Innotech
MohammadHosein
March 7, 2018

Swift Document

Lynda.Swift.3.Essential.Training.The.Basics

In this document first we've got introduction to Swift with playground, definition of comments, available attribute and repl then it'll be indicated syntaxes and in each part after the syntax there gonna be some small examples which have some extra point on it

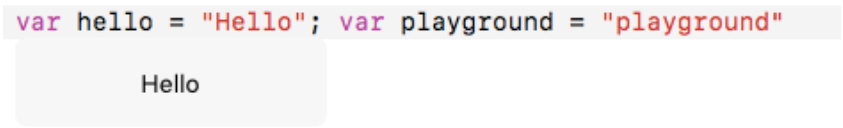
Playground

Playground helps you learning coding in Swift and have awesome features for that. in this mode each line run right after you right your line and if you wanna run it again theres a icon at the bottom to run it again.

in Xcode if you want line numbers there's the solution:

Xcode -> Preference -> text editing

there is a slide bar at the right which you have your line run if you click on the gray square there your result will shown at the bottom of your line of code like the picture bellow:



Comments

// is to comment a line and for multiple lines /* */

Land marks are useful and can be accessed from jump bar at the top :

TODO - FIXME - MARK

In this subject we've got an option called render documents that make your comments a document but i won't describe it. Here's just an example :

```
//: Single-line delimiter

/*: Text on this line not displayed in rendered markup
  ## Header 2
 ### Header 3

 > Block note

 * Milk
 * Bread
 * Bananas

 1. **Learn Swift**
 2. Develop an _awesome_ app
 3. Retire
 ----
```

Single-line delimiter

Header 2

Header 3

> **Note**
> Block note

- Milk
- Bread
- Bananas

1. **Learn Swift**
2. Develop an *awesome* app
3. Retire

Quick help comment : at the bottom of function press cmd + opt + / to define your function , hold option and click on your function enjoy the result :D

```
5  /// Squares an integer.
6  ///
7  /// - warning: Integer must be less 3037000500
8  ///
9  /// - parameter integer: Integer to square
10 ///
11 /// - returns: `integer` squared
12 func square(integer: Int) -> Int {
```

Declaration `func square(integer: Int) -> Int`

Description Squares an integer.

Warning
Integer must be less 3037000500

Parameters `integer` Integer to square

Returns `integer` squared

Declared In 01_05_finished.playground

Available: the available attribute indicates the lifecycle using platform or version of Swift language.

@available(*, introduced: 1.2, deprecated: 3.0, message: "Use `perform(action: () -> Void)` instead")

REPL

REPL is an interactive sand box environment that can be used by typing Swift in terminal; also REPL is being used in Xcode for debugging , at your break point by typing repl you can print your variables and etc.

Data Types

in Swift its not necessary to show the type, the data in front of it will assign the type automatically here are examples of syntaxes:

```
var greetingString: String = "Hello, playground"

let `class` = "Computer Science"

let rocket: Character = "🚀"

let integer: Int = -1

let unsignedInt: UInt = 1_234_567_890
```

* emojis can accessed via ctrl + cmd + space bar
* let types are immutable(instead of immutable variables in objective-c)
* in the last example "_ " is being dismissed and it's just for partitioning the number

Unicode : to define unicodes type \u followed by value in curly braces
(e.g. let myUnicode : Character = "\u{61}")

Strings

String structure have lots of useful functions indicated in example bellow that i'll guarantee there's no need of description for them.

.characters.count / .isEmpty / .uppercased / .hasPrefix / .append / .removeSubrange / .replace Subrange

String(repeating: "🙂", count: 5)

by importing foundation : .replacingOccurrence (of: , with:)

notice that index method uses startIndex or endIndex with offsetBy

Arrays

For initializing arrays there are lots of ways:D

- var animals = ["cat" , "dog" , "rabbit"]
- var scores : [Int]
- var quizScores : Array<Double>()
- var quizScores : Array<Double> = []
- var averageScores = [Float](repeating: 0.0, count: 5)

And lots of functions:

.count / .contains /.sort / .sorted / .append / .first / .remove / .insert

Dictionaries

Initialization

- var stockPrices = ["GOOG" : 782.3 , "AAPL" : 11.92]
- Var birthYears : [String : Int] = [:]
- var raceResults = Dictionary<Int , String>()

Functions

.count / .keys / .isEmpty / .updateValues / .removeValue

e.g

```
raceResults[1] = "Lewis Hamilton"

raceResults = [
    2: "Daniel Ricciardo",
    3: "Sergio Perez"
]

raceResults[3] = nil

let oldValue = raceResults.updateValue("Lewis Hamilton", forKey: 1)

let removedValue = raceResults.removeValue(forKey: 4)
```

Sets

Sets are awesome unordered containers use for store unique variables.

Initialization

var teachers = Set<String>()

var teachers : Set<String> = []

var teachers : Set = [“Charlotte” , : “Cox”]

Function

.insert / .remove / .index / .index / .isSubset / .isStrictSubset / .isSuperSet / .symmetricDifference / .intersection / .formUnion / .subtract

Tuples

Tuples in Swift can be used to store multiple values into a single compound value

let httpStatus200 = (200,”OK”)

var numbers: (Int, Float) to access: numbers.2 (also can be accessed by names)

```
playerScores = ([134_000, 128_500, 156_250], "Scott", "Gardner")

var (scores, firstName, lastName) = playerScores

firstName = "Eric"

let (scottsScores, _, _) = playerScores

var scottsScore = (100, name: (first: "Scott", last: "Gardner"))

scottsScore.name.first = "Eric"
```

* notice that lastName , firstName and scores are optionals (optionals will be introduced later)

Comparison

x == y and x===y : first one checks values and the second one checks their references

Ranges

1..<4 and 1…4

Ternary conditional operators

We've got ternary conditional operator in Swift like other languages.

```
let birthYear = 2005
let generation = birthYear < 1945 ? "Greatest Generation" : "Generation Z"

var selectedSize: String?
let orderSize = selectedSize ?? "M"
```

* second example meant that if selectedSize was empty, make orderSize M

Loops

for loops

if there is an array to search in it:

```
let numbers = [1, 2, 3]
for i in numbers {
   print(i)
}
```

if not:

```
for i in 1...3 {
  print(i)
}
```

Other examples:

```
for i in 2...10 where i % 2 == 0 {
    print(i)
}

for i in stride(from: 2, through: 10, by: 2) {
    print(i)
}
```

while loops

```
while condition {
        statement
}

repeat {
        statement
}while condition
```

in repeat-while loops first compiler execute the statement and then condition will checked.

If Conditionals

if condition {

statement

}else {

statement

}

```
if #available(iOS 10, macOS 10.12, watchOS 3.0, *) {
    // Use applicable API from iOS 10, macOS 10.12, and watchOS 3.0
} else {
    // Use earlier API
}
```

Optionals

optional binding

Optional binding allows an optional value to check if it has a value.

var optionalValue : Type?

and also to handle these values if-let is handy:

if let optional = optionalValue {

statement

}else{

statement

}

and if you want to be able to modify the optional value use var instead of let

* notice that the optional in the above phrase is a local value.

```swift
var firstName: String? = "Betty"

var lastName: String? = "Gardner"

if let firstName = firstName, var lastName = lastName {
    lastName = lastName.uppercased()
    print("Hello, my name is \(firstName) \(lastName)")
}
```

chain optionals

In optionals instead of using nested if-let phrases that causes the pyramid of doom optional chains are being used. Look after the example below:

```swift
if let manager = betty.manager {
    if let manager = manager.manager {
        manager.printName()
    }
}

betty.manager?.manager?.printName()
```

* notice that optional chain will always return an optional

Guard Statement

Guard literally is guarding your variable to not to be sth and actually works as an opposite of if.

guard condition else {

statement

}

Difer Statement

Defer statement delays the execution of it's body of code until right before the control flow exits the scope which defer is in it

defer{ statement }

Functions

func funcName(internalName externalName : Type) -> returnType

func funcName(_ internalName : Type)

```
func add(_ ints: Int...) -> Int {
    return ints.reduce(0, +)
}

add(2, 4, 6, 8, 10)
```

```
var studentsScore = 76.0

func apply(extraCredit: Double, toScore score: inout Double) {
    score += extraCredit
}

apply(extraCredit: 10.0, toScore: &studentsScore)
```

Functions also can be overloaded by different inputs or return types like examples below:

- func processInput(input: String) { }
- func processInput(input: Int) { }
- func processInput(int: Int) { }
- func processInput(input: Int) -> String {
  return "\(input)."
}
- func processInput(input: Int) -> Int {
  return input * input
}

Making An Operator

Swift allows you to create your own operators also like functions. They must first be declared globally

Types of operators: infix - prefix - postfix

Also precedence group must be defined; default precedence like MultiplicationPrecedence con be used or you can define your own like the example bellow:

```
precedencegroup ExponentPrecedence {
    higherThan: MultiplicationPrecedence
    associativity: left // 2 * 3 / 4 == (2 * 3) / 4
}


infix operator ** : ExponentPrecedence

func **(base: Int, exponent: Int) -> Int {
    var exp = exponent
    var result = 1

    while (exp != 0) {
        result *= base
        exp -= 1
    }

    return result
}
```

Handling Errors

Sometimes a function can encounter predictable errors; in this case “throws” keyword is being used and with try and catch, we handle errors.

```
func functionName() throws {
}


do{
        try functionName()
} catch {
print(“error occurred “)
}
```

“try!” means that if the function in front of it throws the error, program will be interrupted.

let maybeSuccess = try? performActionThatMightFail()

And here if the function encounters an error value of maybeSuccess will be nil.

Closures

Closures are block of executable code that can be called in line or stored as a value and passed to be executed at a later time.

For example in the filter function there is a closure which returns a String type. Where the function is going to call, closure can be defined within curly braces and also after the “in” keyword the closure is defined.

```
let namesBeginningWithS = names.filter({ (name: String) -> Bool in
    return name.lowercased().characters.first! == "s"
})

let namesIncludingAnE = names.filter {
    $0.lowercased().characters.contains("e")
}
```

* if function does not get any other requirements the parenthesis can be deleted.
* if an inline closure is the last parameter also called trailing closure it can be out of the parenthesis:

```
let namesBeginningWithS = names.filter(){ (name: String) -> Bool in
    return name.lowercased().characters.first! == "s"
}
```

* when you use closure to assign value, type of the value must be defined.(like String type here)

```
let randomNameGetter: String = {
    let randomIndex = Int(arc4random_uniform(UInt32(names.count)))
    return names[randomIndex]
}()
```

But if the closure only have one expression there is no need of defining type.:D

```
let helloSayer = {print(“Hellooo!!”)}            to call: helloSayer()
```

* if you want to assign the closure, not the return value in the last example:

```
let randomNameGetter: () -> String = {
    let randomIndex = Int(arc4random_uniform(UInt32(names.count)))
    return names[randomIndex]
}
```

* there is a function that we define with closure:

```
func execute(_ closure: @autoclosure () -> Void) {
    closure()
}

execute(print("Hello, again!"))
```

@autoclosure means that instead of using execute(helloSayer) we can define the closure in line.

* there is another important thing in closures (that I wont describe here) as @scaping I do recommend that to check it in other references.

Classes

Swift like other languages also have classes which are reference types. Classes have initializers that are necessary for non-optional values and while the object is created the initializer will called.

```
class ElectricVehicle {

    // Instance properties
    var passengerCapacity = 4
    let zeroTo60: Float
    var color: UIColor

    // Initializers
    init(passengers: Int, zeroTo60: Float, color: UIColor = ■) {
        passengerCapacity = passengers
        self.zeroTo60 = zeroTo60
        self.color = color
    }

    convenience init(zeroTo60: Float) {
        self.init(passengers: 4, zeroTo60: zeroTo60)
    }
}
```

* the second initializer in this example which called convenience initializer is optionally and can call the designated or other convenience initializers. Classes can have multiple convenience initializers but only one designated.

Inheritance

In Swift inheritance can be just from one super class not more but you can use more than on protocols which it'll be explained later.

```
class subClassName : superClassName {
        override func superClassFunction(){
        }
}
```

* in subclasses if no initializers defined it'll inherit all of them from the superclass; if all of the designated initializers defined, then conveniences will inherit from the super class but if you implement some of the each type of initializers your on your own and non of the initializers can be inherited.
* notice that in this document it's been presumed you've been completely familiar with classes and inheritance.

Structure

Structure are value types which can be implemented by "struct" keyword. They don't have deinitializer and no inheritance but moreover they have default memberwise initializer which means there is no necessary initializer to worry about.

* convenience initializers in struct don't need the "convenience" keyword.
* unlike classes that values can be modified simply, in structs functions which change the value of struct needed to have "mutating" keyword.

Enumerations

Enumerations are so important in Swift. An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code. Unlike other languages, enumerations do not have to have a raw datatype they're datatype themselves.

```
enum Direction {

    case up
    case down
    case left
    case right

}

var direction = Direction.up

direction = .down
```

But optionally you can have:

```
enum Heading : Int {

    case north, south, east, west

}

let heading = Heading(rawValue: 0)
```

```
north
```

```
let invalidHeading = Heading(rawValue: 10)
```

```
nil
```

```
enum Proverb : String {

    case fortune = "Fortune favors the bold"
    case late = "Better late than never"
    case practice = "Practice makes permanent"

    // Instance properties
    var uppercased: String {
        return rawValue.uppercased()
    }
}
```

Enumerations above are called case value but there are enumerations with associated values too.This enables you to store additional custom information along with the case value, and permits this information to vary each time you use that case in your code.For example, suppose an inventory tracking system needs to track products by two different types of barcode:

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

Protocols

A protocols defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality that you can be used in classes by implementing their methods.

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```