Contents

About Swift

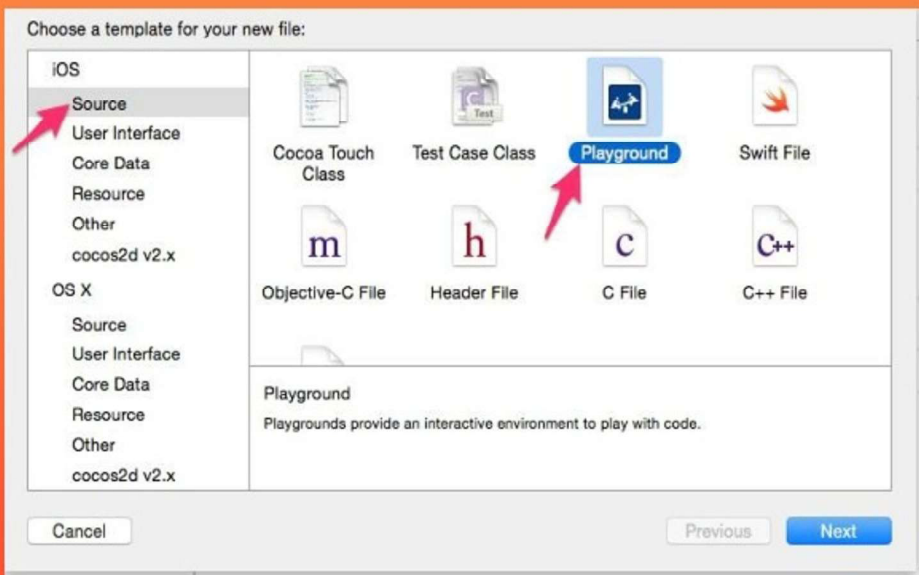
- Swift is a new scripting programming language for iOS and OS X apps.
- Swift is easy, flexible and funny.
- Unlike Objective-C, Swift is not C Compatible. Objective-C is a superset of C but Swift is not.
- Swift is readable like Objective-C and designed to be familiar to Objective-C developers.
- You don't have to write semicolons, but you must write it if you want to write multiple statements in single line.
- You can start writing apps with Swift language starting from Xcode 6.
- Swift doesn't require main function to start with.

Hello World!

As usual we will start with printing "Hello World" message. We will use something called playground to explore Swift language. It's an amazing tool to write and debug code without compile or run.

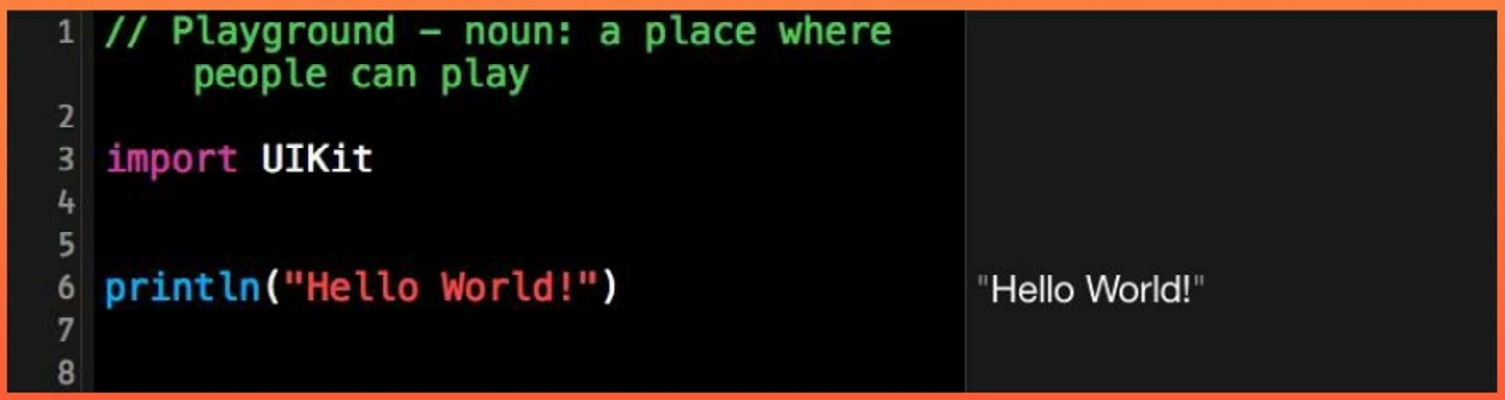
Create or open an Xcode project

and create new playground:



Hello World!

Use "NSLog" or "println" to print message to console. As you see in the right side you can see in real time the values of variables or console message.

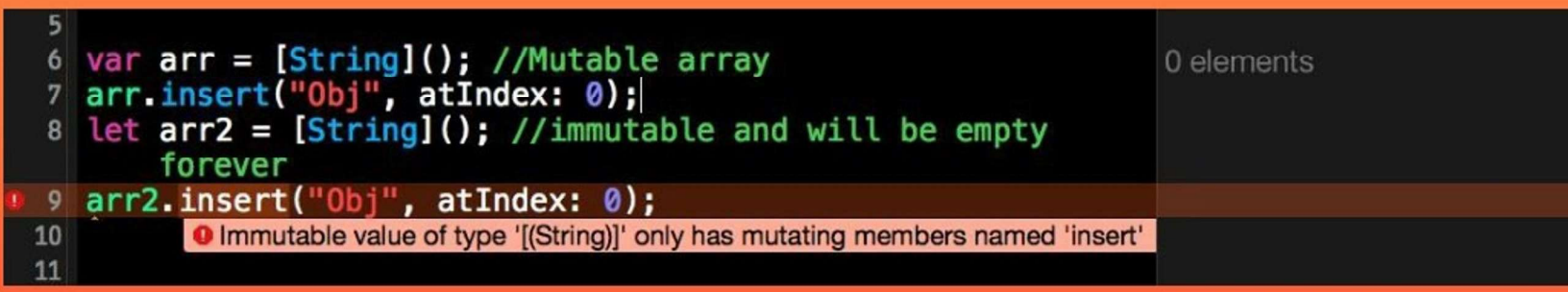


Variables & Constants.

In Objective-C we used to use mutability, for example:

NSArray and NSMutableArray or NSString and NSMutableString

In Swift, when you use var, all objects will be mutable BUT when you use let, all objects will be immutable:

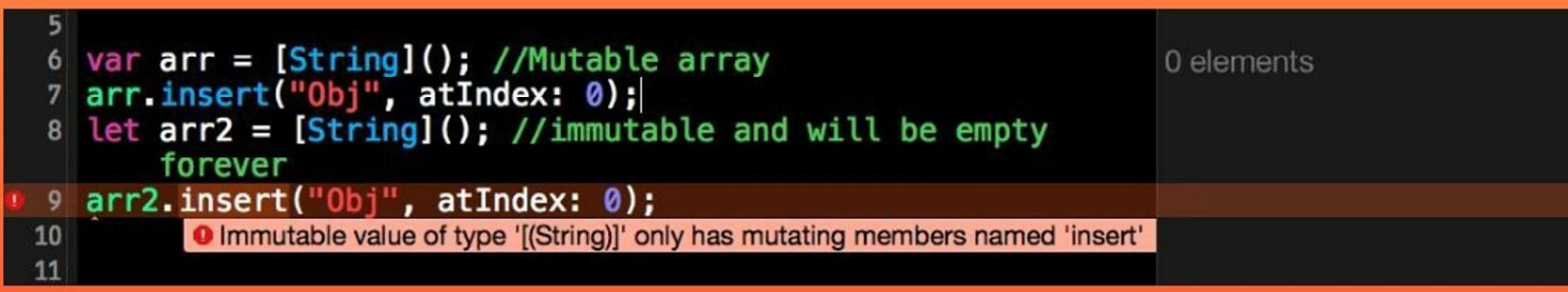


Variables & Constants.

In Objective-C we used to use mutability, for example:

NSArray and NSMutableArray or NSString and NSMutableString

In Swift, when you use var, all objects will be mutable BUT when you use let, all objects will be immutable:

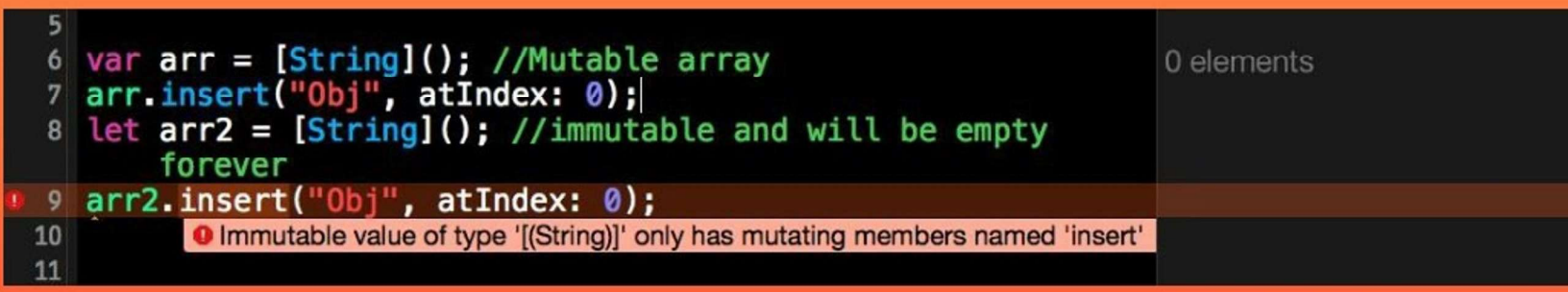


Variables & Constants.

In Objective-C we used to use mutability, for example:

NSArray and NSMutableArray or NSString and NSMutableString

In Swift, when you use var, all objects will be mutable BUT when you use let, all objects will be immutable:



Printing Output

- We introduced the new way to print output using println(). Its very similar to NSLog() but NSLog is slower, adds timestamp to output message and appear in device log. Println() appear in debugger log only.
- In Swift you can insert values of variables inside String using "\()" a backslash with parentheses, check example:

```
6  let team = "Barcelona"                          "Barcelona"
7  let player = "Messi"                            "Messi"
8
9  var goalPerMatch = 3                            3
10 var matches = 100                               100
11
12 println("\(player) scored \(matches *           "Messi scored 300 goals for Barcelona"
       goalPerMatch) goals for \(team)")
```

Type Conversion

```
5
6  var price = 10.5                                    10.5
7  var count = 21                                      21
8
9  //var total = price * count //Compile error
10
11 var total = Double(count) * price //total in        220.5
       double
12 var total2 = Int(total) //total in integer          220
13
```

Here we should convert any one of them so the two variables be in same type.

Swift guarantees safety in your code and makes you decide the type of your result.

Type Conversion

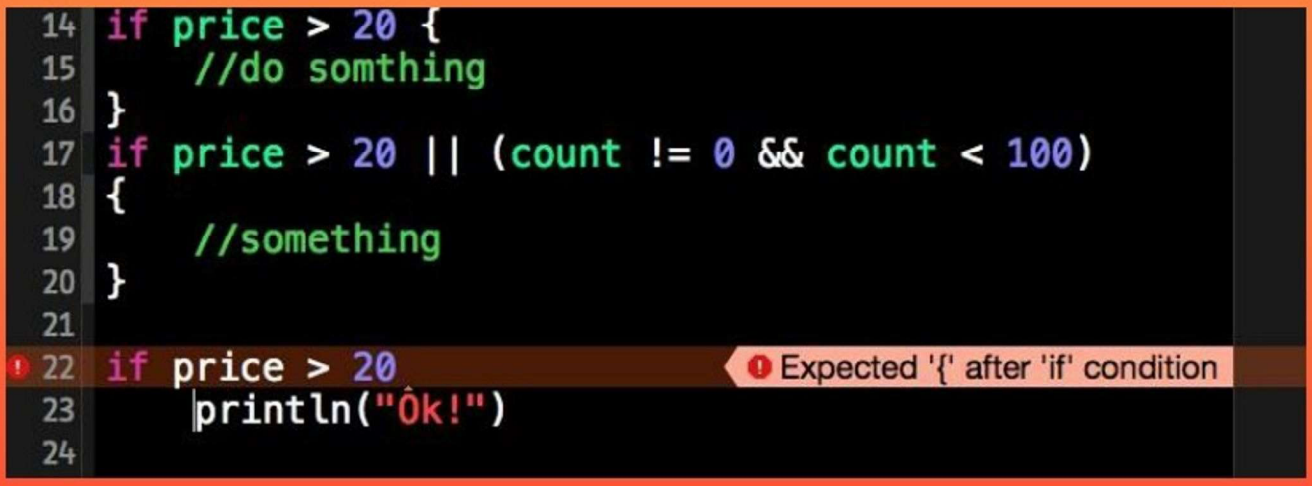
```
5
6  var price = 10.5                                    10.5
7  var count = 21                                      21
8
9  //var total = price * count //Compile error
10
11 var total = Double(count) * price //total in        220.5
       double
12 var total2 = Int(total) //total in integer          220
13
```

Here we should convert any one of them so the two variables be in same type.

Swift guarantees safety in your code and makes you decide the type of your result.

If

- In Swift, you don't have to add parentheses around the condition. But you should use them in complex conditions.
- Curly braces { } are required around block of code after If or else. This also provide safety to your code.



If

- Conditions must be Boolean, true or false. Thus, the next code will not work as it was working in Objective-C :

Objective-C

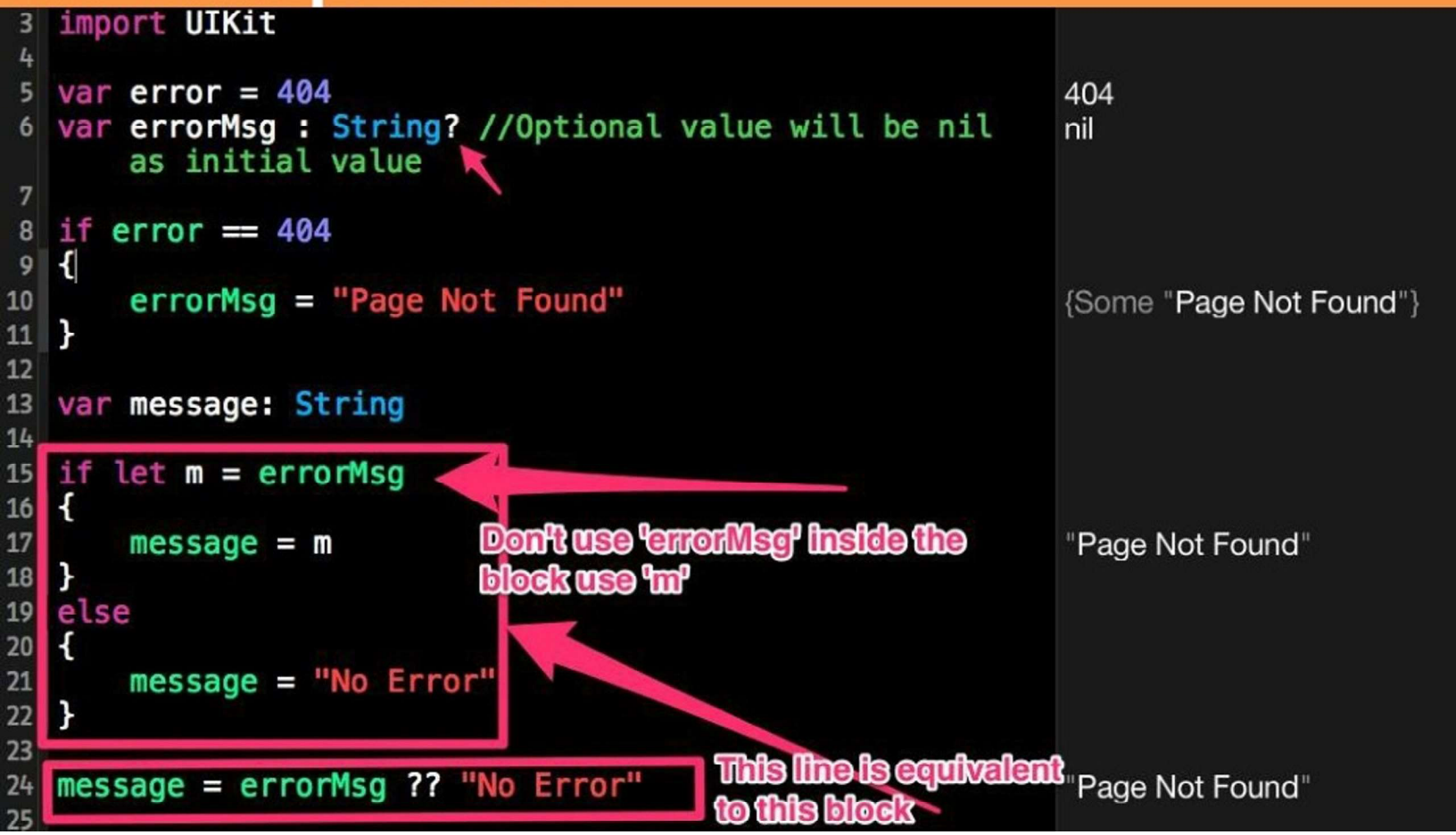
```
UIView *view = self.view;

if (view) { // if view not equal nil
    //Do something
}
```

Swift

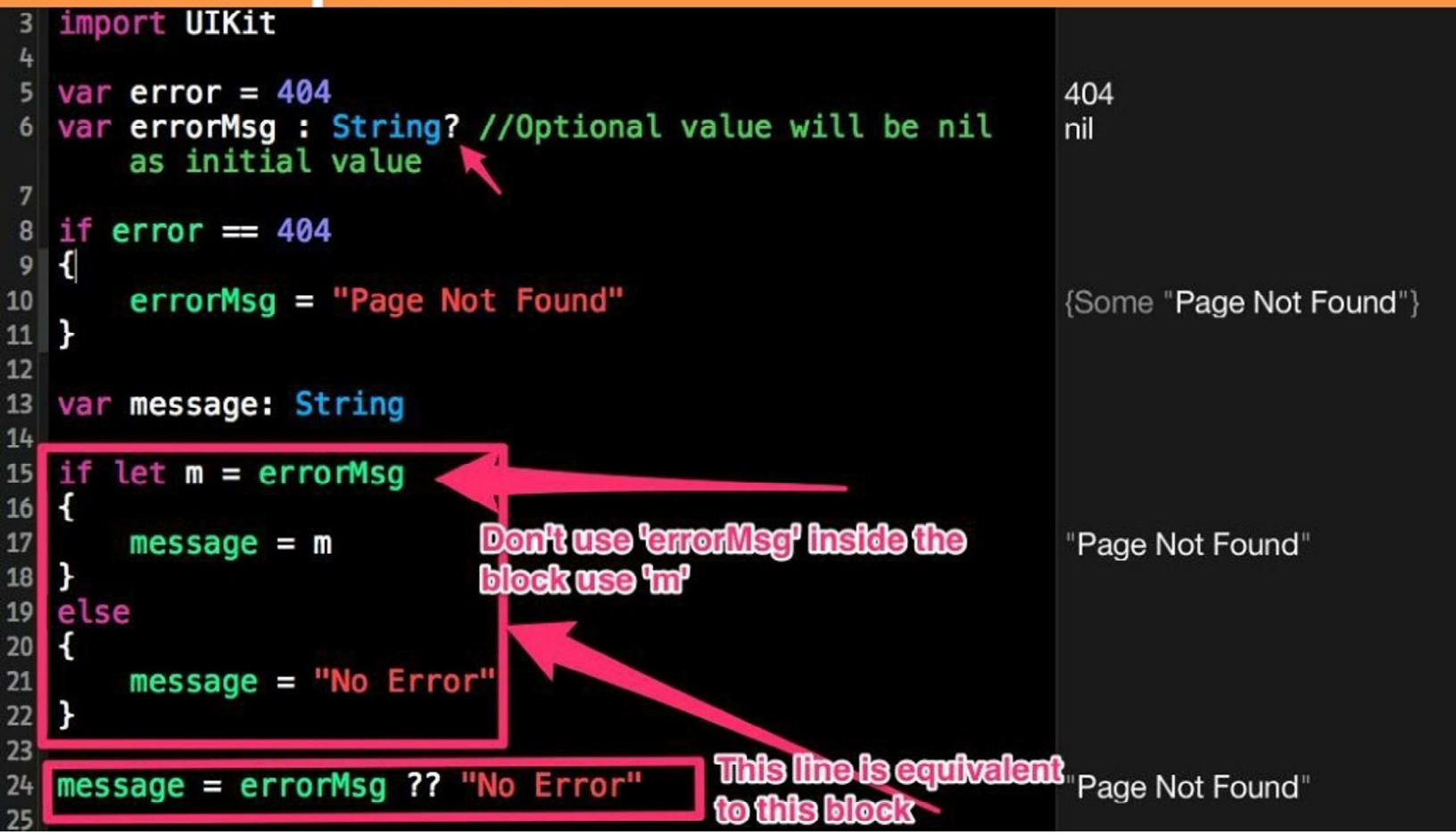
```
23 if score
24 {
25     //Score is greater than 0
26 }
```

As you see in Swift, you cannot check in variable directly like Objective-C.

If With Optionals



If With Optionals



Switch


let country:String = "Egypt"
switch country
{
case "Egypt":
println("Arabic");
case "USA":
println("American English");
case "France":
println("French");
default:
println("Other language")
}
"Egypt"
"Arabic"
break is not required
default is required as you can't cover all possible cases of string

Switch


let country:String = "Egypt"
switch country
{
case "Egypt":
println("Arabic");
case "USA":
println("American English");
case "France":
println("French");
default:
println("Other language")
}
"Egypt"
"Arabic"
break is not required
default is required as you can't cover all possible cases of string

Switch Cont.

```
5
6  let month = "December"
7
8  switch month{
9      case "March", "April", "May":
10         println("We are in Spring")
11
12     case "June", "July", "August":
13         println("We are in Summer")
14
15     case "September", "October", "November":
16         println("We are in Autumn")
17
18     case "December", "January", "February":
19         println("We are in Winter")
20
21     default:
22         println("We can't indentify the season :(")
23
24 }
```

"December"

"We are in Winter"

Switch Cont.

```
5
6  let month = "December"
7
8  switch month{
9      case "March", "April", "May":
10         println("We are in Spring")
11
12     case "June", "July", "August":
13         println("We are in Summer")
14
15     case "September", "October", "November":
16         println("We are in Autumn")
17
18     case "December", "January", "February":
19         println("We are in Winter")
20
21     default:
22         println("We can't indentify the season :(")
23
24 }
```

"December"

"We are in Winter"

Switch With Ranges.

- In Swift you can use the range of values for checking in case statements. Ranges are identified with "..." in Swift :


```
6  let totalGrade = 3_000
7
8  switch totalGrade{
9
10 case 0...999:
11     println("Fail")
12
13 case 1000...1499:
14     println("Pass")
15
16 case 1500...1999:
17     println("Good")
18
19 case 2000...2999:
20     println("Very Good")
21
22 case 3000...3_500:
23     println("Excellent")
24
25 default:
```

3,000

"Excellent"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Tuples.

```swift
var point = (0, 2)
switch point
{
case (0, 0):
    println("Point is on origin")

case (0,_):
    println("Point is on Y-Axes")

case (_,0):
    println("Point is on X-Axes")

case (-2...2, -2...2): //Rang x and rang y
    println("Point is inside the box")

default:
    println("Point is outside the box")

}
```

Using "_" to ignore values

using ranges

(.0 0, .1 2)

"Point is on Y-Axes"

Switch With Value Binding

You can bind the values of variables in switch case statements to temporary constants to be used inside the case body:

```
var point = (1, 0)

switch point
{
case (0, let y): //Any point with x = 0
    println("Point (0, \(y)) is on Y-Axes")

case (let x, 0): //Any point with y = 0
    println("Point (\(x), 0) is on X-Axes")

case (let x, let y): //Any point, equals to default:
    println("Point (\(x), \(y)) is normal point")
}
```

(.0 1, .1 0)

"Point (1, 0) is on X-Axes"

Switch With "Where"

Another example in using "Where":

```
var sport = "Foot ball"

switch sport
{
case "swimming":
    println("Don't need a ball")

case let x where x.hasSuffix("ball"):
    println("It needs a ball")

default:
    println("Other sport")

}
```

"Foot ball"

"It needs a ball"

Switch With "Where"

Another example in using "Where":

```
var sport = "Foot ball"

switch sport
{
case "swimming":
    println("Don't need a ball")

case let x where x.hasSuffix("ball"):
    println("It needs a ball")

default:
    println("Other sport")

}
```

"Foot ball"

"It needs a ball"

Loops

- Like other languages, you can use for and for-in loops without changes. But in Swift you don't have to write the parentheses.
- for-in loops can iterate any collection of data. Also It can be used with ranges

```
for var i = 0; i < 10 ; i++
{
    println("\(i)")
}

for i in 0..<10
{
    println("\(i)")
}

for i in 0...10
{
    println("\(i)")
}
```

(10 times)

Using range "..<" to omits the upper value from 0 to 9

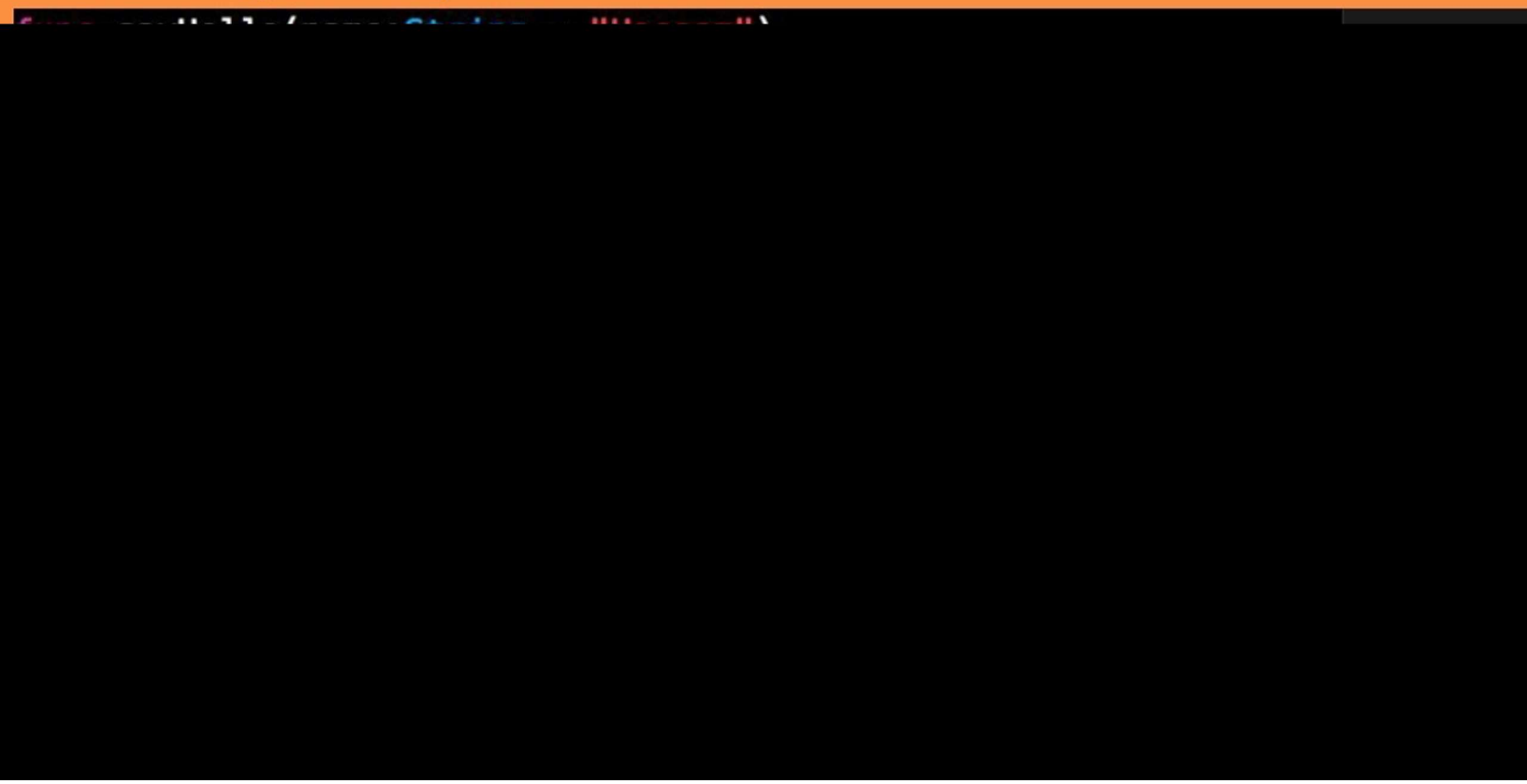
(10 times)

Using range "..." to include the both sides from 0 to 10

(11 times)

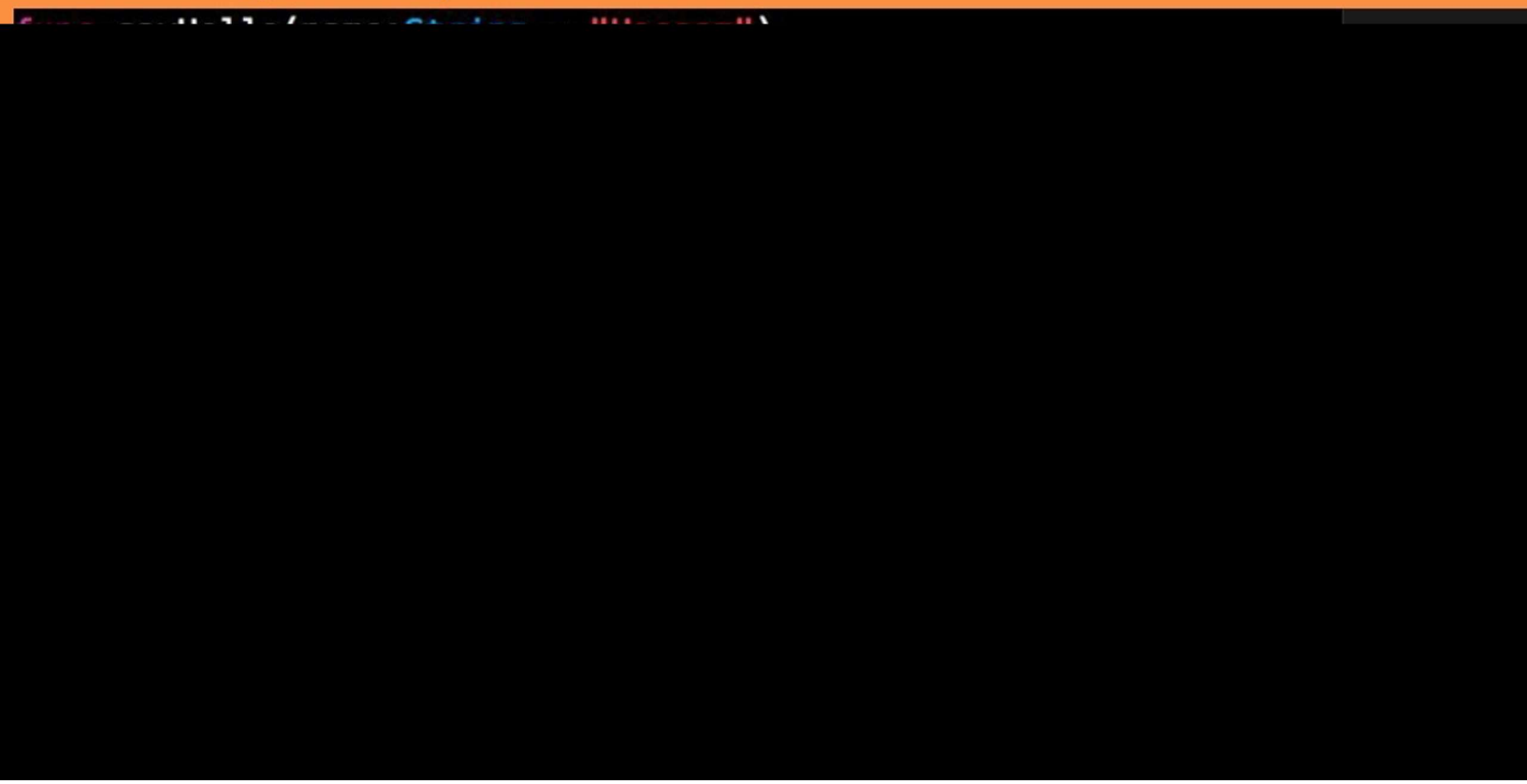
Functions

- Using default parameter value:



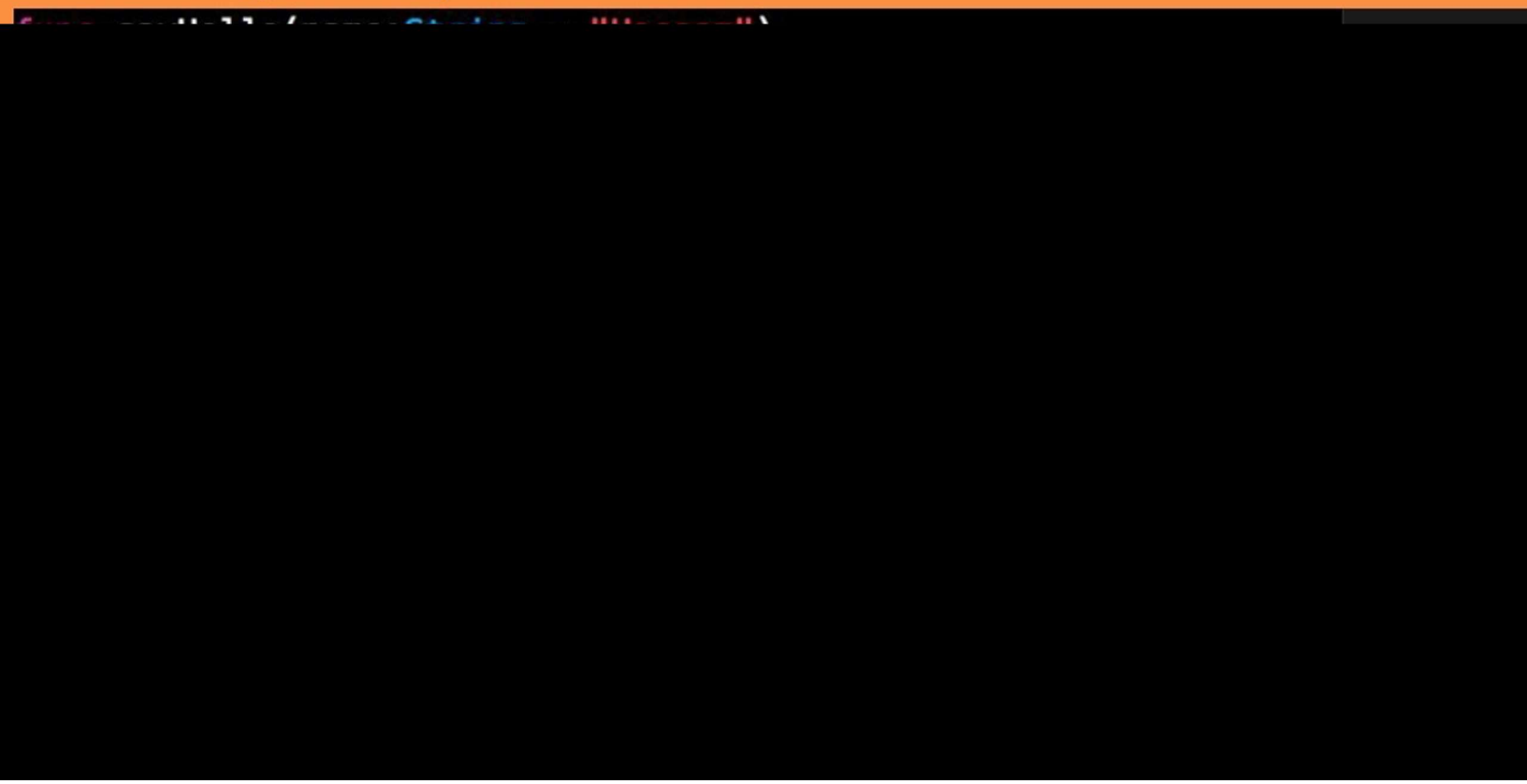
Functions

- Using default parameter value:



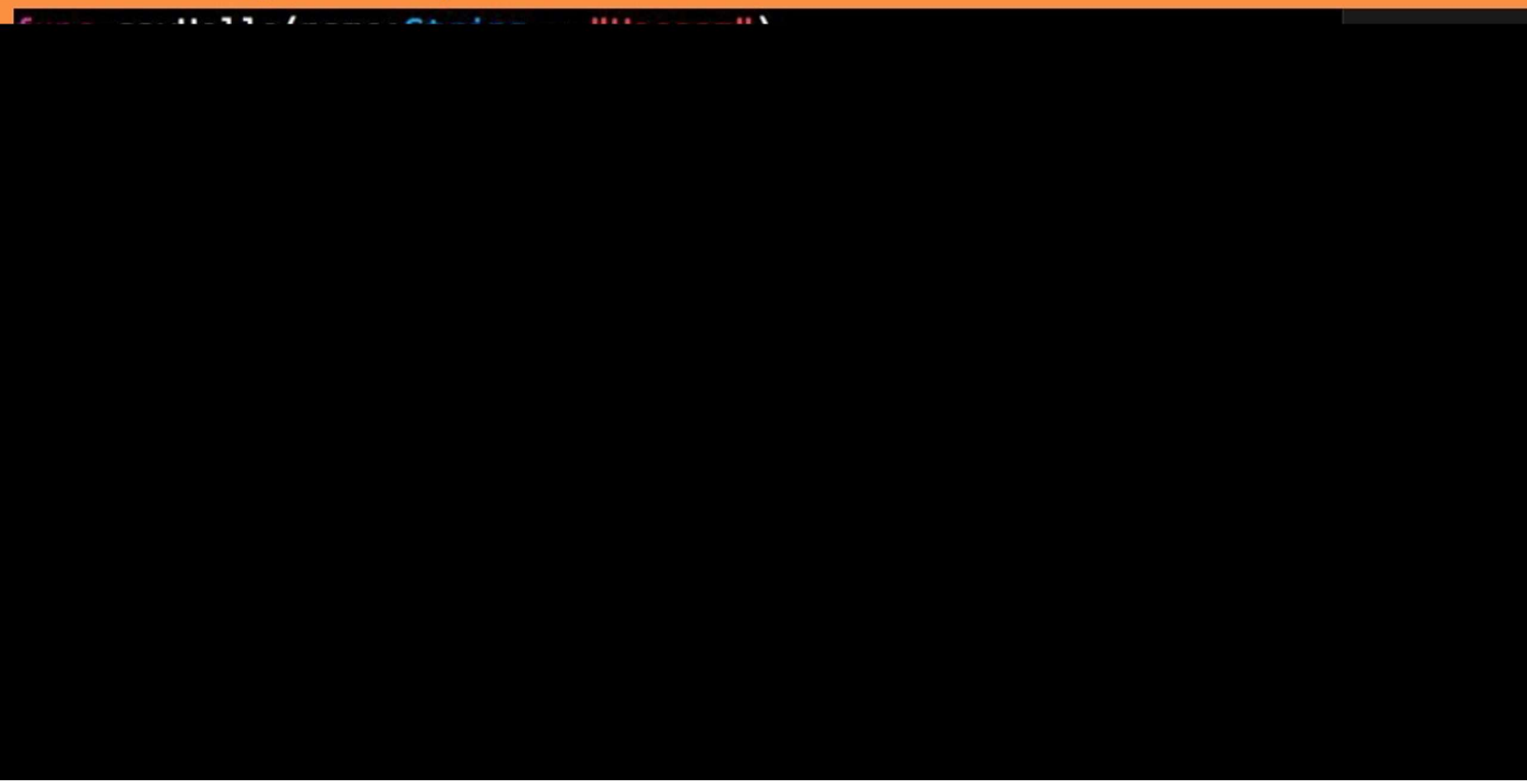
Functions

- Using default parameter value:



Functions

- Using default parameter value:



Passing & Returning Functions

```
func getSuitableFunc(num:Int) -> (Int -> String)
{
    if num < 10
    {
        func print1(x:Int) -> String
        {
            return "The number \(x) is less then
                10"
        }
        return print1
    }
    else
    {
        func print2(x:Int) -> String
        {
            return "The number \(x) is greater
                then 10"
        }
        return print2
    }
}
let function = getSuitableFunc(20)
function(20)
```

```
"The number 20 is greater then 10"

(Function)

(Function)
"The number 20 is greater then 10"
```

Passing & Returning Functions

```
func getSuitableFunc(num:Int) -> (Int -> String)
{
    if num < 10
    {
        func print1(x:Int) -> String
        {
            return "The number \(x) is less then
                10"
        }
        return print1
    }
    else
    {
        func print2(x:Int) -> String
        {
            return "The number \(x) is greater
                then 10"
        }
        return print2
    }
}
let function = getSuitableFunc(20)
function(20)
```

```
"The number 20 is greater then 10"

(Function)

(Function)
"The number 20 is greater then 10"
```

Closures

- Example #3, using the built-in "sorted" function to sort any collection based on a closure that will decide the compare result of any two items

```
var numbers = [100, 3, 30, 4, 6 , 7, 99, 1]

//Decending sorting numbers
sorted(numbers, { (item1, item2) -> Bool in
    return item1 > item2
})
```

[100, 3, 30, 4, 6, 7, 99, 1]

[100, 99, 30, 7, 6, 4, 3, 1]
(19 times)

Closures

- Example #3, using the built-in "sorted" function to sort any collection based on a closure that will decide the compare result of any two items

```
var numbers = [100, 3, 30, 4, 6 , 7, 99, 1]

//Decending sorting numbers
sorted(numbers, { (item1, item2) -> Bool in
    return item1 > item2
})
```

[100, 3, 30, 4, 6, 7, 99, 1]

[100, 99, 30, 7, 6, 4, 3, 1]
(19 times)

Closures

- Example #3, using the built-in "sorted" function to sort any collection based on a closure that will decide the compare result of any two items

```
var numbers = [100, 3, 30, 4, 6 , 7, 99, 1]

//Decending sorting numbers
sorted(numbers, { (item1, item2) -> Bool in
    return item1 > item2
})
```

[100, 3, 30, 4, 6, 7, 99, 1]

[100, 99, 30, 7, 6, 4, 3, 1]
(19 times)

Closures

- Example #3, using the built-in "sorted" function to sort any collection based on a closure that will decide the compare result of any two items

```
var numbers = [100, 3, 30, 4, 6 , 7, 99, 1]

//Decending sorting numbers
sorted(numbers, { (item1, item2) -> Bool in
    return item1 > item2
})
```

[100, 3, 30, 4, 6, 7, 99, 1]

[100, 99, 30, 7, 6, 4, 3, 1]
(19 times)

Arrays

- You can easily iterate over an array using 'for-in' , 'for' or by 'enumerate'. 'enumerate' gives you the item and its index during enumeration.

```
var fruits:[String] = ["Apple",
    "Banana", "Orange"];

for fruit in fruits
{
    println("I love \(fruit)")
}

for (index, fruit) in enumerate
    (fruits)
{
    println("I love \(fruit) at
        index \(index)")
}
```

["Apple", "Banana", "Oran...

(3 times)

(3 times)

Console Output

```
I love Apple
I love Banana
I love Orange
I love Apple at index 0
I love Banana at index 1
I love Orange at index 2
```

Arrays

- You can easily iterate over an array using 'for-in' , 'for' or by 'enumerate'. 'enumerate' gives you the item and its index during enumeration.

```
var fruits:[String] = ["Apple",
    "Banana", "Orange"];

for fruit in fruits
{
    println("I love \(fruit)")
}

for (index, fruit) in enumerate
    (fruits)
{
    println("I love \(fruit) at
        index \(index)")
}
```

["Apple", "Banana", "Oran...

(3 times)

(3 times)

Console Output

```
I love Apple
I love Banana
I love Orange
I love Apple at index 0
I love Banana at index 1
I love Orange at index 2
```

Arrays

- You can easily iterate over an array using 'for-in' , 'for' or by 'enumerate'. 'enumerate' gives you the item and its index during enumeration.

```
var fruits:[String] = ["Apple",
    "Banana", "Orange"];

for fruit in fruits
{
    println("I love \(fruit)")
}

for (index, fruit) in enumerate
    (fruits)
{
    println("I love \(fruit) at
        index \(index)")
}
```

["Apple", "Banana", "Oran...

(3 times)

(3 times)

Console Output

```
I love Apple
I love Banana
I love Orange
I love Apple at index 0
I love Banana at index 1
I love Orange at index 2
```

Arrays

- You can easily iterate over an array using 'for-in' , 'for' or by 'enumerate'. 'enumerate' gives you the item and its index during enumeration.

```
var fruits:[String] = ["Apple",
    "Banana", "Orange"];

for fruit in fruits
{
    println("I love \(fruit)")
}

for (index, fruit) in enumerate
    (fruits)
{
    println("I love \(fruit) at
        index \(index)")
}
```

["Apple", "Banana", "Oran...

(3 times)

(3 times)

Console Output

```
I love Apple
I love Banana
I love Orange
I love Apple at index 0
I love Banana at index 1
I love Orange at index 2
```

Dictionaries

```swift
var emptyDic = [String, Int]() //Empty dic with
    key-value String-Int
var langDic = ["Ar":"Arabic", "EN" : "English",
    "Tr": "Turkish"]
var fullArabicName = langDic["Ar"]
langDic["Sp"] = "Spain" //append or update with key
langDic
langDic["Sp"] = "Spanish" //here it updates
langDic
langDic["Tr"] = nil //Remove with key-value
langDic
langDic.removeValueForKey("Sp")
langDic
langDic.updateValue("ARABIC", forKey: "Ar")
langDic

for (abbreviation, lang) in langDic
{
    println("\(abbreviation) is an abbreviation of
        \(lang)")
}
```

```
0 elements

["EN": "English", "Ar": "Arabic", "Tr": "Turkish"]

{Some "Arabic"}
{Some "Spain"}
["Tr": "Turkish", "Ar": "Arabic", "EN": "English", "Sp": "Spain"]
{Some "Spanish"}
["Tr": "Turkish", "Ar": "Arabic", "EN": "English", "Sp": "Spanish"]
nil
["Ar": "Arabic", "EN": "English", "Sp": "Spanish"]
{Some "Spanish"}   <- return removed value with key
["Ar": "Arabic", "EN": "English"]
{Some "Arabic"}    <- return the old value before update
["Ar": "ARABIC", "EN": "English"]



(2 times)
```

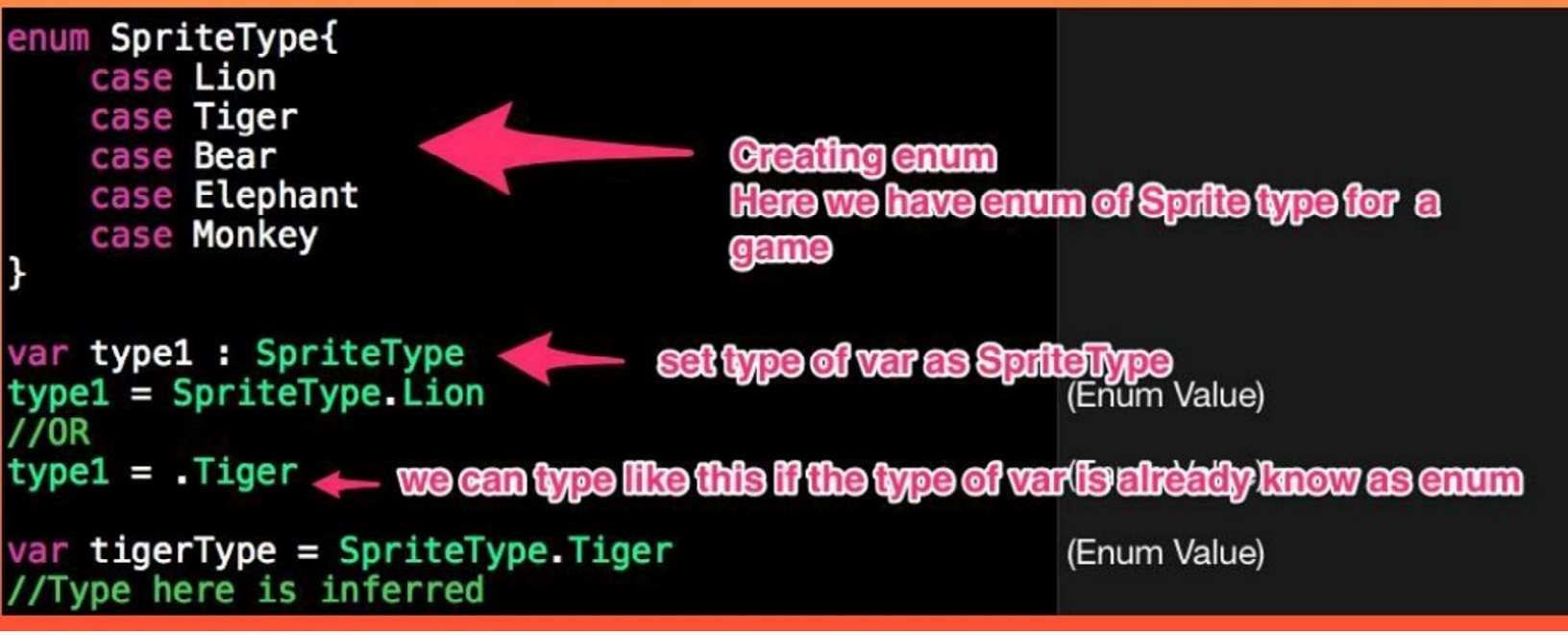
Dictionaries

```swift
var emptyDic = [String, Int]() //Empty dic with
    key-value String-Int
var langDic = ["Ar":"Arabic", "EN" : "English",
    "Tr": "Turkish"]
var fullArabicName = langDic["Ar"]
langDic["Sp"] = "Spain" //append or update with key
langDic
langDic["Sp"] = "Spanish" //here it updates
langDic
langDic["Tr"] = nil //Remove with key-value
langDic
langDic.removeValueForKey("Sp")
langDic
langDic.updateValue("ARABIC", forKey: "Ar")
langDic

for (abbreviation, lang) in langDic
{
    println("\(abbreviation) is an abbreviation of
        \(lang)")
}
```

```
0 elements

["EN": "English", "Ar": "Arabic", "Tr": "Turkish"]

{Some "Arabic"}
{Some "Spain"}
["Tr": "Turkish", "Ar": "Arabic", "EN": "English", "Sp": "Spain"]
{Some "Spanish"}
["Tr": "Turkish", "Ar": "Arabic", "EN": "English", "Sp": "Spanish"]
nil
["Ar": "Arabic", "EN": "English", "Sp": "Spanish"]
{Some "Spanish"}   <- return removed value with key
["Ar": "Arabic", "EN": "English"]
{Some "Arabic"}    <- return the old value before update
["Ar": "ARABIC", "EN": "English"]



(2 times)
```

Enum

- Enum is very popular concept if you have specific values of something.
- Enum is created by the keyword 'enum' and listing all possible cases after the keyword 'case'



Enum

- Enum can be used easily in switch case but as we know that switch in Swift is exhaustive, you have to list all possible cases.

```
switch tigerType{
case .Lion:
    powerOfMonster = 50
case .Tiger:
    powerOfMonster = 40
case .Bear:
    powerOfMonster = 45
case .Elephant:
    powerOfMonster = 60
case .Monkey:
    powerOfMonster = 20
}
```

You will get compile time error if you didn't list all possible cases.
You can use default if you don't want to list all cases

40

Enum With Associated Values

- So we need to represent the barcode with two condition UPC and QR , each one has associated values to give full information.

```
enum BarCode{
    case UPC (Int, Int, Int, Int)
    case QR (String)
}


var product1Code = BarCode.QR("BlaBlaBla")
var product2Code = BarCode.UPC(3, 12443, 12999, 6)

//Associated values with Switch
switch product1Code{
case .UPC(let num1, let num2, let num3, let num4):
    println("Product is in UPC format with nums {\(num1),\(num2),\
        (num3),\(num4)}")
case .QR(let QRDesc):
    println("Proudct is in QR code format with desc \(QRDesc)")
}
```

create with associated values

all barcodes with UPC format

(Enum Value)
(Enum Value)

"Proudct is in QR code format with desc B...

Enum With Associated Values

- So we need to represent the barcode with two condition UPC and QR , each one has associated values to give full information.

```
enum BarCode{
    case UPC (Int, Int, Int, Int)
    case QR (String)
}


var product1Code = BarCode.QR("BlaBlaBla")
var product2Code = BarCode.UPC(3, 12443, 12999, 6)

//Associated values with Switch
switch product1Code{
case .UPC(let num1, let num2, let num3, let num4):
    println("Product is in UPC format with nums {\(num1),\(num2),\
        (num3),\(num4)}")
case .QR(let QRDesc):
    println("Proudct is in QR code format with desc \(QRDesc)")
}
```

create with associated values

all barcodes with UPC format

(Enum Value)
(Enum Value)

"Proudct is in QR code format with desc B...

Enum With Raw Values

- For sure in some cases you need to define some constants in enum with their values. For example the power of monster has different values based on game level (easy = 50, medium = 60, hard = 80, very hard = 120) and these values are constant. So you need to make enum for power values and in same time save these values. You can create enum with cases values but they must be in same type and this type is written after enum name. Also you can use .rawValue to get the constant value.

- You can initialize an enum value using its constant value using this format EnumName(rawValue: value). It returns the enum that map to the given value. Be careful because the value returned is Optional, it may contain an enum or nil, BECAUSE Swift can guarantee that the given constant is exist in enum or not.

Enum With Raw Values Example:

- Raw values can be Strings, Chars, Integers or floating point numbers. In using Integers as a type for raw values, if you set a value of any case, others auto_increment if you didn't specify values for them.

```
enum Gender: Int{
    case Male = 1 //Wil be 1
    case Female  // will be 2
    case Other // will be 3
}

enum HTTPCode: Int{
    case HTTP_Ok = 200
    case HTTP_Created
    case HTTP_Accepted
    case HTTP_BadRequest = 400
    case HTTP_Unauthorized
    case HTTP_PaymentRequired
    case HTTP_Forbidden
    case HTTP_PageNotFound
}

let httpAcceptedCode = HTTPCode.HTTP_Accepted.rawValue
let notFoundCodeValue = HTTPCode.HTTP_PageNotFound.rawValue
```

202
404

Enum With Raw Values Example:

- Raw values can be Strings, Chars, Integers or floating point numbers. In using Integers as a type for raw values, if you set a value of any case, others auto_increment if you didn't specify values for them.

```
enum Gender: Int{
    case Male = 1 //Wil be 1
    case Female  // will be 2
    case Other // will be 3
}

enum HTTPCode: Int{
    case HTTP_Ok = 200
    case HTTP_Created
    case HTTP_Accepted
    case HTTP_BadRequest = 400
    case HTTP_Unauthorized
    case HTTP_PaymentRequired
    case HTTP_Forbidden
    case HTTP_PageNotFound
}

let httpAcceptedCode = HTTPCode.HTTP_Accepted.rawValue
let notFoundCodeValue = HTTPCode.HTTP_PageNotFound.rawValue
```

202
404

Thanks!

If you liked the tutorial, please share and tweet with your friends.

If you have any comments or questions, don't hesitate to email ME