

14/06/2015

Swift essentials.

①

Structure

- ```
//
```

  

```
/* ... */
```

 } Comments.
- NO semicolons needed for every statements
- no main method or main function to start

### Playground

- You can see the graph by clicking on 'O' to the output.
- see 1-1 Swift Playground.

### Declaring Variables

see 1-2 Variables

- Variables are declared by keyword "var"
- Swift is tightly data typed language.
- var can take bool, int, str
- when you type something the compiler "Infer" the data type
- once declared you cannot change the data types.
- if you just write

var name

- ~~you~~ you will get an error as swift cannot understand what type of variable it is i.e. int, str etc..
- so you need to explicitly tell it

### Creating Constants

see 1-3 ~~var~~ constants.

- you can create constants by using ~~var~~ "let"

- You cannot change the constant data type implicitly or explicitly.
- everything else is same as ~~var~~ 'Var'
- You cannot ~~even~~ change the content of it
- It is highly recommended to use let if you know that something does not change.

Print and string interpolation  
see 1-4- Print and string interpolation.

→ to print o/p we use ~~print~~  
println()

→ You can use  
\()

for string interpolation

Converting values  
see 1-5- Converting values.

- Swift does not convert values implicitly.
- You can convert var by wrapping around with data types

Double()  
String()  
Int() etc..

if statements & switch case  
see 1-5- if and switch case

→ if condition should always be bool

②  
→ You can have ranges in switch by doing

case 0...3:

— — —

loops

see 1-7-loops

→ while, do...while, & for.

→ There are two types of "for" loops

for

for...in

→ range operators

$0...100 \Rightarrow 0$  to  $100$  including  $0$  &  $100$

$0..<100 \Rightarrow 0$  to  $99$  including  $0$  but not  $100$

functions

see 1-8-functions.

→ keyword func

→ An input parameter to a function is always a constant

→ to make it mutable add & for input string

16/6/2015

Complex types

Arrays

see 2-1-arrays



- The Indexing of array starts with '0'.
- arrays are type specific eg. str or int
- If you use var then the array is mutable, 'List' is immutable.

Syntax:

a = ["a", "b", ...]

b = [1, 2, ...]

- If you just want a variable where you can put data later of type specific following is the syntax

var some : [String]

- to insert at the end use

some.append("text")

- to insert at some pos. do

some.insert("text", atIndex: —)

- to remove at last

some.removeLast()

- to remove at index pos

some.removeAtIndex(3)

- Count using .count

some.count

## Dictionary

see 2-2-Dictionary

- unlike array dict can have custom index called key and its value.
- Its not type specific that means the key & value can be a combination of str: int or int: str or int: int etc..

### Syntax!

Var some = [ "some1" : "some" ]  
                   ↑                  ↑  
                  key              value.

empty dict

Var some2 : [String: String]

- update or insert: if there is a key ~~with the~~ then the dict is updated and if not the key is inserted.

some ["some1"] = "some some"

or

some.updateValue("some some", forKey: <sup>some1</sup> ~~"some1"~~)

- to delete just making the key equal to nil eg.

some ["some1"] = nil  
                   (or)

some.removeValue(forKey: "some1")

→ You can count by using .count.

→ iter

```
for (key, value) in some {
```

```
 println(" \key) and \value)
```

```
}
```

Tuple

See 2-3-tuple

→ a combination of multiple types.

eg. var some = ("some1", 1)

→ see 2-3-tuple for more examples of functions.

~~Optionals~~

using optionals

see 2-3-optionals.

→ you can use optionals when you don't know what to initialize by ~~you~~ you still want to use it.

Syntax:

var temp: Int?

or  
optionals.

→ If you force-unwrap an optional var you will get a runtime error, always check and be positive and use if condition.

④  
→ You can force unwrap an optional by using '!', use only if you know that there would be an off  
eg.

```
println("some!(*some!)")
```

→ if used with if let you don't have to use '!' see  
the example.

### Enumerations

~~see~~ see "2-5\_enums"

→ Option click on something to find what type of enum it is.

→ You can add your own methods to enums see the example.

### Closures

See "2-6-1\_closures" & "2-6-2\_closure\_example"

→ ~~it~~ it is used to group code <sup>into</sup> as a self-contained units and ~~reusable code~~.

→ Functions are types of closures.

→ also known as lambdas. (λ)

### Classes

Def a class and instantiate it  
See "3-1\_Def\_class"



## Syntax

class Name {

Properties & methods

}

Properties could be var or lets

methods are functions.

calling can be do using dot notation Name.var...

## Adding initialisers

See 3.2 initialisers

→ You can initialise value by using

~~init~~ init() {

}

→ You can de initialise by using

~~deinit~~ {

deinit {

}

}

## Inheritance

see 3.3 inheritance

→ reusing the code from base class (superclass) to subclass.

See the Example.

→ You can close the class by doing final class name { ... }



## Computed properties

⑤

see 3-4-Computed property.

→ They are the read only properties which can do computation when called like

+ , \* , / etc...

## Type properties

See 3-5-Type properties

→ They are static properties that can be written in two ways

eg:

```
class Var some : String {
 return "some"
}
```

(or)

```
static Var some : String = "some"
```

→ You cannot call it through instances.

→ You can only call it through ~~the~~ class.

ClassName.some.

## lazy Property

see 3.6-lazy-property

→ The variable is not accessed unless asked for.

Syntax

lazy var some = "some1".

→ lazy are always vars

## Property Observers

see 3.7-Property-Observers

→ when you <sup>think</sup> that a property ~~to~~ value changes then you can put property observer to a default property

```
var some = "some1" { willSet {
 ... (newValue)
}
```

```
 didSet {
 ... (oldValue)
 }
}
```

→ This will show the changes compared to ~~an~~ oldValue to newValue.

## Access modifiers

See 3.8 access modifier

→ There are three of these

Public → accessible within the source code file.  
 Private → ~~can~~ any code outside the module/project

Internal - access by multiple code files, but must be compiled as a single module. (Default in Swift)

Public class Name {

Public func name1() {

3

Private func name2() {

3

}

## Taking it further

### working with structures

see 4-1-working-with-structures

→ ~~everything~~ <sup>everything</sup> you see in swift is a structure not a class

string, array, Dict.

~~see 4-1~~

see 4-1-3-~~structure~~ for structure  
4-1-3-1-structure

→ when you have struct you get memberwise initialiser

see 4-1-3-2-structure.

That means you can initialise the value of an var's  
~~when~~ while instantiating it.

→ You cannot inherit from other classes.

→ no deinit for structs

## Operators

see 4-2-Operators

### Advance operators

0...100 range b/w 0 & 100 including 0 and 100  
0...<100 range b/w 0 & one less than 100  
including 0 but not 100



nil coalescing operators

7

??  $\Rightarrow$  see 4-3-nil-coalescing.

The % sign in swift works with

negative } numbers,  
decimal }

Importing frameworks by using objective-c class

System/Library/Frameworks

$\uparrow$   
location.

import Foundation.

see 4-4-importing.

Advance language features

type checking & down casting

see 5-1-TypeCheckingAndDownCasting.

Any object and ~~Object~~ Any

see 5-2-AnyObjectAndAny

AnyObject  $\rightarrow$  has to be an object, Any -.

Any  $\rightarrow$  Any type - both object and nonobject

## Syntax

var some: [AnyObject] ←

var some1: [Any] ←

var some2: AnyObject

var some3: Any

} array

→ You can put any combination of types

## Protocols

see 5-3-Protocol

→ Every class can access protocol

→ You can have multiple protocols.

→ You can use protocols by returning something to it  
see the docs.

## Extensions

see 5-4-extensions

→ Add methods and properties to existing types.

→ You can use your existing code to control the flow or the way it looks for a data type by using extensions

## Generics

see 5-5-generics

→ A function can take any data type as an input and give an output of any data type by keeping its data type as is unlike AnyObject