Chapter 18: Functions

Section 18.1: Basic Use

Functions can be declared without parameters or a return value. The only required information is a name (`hello` in this case).

```
func hello()
{
    print("Hello World")
}
```

Call a function with no parameters by writing its name followed by an empty pair of parenthesis.

```
hello()
//output: "Hello World"
```

Section 18.2: Functions with Parameters

Functions can take parameters so that their functionality can be modified. Parameters are given as a comma separated list with their types and names defined.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Note: The `\(number1)` syntax is basic String Interpolation and is used to insert the integer into the String.

Functions with parameters are called by specifying the function by name and supplying an input value of the type used in the function declaration.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Any value of type Int could have been used.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

When a function uses multiple parameters the name of the first parameter is not required for the first but is on subsequent parameters.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//output: "15 Is the magic number"
```

Use external parameter names to make function calls more readable.

```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

Setting the default value in the function declaration allows you to call the function without giving any input values.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Section 18.3: Subscripts

Classes, structures, and enumerations can define subscripts, which are shortcuts for accessing the member elements of a collection, list, or sequence.

Example

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
}
```

Subscript Usage

```
var week = DaysOfWeek()
//you access an element of an array at index by array[index].
debugPrint(week[1])
debugPrint(week[0])
week[0] = "Sunday"
debugPrint(week[0])
```

Subscripts Options:

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variable parameters and variadic parameters, but cannot use in-out parameters or provide default parameter values.

Example:

```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

Usage

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Section 18.4: Methods

Instance methods are functions that belong to instances of a type in Swift (a class, struct, enumeration, or protocol). **Type methods** are called on a type itself.

Instance Methods

Instance methods are defined with a `func` declaration inside the definition of the type, or in an extension.

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

The `increment()` instance method is called on an instance of the `Counter` class:

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Type Methods

Type methods are defined with the `static func` keywords. (For classes, `class func` defines a type method that can be overridden by subclasses.)

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Section 18.5: Variadic Parameters

Sometimes, it's not possible to list the number of parameters a function could need. Consider a `sum` function:

```
func sum(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

This works fine for finding the sum of two numbers, but for finding the sum of three we'd have to write another function:

```
func sum(_ a: Int, _ b: Int, _ c: Int) -> Int {
    return a + b + c
}
```

and one with four parameters would need another one, and so on. Swift makes it possible to define a function with a variable number of parameters using a sequence of three periods: `...`. For example,

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

Notice how the `numbers` parameter, which is variadic, is coalesced into a single `Array` of type `[Int]`. This is true in general, variadic parameters of type `T...` are accessible as a `[T]`.

This function can now be called like so:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

A variadic parameter in Swift doesn't have to come at the end of the parameter list, but there can only be one in each function signature.

Sometimes, it's convenient to put a minimum size on the number of parameters. For example, it doesn't really make sense to take the `sum` of no values. An easy way to enforce this is by putting some non-variadic required parameters and then adding the variadic parameter after. To make sure that `sum` can only be called with at least two parameters, we can write

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // ok
sum(3, 4, 5, 6, 7) // ok
sum(1) // not ok
sum() // not ok
```

Section 18.6: Operators are Functions

Operators such as `+`, `-`, `??` are a kind of function named using symbols rather than letters. They are invoked differently from functions:

- Prefix: >x

- Infix: `x > y`
- Postfix: `xb>`

You can read more about basic operators and advanced operators in The Swift Programming Language.

Section 18.7: Passing and returning functions

The following function is returning another function as its result which can be later assigned to a variable and called:

```
func jediTrainer () -> ((String, Int) -> String) {
  func train(name: String, times: Int) -> (String) {
    return "\(name) has been trained in the Force \(times) times"
  }
  return train
}

let train = jediTrainer()
train("Obi Wan", 3)
```

Section 18.8: Function types

Every function has its own function type, made up of the parameter types and the return type of the function itself. For example the following function:

```
func sum(x: Int, y: Int) -> (result: Int) { return x + y }
```

has a function type of:

```
(Int, Int) -> (Int)
```

Function types can thus be used as parameters types or as return types for nesting functions.

Section 18.9: Inout Parameters

Functions can modify the parameters passed to them if they are marked with the `inout` keyword. When passing an `inout` parameter to a function, the caller must add a `&` to the variable being passed.

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
}

var apples = 30 // Prints "There's 30 apples"
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("There's now \(apples) apples") // Prints "There's 29 apples".
```

This allows reference semantics to be applied to types which would normally have value semantics.

Section 18.10: Throwing Errors

If you want a function to be able to throw errors, you need to add the `throws` keyword after the parentheses that hold the arguments:

```
func errorThrower()throws -> String {}
```

When you want to throw an error, use the `throw` keyword:

```
func errorThrower()throws -> String {
  if true {
    return "True"
  } else {
    // Throwing an error
    throw Error.error
  }
}
```

If you want to call a function that can throw an error, you need to use the `try` keyword in a `do` block:

```
do {
  try errorThrower()
}
```

For more on Swift errors: Errors

Section 18.11: Returning Values

Functions can return values by specifying the type after the list of parameters.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Functions can also return multiple values using tuples.

```
func maths(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = maths(5)
//resultTuple = (10, 15)
```

Section 18.12: Trailing Closure Syntax

When the last parameter of a function is a closure

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

the function can be invoked using the Trailing Closure Syntax

```
loadData("123") { result in
    print(result)
```

```
}
```

Section 18.13: Functions With Closures

Using functions that take in and execute closures can be extremely useful for sending a block of code to be executed elsewhere. We can start by allowing our function to take in an optional closure that will (in this case) return `Void`.

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

    if let block = block {
        block()
    }
}
```

Now that our function has been defined, let's call it and pass in some code:

```
closedFunc() { Void in
    print("Over already")
}
```

By using a **trailing closure** with our function call, we can pass in code (in this case, `print`) to be executed at some point within our `closedFunc()` function.

The log should print:

> Just beginning
>
> Over already

A more specific use case of this could include the execution of code between two classes:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // Notice how this is executed before the  closure
        print("1")
        // Make sure closure isn't nil
        self.closure = closure?()
    }
}
```

The log should print:

> 1
>
> 2