9. Closures

Closures are self contained chunks of code that can be passed around and used in your code. Closures can capture and store references to any constants or variables from the context in which they are defined. This is know as closing over those variables, hance the name closures. Closures are use intensively in the Cocoa frameworks - which are used to develop iOS or Mac applications.

Functions are a special kind of closures. There are three kinds of closures:

- *global functions* - they have a name and cannot capture any values
- *nested functions* - they have a name and can capture values from their enclosing functions
- *closure expressions* - they don't have a name and can capture values from their context

The thing to keep in mind for the moment is that you already have an intuition about closures. They are almost the same as functions but don't necessarily have a name.

```
// a closure that has no parameters and return a String
var hello: () -> (String) = {
    return "Hello!"
}

hello() // Hello!

// a closure that take one Int and return an Int
var double: (Int) -> (Int) = { x in
    return 2 * x
}

double(2) // 4

// you can pass closures in your code, for example to other variables
var alsoDouble = double

alsoDouble(3) // 6
```

Remember the array `sortInPlace` method?

```
var numbers = [1, 4, 2, 5, 8, 3]

numbers.sortInPlace(<) // this will sort the array in ascending order
numbers.sortInPlace(>) // this will sort the array in descending order
```

The parameter from the `sortInPlace` method is actually a closure. The `<` and `>` operators are defined as functions, which can be referenced as closures. Here is an example for calling `sortInPlace` that uses a closure:

```
var numbers = [1, 4, 2, 5, 8, 3]

numbers.sortInPlace({ x, y in
    return x < y
})

print(numbers)
// [1, 2, 3, 4, 5, 8]
```

Declaring a closure

The general syntax for declaring closures is:

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

If the closure does not return any value you can omit the arrow (`->`) and the return type.

{ (`parameters`) **in**
`statements`
}

Closures can use variable and inout parameters but cannot assign default values to them. Also closure parameters cannot have external names.

Let's look at some examples:

```
var noParameterAndNoReturnValue: () -> () = {
    print("Hello!")
}

var noParameterAndReturnValue: () -> (Int) = {
    return 1000
}

var oneParameterAndReturnValue: (Int) -> (Int) = { x -> Int in
    return x % 10
}

var multipleParametersAndReturnValue: (String, String) -> (String) =
    { (first, second) -> String in
    return first + " " + second
}
```

The examples from above don't declare the type of each parameter, if you do so you don't need to state the type of the closure because it can be inferred.

```
var noParameterAndNoReturnValue = {
    print("Hello!")
}

var noParameterAndReturnValue = { () -> Int in
    return 1000
}

var oneParameterAndReturnValue = { (x: Int) -> Int in
    return x % 10
}

var multipleParametersAndReturnValue =
    { (first: String, second: String) -> String in
    return first + " " + second
}
```

Shorthand Parameter Names

Swift provides shorthand parameter names for closures. You can refer to the parameters as `$0`, `$1`, `$2` and so on. To use shorthand parameter names ignore the first part of the declaration.

```
numbers.sort({ return $0 < $1 })

var double: (Int) -> (Int) = {
    return $0 * 2
}

var sum: (Int, Int) -> (Int) = {
    return $1 + $2
}
```

Capturing Values

In the beginning of the chapter I mentioned that closures can capture values. Let's see what that means:

```
var number = 0

var addOne = {
    number += 1
}

var printNumber = {
    print(number)
}

printNumber() // 0
addOne() // number is 1
printNumber() // 1
addOne() // number is 2
addOne() // number is 3
addOne() // number is 4
printNumber() // 4
```

So a closure can remember the reference of a variable or constant from its context and use it when it's called. In the example above the `number` variable is in the global context so it would have been destroyed only when the program would stop executing. Let's look at another example, in which a closure captures a variable that is not in the global context:

```
func makeIterator(start: Int, step: Int) -> () -> Int {
    var i = start
    return {
        let currentValue = i
        i += step
        return currentValue
    }
}

var iterator = makeIterator(1, step: 1)

iterator() // 1
iterator() // 2
iterator() // 3

var anotherIterator = makeIterator(1, step: 3)

anotherIterator() // 1
anotherIterator() // 4
anotherIterator() // 7
anotherIterator() // 10
```

Trailing Closure Syntax

If the last parameter of a function is a closure, you can write it after the function call.

```
numbers.sort { $0 < $1 }

func sum(from: Int, to: Int, f: (Int) -> (Int)) -> Int {
    var sum = 0
    for i in from...to {
        sum += f(i)
    }
    return sum
}

sum(1, 10) {
    $0
} // the sum of the first 10 numbers

sum(1, 10) {
    $0 * $0
} // the sum of the first 10 squares
```

Closures are reference types

Closures are reference types. This means that when you assign a closure to more than one variable they will refer to the same closure. This is different from value type which make a copy when you assign them to another variable or constant.

```
// a closure that take one Int and return an Int
var double: (Int) -> (Int) = { x in
    return 2 * x
}

double(2) // 4

// you can pass closures in your code, for example to other variables
var alsoDouble = double

alsoDouble(3) // 6
```

Implicit Return Values

Closures that have only one statement will return the result of that statement. To do that simply omit the `return` keyword.

```
array.sort { $0 < $1 }
```

Higher order functions

A higher order function is a function that does at least one of the following:

- takes a function as input
- outputs a function

Swift has three important higher order functions implemented for arrays: map, filter and reduce.

Map

Map transforms an array using a function.

```
[ x1, x2, ... , xn].map(f) -> [f(x1), f(x2), ... , f(xn)]
```

Let's take as an example the problem of converting an array of numbers to an array of strings.

```
[1, 2, 3] -> ["1", "2", "3"]
```

One way of solving this problem would be to create an empty array of strings, iterate over the original array transforming each element and adding it to the new one.

```
var numbers = [1, 2, 3]

var strings: [String] = []

for number in numbers {
    strings.append("\(number)")
}
```

The other way of solving this problem is by using map:

```
var numbers = [1, 2, 3]

var strings = numbers.map { "\($0)" }
```

`{ "\($0)" }` is the closure we provided to solve this problem. It takes one parameter and converts it into a string using string interpolation.

The closure that we need to give to map take one parameter and will be called once for each of the elements from the array.

Filter

Filter selects the elements of an array which satisfy a certain condition.

For example let's remove all the odd numbers from an array:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8]

var evenNumbers = numbers.filter { $0 % 2 == 0 }
// evenNumbers = [2, 4, 6, 8]
```

Or remove all the even numbers:

```
var oddNumbers = numbers.filter { $0 % 2 == 1 }
// oddNumbers = [1, 3, 5, 7]
```

The closure that we need to give to map takes one parameter and will be called once for each of the elements from the array. It should return a `Bool` value, if it's `true` the element will be copied into the new array otherwise no.

Reduce

Reduce combines the value of an array into a single value.

For example we can reduce an array of numbers to their sum.

```
var numbers = [1, 2, 3, 4, 5]

var sum = numbers.reduce(0) { $0 + $1 } // 15
```

Reduce take two parameters, an initial value and a closure that will be used to combine the elements of the array. The closure provided to reduce takes two parameters, the first one is the partial result and the second one will be an element from the array. The closure will be called for each element once. In the sum example we started the sum from 0 and the closure added the partial sum with each element.

Here is another cool way in which we can use the fact that Swift operators are implemented as functions:

```
var numbers = [1, 2, 3, 4, 5]

var sum = numbers.reduce(0, +) // 15
```