Smartphone programming Class - 3

short line

# Swift Programming Language:

## **Constants and Variables:**

Constants are declared as:

Constant: let thisIsAConstant = 10

Variable: var thisIsAVariable = "String Variable"

Declare multiple constants or variable in a single line:

var x = 0.0, y = 0.0, z = 0.0

## **Type Annotation**:

You can declare variables and have type associated with it:

var typeAnnotedVariable: String

Multiple constants or variable can be declared too:

var x, y, z: Double

## **Variable names:**

They can be anything including unicode:

let ⌚ = "watch"

## **Printing:**

var iWantToPrintThis = "Some variable"

print("My Name is Ashish")

print("This is how to print variable \(iWantToPrintThis)")

## **Comments:**

// This is a comment.

/\* This is also a comment

but is written over multiple lines. \*/

/\* This is the start of the first multiline comment.

/\* This is the second, nested multiline comment. \*/

This is the end of the first multiline comment. \*/

## **Semicolons:**

You don't need to write semicolon but if you wish you can use them. You need semicolon if you are writing multiple statements in same line

let cat = "🐱"; print(cat)

## **Integers:**

Integers can be signed or unsigned and they can be 8, 16, 32 and 64 bits. We can declare values like UInt8, UInt16 etc. Bounds can be got by min and max values.

let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8

let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8

## **Floating point numbers**:

**Double** represents a 64-bit floating-point number.

**Float** represents a 32-bit floating-point number.

## **Type Inference**:

If there is no ambiguity, we dont need to specify the type of variable or constant. Swift will figure out the type automatically.

## **Numeric Literals:**

|  |  |  |
| --- | --- | --- |
| decimal | No prefix | let decimal = 15 |
| binary | 0b | let binary = 0b1111 |
| octal | 0o | let octal= 0o17 |
| hexadecimal | 0x | let hexadecimalInteger = 0xF |

## **Booleans:**

Booleans are defined as Bool their values can be true or false

let booleanVal = true

## **Tuples**:

Tuples can group multiple values into one value

let tupleValue = (404, "Not Found")

print("Key = \(tupleValue.0)")

print("value = \(tupleValue.1)")

let (intCode, status) = (404, "Not Found")

print("Key = \(intCode)")

print("value = \(status)")

## **Optionals (?)**:

Optionals are used when the value might not be present.

let optionalVar?

optionalVar = 5;

print("Optional Var = \(optionalVar!)"")

Lets say if the optional was not initialized the above line would crash.

**Forced Unwrapping !**:

When we know that the value is definitely present we can write ! to force unwrap and print the value

Generally it is not good idea to do forced unwrapping, we check the value before force unwrapping:

* if optionalVar!= nil {
* print("Optional Var = \(optionalVar!).")
* }

Or we can have the value inside a variable and then print it out:

* if let actualNumber = Int(possibleNumber) {
* print("The string \"\(possibleNumber)\" has an integer value of \(actualNumber)")
* } else {
* print("The string \"\(possibleNumber)\" could not be converted to an integer")
* }

## **Error Handling:**

When you know that a function might have errors or crash you can add throw in the declaration. If someone is using a function that throws they should have the function call inside a try catch

* func canThrowAnError() throws {
* // this function may or may not throw an error
* }
* do {  
   try canThrowAnError()  
   // no error was thrown
* } catch {
* // an error was thrown
* }

## **Assertions and Preconditions:**

While Development you can use asserts to find if there are errors:

* let age = -3
* assert(age >= 0, "A person's age can't be less than zero.")

Preconditions are used in production instead of assert

precondition(age > 0, "Index must be greater than zero.")

## **Basic Operators:**

|  |  |  |
| --- | --- | --- |
| Assignment | = | a=b  let (x, y) = (1, 2) |
| Arithmetic operators | +, - , \* , /, % | "hello, " + "world" |
| Unary Minus | -(variable) | Negates the value |
| Compound Assignment | += | A += 5 |
| Comparison | ==, !=, > , < , >=, <= |  |
| Closed range operator | (a...b) | for index in 1...5 |
| Half open Range | (a..<b) | for index in 1..<5 |
| One Sided Ranges | [a…]  [..<5] | * for name in names[2...] { * print(name) * } |
| Logical Operators | ! && || |  |

## **String and Characters:**

let someString = "Some string value"

Multi line strings can be written with three quotations:

let multiLine = """

This is a multiLine String "You can have Quotations,

inside this?" he asked.

"And Blank Lines too," all of this, "should be

enclosed inside triple quotations."

"""

let singleLineString = "These are the same."  
let multilineString = """  
These are the same.  
"""

Finding if string is empty:

if emptyString.isEmpty {  
 print("Nothing to see here")  
}

Character arrays can be converted to string

let dogArray: [Character] = ["D", "o", "g", "🐶"]

let dogString = String(dogArray)

print(dogString)

Concatenation:

* let string1 = "First"
* let string2 = " Second"
* var combined = string1 + string2

Append:

combined.append(“!”)

Counting Characters:

print(“Char count = \(combined.count)”)

Substring:

let greeting = "Hello, world!"

let index = greeting.firstIndex(of: ",") ?? greeting.endIndex

let beginning = greeting[..<index]

// beginning is "Hello"

// Convert the result to a String for long-term storage.  
let newString = String(beginning)

## **Collection Types:**

Arrays:

var intArray= [Int]()

intArray.append(5)

// intArraynow contains 1 value of type Int  
intArray= []

// intArrayis now an empty array, but is still of type [Int]

Creating array with repeating default value

var intArray = Array(repeating: 5, count: 3)

isEmpty property can check if the array is empty

Iteration:

* for item in shoppingList {
* print(item)
* }

## **Set (Hash Set):**

Var animals Set<String> = ["Dog", "Cat", "Horse"]

## **Dictionary (Hash Table):**

var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]

for (airportCode, airportName) in airports {  
 print("\(airportCode): \(airportName)")

}

## **Functions with Default Values:**

* func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {  
   // If you omit the second argument when calling this function, then
* // the value of parameterWithDefault is 12 inside the function body.
* }

## **Function Types:**

* func addTwoInts(\_ a: Int, \_ b: Int) -> Int {  
   return a + b  
  }
* func multiplyTwoInts(\_ a: Int, \_ b: Int) -> Int {  
   return a \* b
* }

var mathFunction: (Int, Int) -> Int = addTwoInts

* print("Result: \(mathFunction(2, 3))")  
  // Prints "Result: 5"
* mathFunction = multiplyTwoInts
* print("Result: \(mathFunction(2, 3))")  
  // Prints "Result: 6"

## **Function Types as Parameter types:**

func printMathResult(\_ mathFunction: (Int, Int) -> Int, \_ a: Int, \_ b: Int) {

print("Result: \(mathFunction(a, b))")

}

printMathResult(addTwoInts, 3, 5)

## **Closures:**

Functions have following template

func functionName( parameter: parameter type) -> returnType{

// Do something

return something

}

We can have a function as input of another function and a function as output to another function too.

* func addTwoInts(\_ a: Int, \_ b: Int) -> Int {  
   return a + b  
  }
* func multiplyTwoInts(\_ a: Int, \_ b: Int) -> Int {  
   return a \* b
* }
* We are passing a function as a parameter
* func Calculate(\_ a: Int, \_ b: Int, operation: (Int, Int) -> Int ) -> Int {  
   return operation(a,b);
* }
* Calculate(a:2, b:3, Operation:addTwoInts)
* Calculate(a:2, b:3, Operation:multiplyTwoInts)

Closure is anonymous function. If there is a function then it is written as follows:

Ifunc functionName( parameter: parameter type) -> returnType{

return something

}

We remove the func key word and function name and make it

* { (parameters) -> return type in
* statements
* }

This following function becomes:

* func multiplyTwoInts(\_ a: Int, \_ b: Int) -> Int {  
   return a \* b
* }

{( a: Int, b: Int) -> Int in   
 return a \* b

}

With Type inference we can remove the input params and output param too

{( a, b) in   
 return a \* b

}

We can make it shorter too by removing return, compiler will figure out return value.

{( a, b) in   
 a \* b

}

The first param can be referred as $0 and second as $1 so closure becomes: {$0 \* $1}

Let result = resultCalculator(a:2, b:3, {$0 \* $1 })