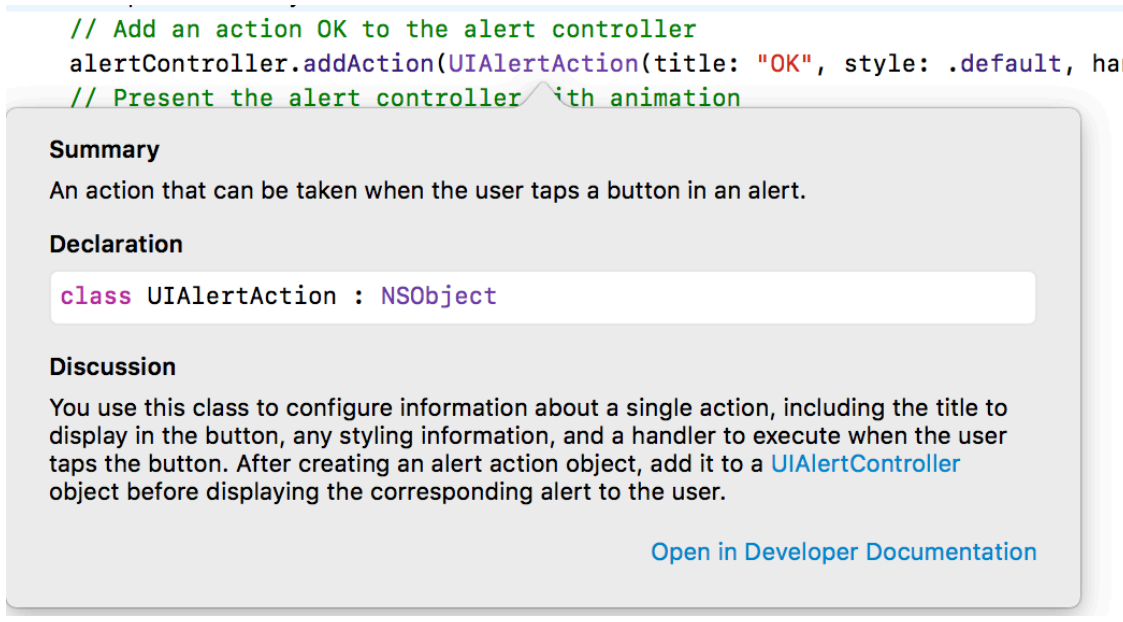
```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```

EXERCISE #1

Remember these lines of code in the ViewController?

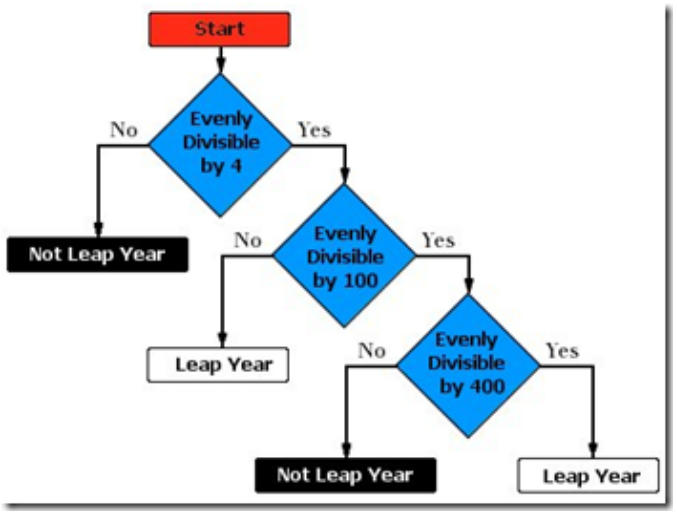
```
// Add an action OK to the alert controller
alertController.addAction(UIAlertAction(title: "OK", style: .default, handler: nil))
```

Use option click to check what a UIAlertAction is in the documentation. Read the overview of this class and list some of its methods.



EXERCISE #2

In a playground file, write a function `isLeapYear(for year: Int)` to calculate if a year is a leap year. This following flowchart will help you with the testing conditions. If you have trouble writing `if-else` statements, please go back to the section IV to go through if-else again.



Hint: You can write nested `if-else` statements, or you can use several `else if`s to make your code cleaner.

SUMMARY

In this chapter, we have worked on these topics:

- checked documentation in Xcode and online
- used Xcode playgrounds
- learnt variables and constants
- learnt simple data structures like arrays and dictionaries
- learnt control flow: loops and conditionals
- learnt functions and closures

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