Xcode & IOS development

I. Introduction to Swift language

I. Introduction to Swift language

Swift...

- Swift is the programming language for iOS, OSX, watchOS, ...
- Open source, it is an object language integrating mechanisms from objective C, C++ and Python
- It uses the named parameters of objective-C for a "better readability" and to facilitate overriding methods
- The basic example: "*Hello World*"
  - open your Ipad
  - create a playground
  - test the example : `print("Hello, world!")`

Variables and constants

- `let` to define constants
- `var` to define variables

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

- Dynamic typinf

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

- No implicit conversion ⟹ need for explicit conversion

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

Arrays and Dictionaries

- Arrays and Dictionaries use [ ]

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
var occupations
=["Malcolm":"Captain","Kaylee":"Mechanic"]
occupations["Jayne"] = "Public Relations"
```

- Arrays in swift are object but value types
- Array are created by calling init method:

```
var tab : [Int] =[]
var tabint = [Int]()
var tabinit = [Int](repeating: 0, count: 10)
tabinit.count
```

Control statements

We find the same as usual: `if`, `switch`, `for-in`, `for`, `while` and `repeat-while`

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

```
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.")
}
```

Switch-case of Swift is richer than the one of C or other language:

- stop as soon a case is solved
- pattern for testing a case are possible
- manage intervals : `case 1..<5`

for-in statement is similar to the one of Python :

```
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 {
    print("kind=\(kind), number=\(number)")
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
```

Optional values

- Specific to Swift (similar feature in Kotlin and Typescript)
- Used to indicate that a variable or function may not have a value
- Is indicated by the char `?`
- Similar to `nil` or `NULL` but valid for any variable, whatever its type, even Int or Float
- Defines a different type : `Int?` is not the same type that `Int`

```
var optionalName: String? = nil // (String | Vide)
var greeting = "Hello!"
if (optionalName != nil) {
    greeting = "Hello, \(optionalName!)"
}
var name: String = optionalName
error, can’t assign optional String to String
```

Optional values

- By default an optionnel value is nil
- It is possible to force évaluation of an optionnel value with char `!`
- Some opérations (e.g. cast) may not return a value ⇒ require to use optional values

```
let snumber = "123"
let n : Int? = Int(snumber)
if (n != nil) {
  print("snumber converti contient une valeur entière")
}
```

- Optional binding can be used to control if an optional variable has a value:

```
let tnumber = "123"
if let nn : Int = Int(tnumber) {
  print("tnumber converti contient \(nn)")
}
else{
  print("\(tnumber) n'a pas pu être converti")
}
```

`guard` to check optionals

guard operator is initially used to handle error cases.

It provides an elegant way to handle cases where a variable might not have a value and cause an error.

Let's compare the 3 ways of handling a value that might be undefined:

- "the old way", i.e. as in C, Java or Objective C ;
- by using the conditional assignments of Swift ;
- with the guard operator.

Note: guard requires in its else clause to exit from block code or function, by using break, continue, return or throw (*or possibly by a function like some error functions*)

Method used in traditional languages

```
var x : Int? = some_func()
if (x==nil) || (x!<=0) { // to compare x (Int?) to 0 (Int),
we need to force évaluation of x to avoid a compilation error
  return // no valuable value ⇒ manage this error case
}
// doing something with x value
x!.description
```

Drawbacks :

- the condition checks for unwanted values: this can be confusing, especially if there are multiple nested checks
- the optional value x will have to be unwrapped in all the rest of the code

Method with optional binding

```
var x : Int? = some_func()
if let x=x, x>0 {
    // do something with x value
    x.description
}
// no valuable value, manage this error case
return
```

Previous drawbacks have been removed but now the code that should run normally is inside a if-then block.

This can make readability worst, and especially becomes complex if there are multiple nested checks.

It is always better to check all pre-conditions at the beginning of a function.

Method with *guard*

```
var x = some_func()
guard let x=x, x>0 else {
    // no valuable values, manage this error case
    return
}
// do something with x value
x.description
```

Advantages :

- the error is checked
- error cases can all be checked independently at the beginning of the function, the rest of the code being the function itself
- after the guard, the x variable is now considered as non-optional and does not need to be unwrapped

Functions

- `func` is used to define a function
- calling a fonction is done classically by using fonction name
- parameter are named, and the call must include parameter names
- type of fonction is introduced by an arrow `->`

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

- tuples allow to return multiple values
- functions can be defined with a variable number of parameter which will then be collected in an array

```
func calculateStatistics(scores: UInt...)->(moyenne:Float,
min: UInt, max: UInt) {
  var min : UInt = scores[0]
  var max : UInt = scores[0]
  var sum : UInt = 0
  for score in scores {
    if score > max { max = score}
    else if score < min { min = score }
    sum += score
  }
  return (Float(sum)/Float(scores.count),min, max)
}
let resultats = calculateStatistics(scores: 5, 3, 100, 3, 9)
print(resultats.moyenne)
print(resultats.2)
```

functions

- parameters can have a different local name than the calling name:

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

- functions can be nested
- functions can takes functions as parameter
- fonction can return functions

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

Closures

- functions are in fact particular cases of « Closures »
- Clôtures are kind of anonymous functions
- Written inside `{ }` , code is separated from parameter by keyword `in`

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

- type of return value or parameters can be omitted

```
let mappedNumbers = numbers.map({ number in 3 * number })
```

- On peut même se référer aux paramètres par leur numéro, et en dernier paramètre on peut omettre les ( )

```
let sortedNumbers = numbers.sorted(by: { $0 > $1 })
```

Objects

- `Class` to define an object
- *properties* are defined like are define variables and constants
- *methods* are defined like are defined functions
- *dot notation* is used to access to methods and properties
- `init` is initializer of instances: it is not a *constructor*!
- `deinit` to deinitializer: beware not a *destructor*!
- `self` is used to distinguish parameter and local variables from methods and properties
- properties can define specifics *getter* and *setter*

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String
    init(name: String) {
        self.name = name
    }
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

- inheritage is possible (classical notation '`:`')
- `override` can be used to override methods *and* properties
- to forget `override` throws an error
- a property can be computed and an override property can override a stored property, the contrary is not possible.

```
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)
triangle.perimeter = 9.9
```

- You can add *property observers*

```
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)
    }
}
```

- Caution: access to properties of a class overriding properties of a superclass having observers triggers also call to observers

- Properties and Constants Class :
    - defined with keyword `static`.
    - one can also used keyword `class` so they can be overrided.

```
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        // return an Int value here
    }
    class var overrideableComputedTypeProperty: Int {
        // return an Int value here
    }
}
```

Protocols : abstract class

- Protocols define an abstract type
- Any class can implement a Protocol
- A class can implement several protocols

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

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

Extensions

- Extensions add new functionality to an existing class, structure, enumeration, or protocol type.

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

Generics

- type or function can be generic:

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

- Constraints on generic types:

```
func anyCommonElements <T: SequenceType, U: SequenceType where
T.Generator.Element: Equatable, T.Generator.Element ==
U.Generator.Element> (_ lhs: T, _ rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem { return true }
        }
    }
   return false
}
```