Swift

Swift is a new programming language in development by Apple. It is framed upon the good parts of C and Objective-C, and maintains the readability of Objective-C, with access to existing Cocoa frameworks.

Before jumping into Swift basics, we need the tools and should get familiar with using these for coding in Swift. Read through this for learning about Xcode.

Introduction to Swift

Assuming that you have learnt the usage of the tools needed for coding, we can jump to basics of Swift and language conventions.

As said earlier, Swift is a new and in development language by Apple. The language is designed such that it is easy to read as code, and easy to learn from scratch.

First of all, you'll need to download the Swift book and read through it for detailed understanding of each components of the language. You can get the book(The Swift Programming Language 3) from iBooks store here and also you can read it online on Apple developer website having web version of the book.

Let's get into basics of Swift. It's a fun programming language to learn, so it's mandatory that you to read through code snippets and try them on playground simultaneously.

As in any other language in Swift too, we can start by displaying "Hello world!", and it's as simple as that, just `print("Hello World!")` and you get the output. This line of code is a full program in Swift.

Variables and constants

To declare a variable use the keyword `var` before the identifier for the variable, such as `var a = 10`. This gives an identifier "a", an integer value of 10 initially, which can be changed further down the code.

To have a constant, which can be initialized just once and which will hold the value which will be immutable, we can declare it with a keyword `let`, such as `let b = 10`. This will give the identifier "b" with an integer value of 10, which cannot be changed further down the code.

If you have already noticed, we don't need to always specify the datatype explicitly. Swift compiler has the capacity to infer the type of constant or variable when you assign a value to it, this is called *type inference*. If you want the identifier to be tied to a specific type, then you can explicitly specify it to be of that type by `var a: Int`, so now "a" can hold only Int values, where the compiler would show an error if given any other type of value.

Note that, by now you would have noticed, there is a space in between each keyword and identifier. This is the Swift way of code, where you are expected to give a blank space between each identifier, keyword, and symbols. If not the compiler will show an error.

Coding style in Swift

- When coding in Swift, you should follow the Swift way of naming, usage of grammar and case conventions.
- These are design guidelines that Apple has created, just so that when followed properly while coding, will result in very clear and readable code. For detailed explanation on these guidelines click here.

Comments

- To have a comment in the code block, write a comment by prefixing the line with `//` (two forward slashes). As `// This is a comment.` This is a one line comment.
- We can have multiple line comments which starts `/*` and ends with `*/`, as

```
/* comment starts here
this is the second line
this is the ending line*/
```

NOTE: You can access this playground file, which consist of every example given in this document, so that you can experiment with the code yourself.

- The comments are useful for giving information about a method, a property in the code.
- The comments can be used for grouping methods in a file, for this we use the `// MARK:` comments followed by the name for that section. Such as `// MARK: UITableView Methods`. The methods below a `// MARK:` will be grouped under the section name given for that comment, until the next `// MARK:` comment.

- When coding, if you have a planned design and should make a note that you should be doing this further, you can create a `// TODO:` comment and continue to work on current code.
  Such as `// TODO: Create a function to get the values`.

String

- `String` is a datatype in Swift that are Unicode compliant, so that any Unicode value can be used in Swift without error, as each character in Swift String is encoded as a unicode scalar which is a unique 21 bit number.

- To represent a String, you should put it between `""` (two double quotes). Such as `"This is a String"`, and you can assign an identifier with the value, `let string = "This is a String"`.

- If a String value is assigned to be `""` then it is an empty String.

- You can concatenate two strings by using the concatenating operator, which is `+`, such as `var newString = string + " and this is second string"`, this will give `newString` a value of `This is a String and this is second string`.

- To have a value in the `print` statement, and to interpolate values into String we use the values inside the string literals by prefixing the literals with `\`. Which is

```
  let numOfStates = 50
  let country = "USA"
  let interpolatedString = "The number of states in \(country) is \(numOfStates)."
  print(interpolatedString)
  // prints The number of states in USA is 50.
  ```

Optionals

Swift introduces another property, which is you can specify a variable to be an opt

An optional value is one which may or may not have a value, as it may have a `nil`

We need optional values since, we can convert from one datatype to another, and thi

Value type and reference type

 * A value type is any type data that creates a copy of itself when called or assig
```
```

```swift
// Value type example
struct Example {
var data: Int = -1
}
var a = Example()
var b = a // a is copied to b.
a.data = 22 // Changes a, not b.
print("(a.data), (b.data)") // prints "22, -1".
```

![ValuetypeExample](ScreenShots/Valuetype Example.png "Value type")

```
// Reference type example
class Example {
  var data: Int = -1
}
var x = Example()
var y = x                         // x is copied to y.
x.data = 22                     // changes the instance referred to by x and y.
print("\(x.data), \(y.data)")    // prints "22, 22".
```

![ReferencetypeExample](ScreenShots/Referencetype Example.png "Reference type")

- Well, having gone through the above definition of value types and reference type, you now know it's meaning on the surface, but in Swift this mechanism works very intelligently. In Swift, when we assign a value type to another identifier, the identifier will have reference to same address space as the initial source, until it is mutated. This is called as ***copy on write*** feature.

- The value type and reference type are further more linked to the `let` and `var`. As in for reference types, `let` means the reference must remain constant. In other words, you can’t change the instance which the constant references, but you can mutate the instance itself. See this:

  ```swift
  class Car {
  var wasWashed = false
  }
  let mercedes = Car()
  let s350 = mercedes
  s350.wasWashed = true
  if mercedes.wasWashed {
  ```

```
print("Mercedes cars were washed")
} else {
print("Just the S350 was washed")
}
// prints Mercedes cars were washed
```

~~~
In the above reference example, though mercedes and s350 are declared as `let`, but

 * For value types, `let` means the instance must remain constant. No properties of

 ```swift
 struct Dog {
     var wasFed = false
 }
 let labrador = Dog()
 let puppy = labrador
 labrador.wasFed = true // Cannot do this!
 puppy.wasFed = true // Cannot do this!
 ```

 * Having gone through the details about value types and reference types, we now ha

 * For detailed view on value and reference type in Swift, make sure you read throu

 * You can read the [Raywenderlich](https://www.raywenderlich.com)'s article on sam

## Switch statement

 * In Swift switch cases have to be exhaustive. Which is to say that, the switch sh

 * If you cannot explicitly handle every case for some reason, only then should the

 * More so, in Swift if the switch case is handling a value, having 'n' cases, then

## SwiftLint

 * SwiftLint is a package that let's you see every mistake in your code regarding t

## Pattern Matching

Swift provides extensive pattern matching for comparing, type casting, and many suc
More so, patterns are rules, on which values are matched against.

For example, for checking if an Int value is an even number, we can give it in `if`
~~~

```
swift
func isEven(number: Int) -> Bool {
return number % 2 == 0
}
```

if isEven(2) {
print("The number is an even number")
// Do something
} else {
print("The number is an odd number")
// Do something
}

```
In the `isEven()` above, the rule which is the pattern is that `number % 2` and val

Similarly, we have pattern to match any value in the range of values in `for` loop.
```

swift
for _ in 1…3 {
// Do something
}
```

Which will do something in the loop for three times, without regard to the range value of the iteration. This pattern `_` (underscore) is called a "Wildcard Pattern", and it matches any value.

Note that in any `for` loop, we can specify `prefixValue..<postfixValue` (two `.` 's with `<` ) for matching the range to include prefixed value as starting of range and loop till one value less than the postfixed value. To include both the prefix and postfix value, specify `prefixValue...postfixValue` (three `.` 's).

Similarly, for tuple values, you can have a pattern matching it for same number of values in that tuple. Such as `(1,2)` can be matched by tuple of `(x,y)`, where x represents 1 and y represents 2. This can be implemented in `for` and `Array` as

```
let tuples = [(1,2), (3,4), (5,6)]
for (x, y) in tuples {
// Do something with x
// Do something with y
}
```

Swift provides inbuilt regular expression matching option, which is we can give a String value and match it with a regular expression for our functionality. Such as if we could extend the above tuple example to get tuples and print the value of each tuple with a proper statement specifying each iteration count value also, then it would be as seen here:
```

```
class TupleExample {
    var count = 1
    enum iterationNumber: Character {
        case first = "1"  // for extension st
        case second = "2" // for extension nd
        case third = "3"  // for extension rd
    }

    func getTuples(howManyTuples howManyTuples: Int) -> [(Int, Int)] {
        var i = 1, j = 2, tuples: [(Int,Int)] = []
        for _ in 1...howManyTuples {
            tuples.append((i,j))
            i += 2
            j += 2
        }
        return tuples
    }

    func doThisFirst() {
        let tuples = getTuples(howManyTuples: 10)
        for (x, y) in tuples {
            let countLastCharacter = String(count).characters.last!
            let characterCount = String(count).characters.count

            //The pattern .1. matches xx..x1xx..x such as 514,1213,2123,4533144 lik
            if doesMatchPattern(pattern: ".1.", count: count) {

                // For count between 100 and 1000.
                if characterCount == 3 {
                    printValue(x: x, y: y, withExtension: "th")
                } // For count above 1000, and matching only which has the pattern 
                else if characterCount > 3 && doesMatchPattern(pattern: "..1.", cou
                    printValue(x: x, y: y, withExtension: "th")
                } // For every other count value in the range above 1000 and not in
                else {
                    evaluateAndPrint(x: x, y: y, countLastCharacter: countLastChara
                }
            } else {
                if characterCount < 3 && doesMatchPattern(pattern: "1.", count: cou
                    printValue(x: x, y: y, withExtension: "th")
                } else {
                    evaluateAndPrint(x: x, y: y, countLastCharacter: countLastChara
                }
            }
            count += 1
        }
    }
    func doesMatchPattern(pattern pattern: String, count: Int) -> Bool {
        //Evaluates if given count matches with the pattern.
        return String(count).rangeOfString(pattern, options: .RegularExpressionSear
    }
    func evaluateAndPrint(x x: Int, y: Int, countLastCharacter: Character) {
        switch countLastCharacter {
        case iterationNumber.first.rawValue:
                printValue(x: x, y: y, withExtension: "st")
            case iterationNumber.second.rawValue:
                printValue(x: x, y: y, withExtension: "nd")
            case iterationNumber.third.rawValue:
                printValue(x: x, y: y, withExtension: "rd")
            default:  //Any number ending with number other than 1,2,3 and not between
                printValue(x: x, y: y, withExtension: "th")
            }

        }
        func printValue(x x: Int, y: Int, withExtension: String ) {
                print("This is the value of (x,y):(\(x),\(y)) for \(count)\(withExtensi
        }
    }
    let callTuple = TupleExample()
    callTuple.doThisFirst()
```

The regular expression can be given in String's rangeOfString method, in which the String parameter will be the regular expression and the options will have .RegularExpressionSearch. This method will return the range for the given String if it has any substring in it that matches the given regular expression. Experiment with the code for the above example in the provided playground file. Change the regular expression and find the variations in the results.

For the Apple's documentation on regular expression click here
For details about regular expressions click here
Read through Raywenderlich's tutorial on regular expression for details on how to use them in Swift.