Smartphone programming Class - 4

short line

# Swift Programming Language: Functions

# Declaring and calling a function:

You can declare a function by keyword ‘func’ and then provide the function name the syntax of a function with no argument list is as follows:

func **SimpleFunctionNoArguments**(){

print("This is a simple function which has no arguments")

}

# Function with Arguments:

In following example there is a function with arguments

func FunctionWithArguments(intArgument: Int, stringArgument: String){

print("Integer Argument = \(intArgument)")

print("String Argument = \(stringArgument)")

}

FunctionWithArguments(intArgument: 5, stringArgument: "Hello Seattle!!")

# Function with Arguments and return value:

func FunctionWithArgumentsAndReturn( name: String) -> String{

return "Hello \(name) welcome to Seattle!!!"

}

print(FunctionWithArgumentsAndReturn(name: "Ashish"))

# Function with multiple return values:

func FindSmallestSecondSmallest(array: [Int])->(smallest: Int, secondSmallest: Int){

var smallest = min(array[0], array[1])

var secondSmallest = max(array[0], array[1])

for value in array[2...array.count-1]{

if value < smallest{

secondSmallest = smallest

smallest = value

}

else if value < secondSmallest {

secondSmallest = value

}

}

return (smallest, secondSmallest)

}

let array = [1,-2,-3,4,5,7,2,5,6,2]

let twoValues = FindSmallestSecondSmallest(array: array)

print(twoValues.smallest)

print(twoValues.secondSmallest)

# Argument Labels:

You can add a label to the argument separated by a space

func Multiply(a: Int, by b: Int)->Int{

return a \* b

}

print(Multiply(a: 3, by: 4))

# Optional Argument Labels while calling function:

If you do not want to have argument labels while calling the function you can add underscore \_ before the variable name.

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

return a + b

}

print(AddNumbers( 3, 4))

# Variadic Parameters:

A variadic parameter accepts zero or more values of a specified type.

func countOccurancesVariadicParams(\_ numbers: Int..., count x: Int) -> Int {

var count = 0

for number in numbers {

if number == x {

count += 1;

}

}

return count

}

let val = countOccurancesVariadicParams(2,3,4,5,6, count: 2)

# In Out parameters:

You can pass arguments as a reference too by adding “inout” in the argument, when you call the function you would have to add an & in front of the argument:

var a = 5

var b = 10

func swapTwoInts(\_ a: inout Int, \_ b: inout Int) {

let temp = a

a = b

b = temp

}

swapTwoInts(&a, &b)

print("Value of a = \(a)")

print("Value of b = \(b)")

# Function Types:

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

return a + b

}

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

return a \* b

}

The type of both of these functions is (Int, Int) -> Int.

A function that has two parameters, both of type Int, and that returns a value of type Int.

func SimpleFunctionNoArguments(){

print("This is a simple function which has no arguments")

}

Type of above function is () -> Void

var Calculate: (Int, Int) -> Int = AddTwoInts

// Define a function called calculate which is of Type (Int, Int) -> Int

// Now we can assign this variable to a function which has same type

// Potentially we can assign this to AddTwoInts and MultiplyTwoInts

print(Calculate(7,4))

// This will print 11 since Calculate is currently assigned to AddTwoInt function

// Now dynamically based on some other condition we can assign the same

// function to multiply too

Calculate = MultiplyTwoInts

print(Calculate(7,4))

// Now result would be printing 28 since now the Calculate function is assigned to Multiply

# Functions as Parameter Types:

We can pass Function Type as a parameter too so in above condition we can have a function which will take function type

func PrintResultOfCalculate( a: Int, b: Int,\_ funcName: (Int, Int)->Int ){

let result = funcName(a,b)

print(result)

}

PrintResultOfCalculate(a: 3,b: 4, AddNumbers)

# Return Types as functions:

var totalMoney = 100000

func WinMoney(amount: Int) -> Int{

return totalMoney + amount;

}

func LooseMoney(amount: Int) -> Int{

return totalMoney + amount;

}

func DoTransaction(win: Bool) -> (Int) -> Int {

return win ? WinMoney: LooseMoney

}

func chooseStepFunction(backward: Bool) -> (Int) -> Int {

func stepForward(input: Int) -> Int { return input + 1 }

func stepBackward(input: Int) -> Int { return input - 1 }

return backward ? stepBackward : stepForward

}

var currentValue = -4

let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

// moveNearerToZero now refers to the nested stepForward() function

while currentValue != 0 {

print("\(currentValue)... ")

currentValue = moveNearerToZero(currentValue)

}

print("zero!")

// -4...

// -3...

// -2...

// -1...

// zero!

# Closures:

This is format of closure

{ (parameters) -> return type in

statements

}

Basically we remove keyword func and function name and move the curly braces in start and add key word in

For example, let’s say we have a function Add like this:

func Add(a: Int, b: Int)-> Int {

return a + b

}

let sum = Add(a: 3, b: 4)

Instead we can do this:

func MathCalculate(a: Int,b: Int,functionName: (Int, Int)-> Int) -> Int{

return functionName(a,b)

}

let test = MathCalculate(a: 2, b: 3) { (no1, no2) in no1 \* no2 }

//let test = MathCalculate(a: 2, b: 3, functionName: { $0 \* $1 })

// Trailing closure if last param is a closure then you can omit the function name

let test = MathCalculate(a: 2, b: 3) { $0 \* $1 }

func AddOne(a: Int) -> Int{

return a + 1

}