A Swift Introduction

Part Three

Functions and Closures

Functions

- Use `func` to declare a function
- Call a function by following its name with a list of arguments in parentheses
- Use `->` to separate the parameter names and types from the function’s return type
- Basic function syntax with no (`Void`) return value:

```
func greet(person: String, day: String) {
    print("Hello \(person), today is \(day).")
}

greet(person: "Bob", day: "Tuesday")
```

- Basic function syntax with `String` return value:

```
func greeting(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}

let greetingMessage = greeting(person: "Bob", day: "Tuesday")
print(greetingMessage)
```

Argument Labels and Parameter Names

- By default, functions use their **parameter names** as labels for their arguments

- Custom **argument labels** may be specified before the parameter name

- Or write _ to omit an argument label in a function call

```swift
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the value
    // for that parameter.
    print(parameterName)
}

someFunction(argumentLabel: 10)

// Another example
func greet(_ person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}

let message = greet("Ethan", from: "Kansas City")
```

Parameter Values

- Supply default parameter values by using = after the parameter type

```
let people = ["dave":23, "sarah":19, "jill":31]

func matchedNames(of people: [String:Int], aboveAge cutoffAge: Int = 18) -> [String]? {
    var matchedNames = [String]()
    for (name, age) in people where age > cutoffAge {
        matchedNames.append(name)
    }

    return (matchedNames.count > 0) ? matchedNames : nil
}

let names1 = matchedNames(of: people, aboveAge: 20)
let names2 = matchedNames(of: people)
```

- Use a variadic parameter to pass lists of values

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total = 0.0
    for number in numbers {
        total += number
    }

    return total / Double(numbers.count)
}

let mean1 = arithmeticMean(1, 2, 3, 4, 5)
let mean2 = arithmeticMean(3, 8.25, 18.75)
```

Returning Compound Values

- Use a tuple to return a compound value
- The elements of a tuple can be referred to either by name or by number

```swift
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}

let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
print(statistics.2)
```

Nested Functions

- Nested functions have access to variables that were declared in the enclosing function

```swift
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }

    add()
    return y
}

returnFifteen()
```

- Use nested functions to organize the code in a function that is long or complex

```swift
func step(location: Int, forward: Bool) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }

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

    return forward ? stepForward(input: location) : stepBackward(input: location)
}

var currentLocation = -4
while currentLocation != 0 {
    print("\(currentLocation)... ")
    currentLocation = step(location: currentLocation, forward: true)
}

print("zero!")
```

First Class Citizens

In programming language design, a **first-class citizen** is an entity which supports all the operations generally available to other entities.

These operations typically include:

- Being passed as an argument to a function
- Being returned from a function
- Being assigned to a constant or variable

First Class Citizens

- In Swift, the following are all considered to be first-class citizens:
    - **Functions**
    - **Structs**
    - **Enums**
    - **Protocols**
    - **Classes**

Functions as Return Values

- A function can return another function as its value

```swift
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }

    return addOne
}

var increment = makeIncrementer()
increment(7)
```

Functions as Parameters

- A function can take another function as one of its arguments

```
func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }

    return false
}

func lessThanTen(number: Int) -> Bool {
    return number < 10
}

var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Function Types

- Every function has one

- Is the combination of the **types of its parameters** and **its return type**

```swift
// Type: (String) -> Int?
func printAndCountCharacters(message: String) -> Int? {
    print(message)
    return message.count > 0 ? message.count : nil
}

// Type: (Int, Int) -> Int
func customAdd(number: Int, to anotherNumber: Int) -> Int {
    return number + anotherNumber
}
```

Void and ()

- `Void` is actually a typealias for `()`
- If a function does not have a return value, it technically returns `Void` although it is considered superfluous to write it in standard practice

```
// Type: (String) -> ()
// Alternatively: (String) -> Void
func greeting(name: String) -> Void {
    print("Hello \(name)")
}
```

- `()` is also used for functions with no parameters

```
// Type: () -> Int
func giveMeANumber() -> Int {
    return Int(arc4random())
}
```

- `()` is also used for functions with neither parameters nor return values

```
// Type: () -> ()
func doSomething() {
    print("I'm doing something!")
}
```

Functions as Constant/Variables

- Ex:

```
let printFunction: (String) -> Int? = printAndCountCharacters

let result = printFunction("Hello world")   // Notice there are no parameter labels in the call.
                                             // Parameter labels do not affect function types at all.
print("\(result) characters")
```

- Ex:

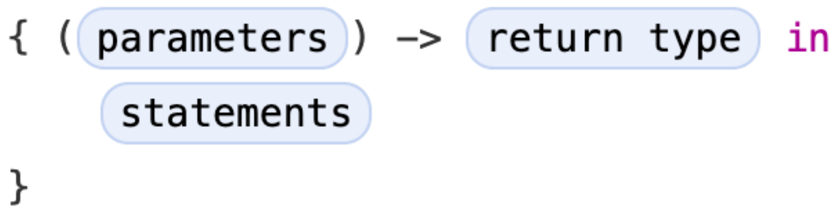
```
var mySavedFunction: (Int, Int) -> Int = customAdd
print(mySavedFunction(5, 12))

// Let's change the example we have stored
func customMultiply(number: Int, by anotherNumber: Int) -> Int {
    return number * anotherNumber
}

mySavedFunction = customMultiply
print(mySavedFunction(5, 12))

// This won't work since the function types don't match
mySavedFunction = giveMeANumber
```

Closures

- Self-contained blocks of functionality that can be passed around and used in code
- Functions are actually special types of closures
- Some closure types can capture and store references to any constants and variables from the context in which they are defined
- This is known as **closing over** those constants and variables (which is where the name **closure** comes from)

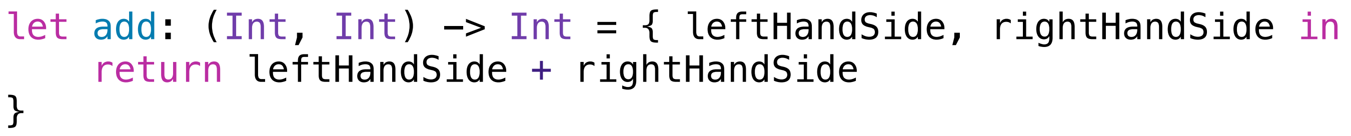
Closures

- Are **reference types**
  - Assigning a function or a closure to a constant or a variable actually sets that constant or variable to be a **reference** to the function or closure
  - If a function or closure is assigned to two or more different constants or variables; those constants or variables will refer to the **same closure**
- Expression syntax follows the general form:



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

- Ex:



```
let add: (Int, Int) -> Int = { leftHandSide, rightHandSide in
    return leftHandSide + rightHandSide
}
```

- The keyword `in` is used to separate the parameters and return type from the closure body

Shorthand Argument Names

- Swift automatically provides shorthand argument names to inline closures
- Can be used to refer to the values of the closure's arguments by the names `$0`, `$1`, `$2`, and so on
  - Using shorthand argument names within a closure expression allows:
  - The closure's argument list to be omitted from its definition
  - The number and type of the shorthand argument names to be inferred from the expected function type
  - The `in` keyword to also be omitted, because the closure expression is made up entirely of its body

```
let add: (Int, Int) -> Int = { $0 + $1 }
```

Closure Types

1. **Global functions** are closures that **have a name** and **do *not* capture** values from their enclosing environment.

    - Functions are not typically declared at global scope in apps.

2. **Nested functions** are closures that **have a name** and ***can* capture** values from their enclosing environment.

    - Methods on structs, enums, and classes are considered a type of nested function, as well as actual functions declared inside functions

3. **Unnamed closures *can*** capture values from their surrounding context.

Escaping Closures

- A closure is said to **escape a function** (`@escaping`) when the closure is passed as an argument to the function, but can be called **after the function returns**

```
class ServiceClient {
    func makeServiceCall(completion: @escaping (Int?) -> ()) {
        // Make network call asynchronously here
        // Then call completion handler when done with
        // the downloaded data (in this case the number 10)
        completion(10)
    }
}
```

Non-Escaping Closure

- A closure that’s called **within** the function it was passed into, i.e. *before* it returns

```
func step(location: Int, forward: Bool) -> Int {
    let stepForward: (Int) -> Int = { $0 + 1 }
    let stepBackward: (Int) -> Int = { $0 - 1 }
    let step = forward ? stepForward(location) : stepBackward(location)

    return step
}
```

Higher Order Functions

- A function that does at least one of the following:
  - Takes one or more functions as arguments
  - Returns a function as its result
- Collection types in Swift contain many higher order functions to aid programmers in determining how they want to work with a given sequence, including:
  - first
  - firstIndex
  - map
  - compactMap
  - flatMap
  - filter
  - reduce
  - forEach
  - sort
  - sorted

Higher Order Functions

```
var stringArray = ["One", "Two", "Three", "Four", "Five", "Six"]

//------------------------------------------------Examples------------------------------------------------------//

let firstExample = stringArray.first { $0.count == 5 } // "Three"

let firstIndexExample = stringArray.firstIndex { $0 == "One" } // 0

let mapExample = stringArray.map { $0.lowercased() } // ["one", "two", "three", "four", "five", "six"]

let optionalStringArray = ["One", "Two", "Three", nil, "Four", "Five", "Six", nil, "Seven"]
let compactMapExample = optionalStringArray.compactMap { $0 } // ["One", "Two", "Three", "Four", "Five", "Six", "Seven"]

let stringArrayArray = [["One", "Two"], ["Three", "Four"], ["Five", "Six"]]
let flatMapExample = stringArrayArray.flatMap { $0 } // ["One", "Two", "Three", "Four", "Five", "Six"]

let filterExample = stringArray.filter { $0.count > 3 } // ["Three", "Four", "Five"]

let reduceExample = stringArray.reduce("") { "\($0) \($1)" } // " One Two Three Four Five Six"

stringArray.forEach { print($0) }
/*
Prints:
    One
    Two
    Three
    Four
    Five
    Six
 */

stringArray.sort { $0 < $1 } // ["Five", "Four", "One", "Six", "Three", "Two"]

let sortedExample = stringArray.sorted { $0.count > $1.count }
```

Points to Remember

- Function return types come after the arrow -> or are empty if void
- By default, Swift uses **named parameters** for functions
- **Argument labels** and **parameter names** can be specified on a function
- Argument labels may be omitted
- Function parameters can have default values
- Variadic parameters accept zero or more values of a specified type
- Functions can be nested within functions

Points to Remember

- Every function has a type which is the combination of its parameter types and its return value type.

- Functions without return values actually return `Void` or `()`

- Functions with no parameters have the parameter type `()`

- Functions can be stored in variables or constants with the matching function type. (Under the covers, these are function pointers)

- The keyword `in` is used to separate the parameters and return type from the closure body

- Swift has several mechanisms for simplifying closure syntax as it helps readability

- Swift gives us shorthand syntax to refer to function/closure arguments by their position