Functions

Outline

1. Function Declarations
2. Function Usage
3. Function Types
4. Nested Functions

1. Function Declarations

- Functions in Swift are declared using the **`func`** keyword:

```
func double(x: Int) -> Int {
    return x * 2
}
```

2. Function Usage

1. Calling Functions
2. Parameters
3. Return Values

2.1 Calling Functions

- Calling functions uses the traditional approach:

```
val result = double(2)
```

- Calling member functions uses the dot notation:

```
Sample().foo()
```

2.2 Parameters

- Functions can have no parameters, multiple parameters or variadic parameters

```
// No parameters
func sayHello() {
    print("Hello")
}

// Multiple parameters
func addTwoInts(a: Int, b: Int) -> Int {
    return a + b
}

// Variadic parameters
func addMultipleInts(numbers: Int...) -> Int {
    var sum = 0
    numbers.forEach { sum += $0 }
    return sum
}
```

2.2 Parameters (cont)

- **Argument Labels**

❖ Each function parameter has *an argument label*. It should be unique to make code more readable.

```
func greet(person: String, from hometown: String) {
    print("Hello \(person)! Glad you could visit from \(hometown)")
}
greet(person: "Ivan", from: "Russia")
// Hello Ivan! Glad you could visit from Russia
```

2.2 Parameters (cont)

- **Argument Labels**

❖ If you don’t want an argument label for a parameter, write an underscore (_) instead of an explicit argument label for that parameter.

```
func someFunction(_ firstParameter: Int, secondParameter: Int) { ... }
someFunction(1, secondParameter: 2)
```

2.2 Parameters (cont)

Default Parameter Values

- You can define a default value for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
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.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6)
// parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```

2.2 Parameters (cont)

- **In-Out Parameters**

❖ Function parameters are constants by default. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, defined that parameter as an *in-out parameter* instead.

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}
```

2.3 Return Values

- Functions can return no values, single value or multiple values

```
// Funtion without return values
func greet(person: String) {
    print("Hello \(person)!")
}

// Function return single value
func addTwoInts(a: Int, b: Int) -> Int {
    return a + b
}

// Function return multiple values
func minMax(array: [Int]) -> (min: Int, max: Int) {
    guard let min = array.min(),
        let max = array.max() else { return (0, 0) }
    return (min, max)
}
```

2.3 Return Values (cont)

- If the entire body of the function is single expression, the function implicitly returns that expression.

```
func greeting(for person: String) -> String {
    "Hello, \(person)!"
}

func anotherGreeting(for person: String) -> String {
    return "Hello, \(person)!"
}

print(greeting(for: "Jason")) // Hello, Jason!
print(anotherGreeting(for: "Jason")) // Hello, Jason!
```

3. Function Types

- You use function types just like any other types in Swift.

```
var mathFunction: (Int, Int) -> Int
```

3. Function Types (cont)

- You can use a function type as a parameter type for another function.

```
func printResult(_ a: Int, _ b: Int, _ mathFunction: (Int, Int) -> Int) {
    print("Result: \(mathFunction(a, b))")
}
printResult(3, 5, addTwoInts) // Prints "Result: 8"
```

3. Function Types (cont)

- You can use a function type as the return type of another function.

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

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

func chooseStep(_ backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}

var currentValue = 3
let moveNearerToZero = chooseStep(currentValue > 0)
// moveNearerToZero new refers to the stepBackward() function
```

4. Nested Functions

- You can also define functions inside the bodies of other functions, known as *nested functions*.

```
func chooseStep(_ 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 = -10
let moveNearerToZero = chooseStep(currentValue > 0)
// moveNearerToZero new refers to the stepForward() function
```

Question & Answer?