PG5600 **iOS programmering**

Lesson # 2

Reminder

Everything is on github https://github.com/BeiningBogen/iOS-Kristiania

We have a Discord! See Canvas for invite link.

https://stackoverflow.com/help/how-to-ask

Review

- iOS development ecosystem
- Xcode and Playgrounds
- Swift
  - Strings
  - Loops
  - If & switch
  - Optionals
  - Numbers

Today - Swift (cont'd)

- Functions
- Closures
- Enums
- Classes and Structs
- Properties
- Methods
- Access control

Functions

```
// You should be familiar with the syntax

func functionName() {
    print("Hello")
}

functionName() // "Hello"
```

Functions with return values

```swift
func functionName() -> String {
    return "Hello World"
}

print(functionName()) // "Hello World"
```

Functions with optional values

```
func functionName() -> String? {
    return nil
}

print(functionName()) // nil
```

Functions with multiple return values (tuples)

```
func getError() -> (code: Int, message: String) {
    return (500, "Internal server error")
}

print(getError().message) // "Internal server error"
```

Functions with parameters

```
func greet(prefix: String, name: String) {
    print("Hello, \(prefix) \(name)!")
}

greet(prefix: "Mr", name: "Anderson") // "Hello, Mr Anderson!"
```

Functions with optional parameters

```
func greet(prefix: String?, name: String) {
    if let actualPrefix = prefix {
        print("Hello, \(actualPrefix) \(name)!")
    } else {
        print("Hello \(name)!")
    }
}

greet(prefix: "Mr", "Anderson") // "Hello, Mr Anderson!"
greet(prefix: nil, "Anderson") // "Hello, Anderson!"
```

Function with default parameters

```swift
// NB: Default parameters are the last parameters

func greet(name: String, prefix: String = "") {
    print("Hello, \(prefix) \(name)!")
}

greet(name: "Anderson")  // "Hello,  Anderson!"
greet(name: "Anderson", prefix: "Mr") // "Hello, Mr Anderson!"
```

External and internal parameter names

```
func greet(name n: String, prefix p: String) {
    print("Hello, \(p) \(n)!")
}

greet(name: "Anderson", prefix: "Mr") // "Hello, Mr Anderson!"
```

Omit parameter names

```
// NB: Usually not recommended because of readability.

func greet(_ name: String, _ prefix: String) {
    print("Hello, \(prefix) \(name)!")
}

greet("Anderson", "Mr") // "Hello, Mr Anderson!"
```

Playground Demo

Multiple value parameter (variadic parameter)

```
func greet(names: String...) {
    for name in names {    // names is of type [String]
        print("Hello \(name)")
    }
}

greet(names: "Agent Smith", "Mr. Anderson")

/*
"Hello Agent Smith"
"Hello Mr. Anderson"
*/
```

— Maximum one variadic parameter

— Always the last parameter (even after default parameters, if they exist)

Modifying value of input parameters (in-out parameters)

```swift
func swapInts(first: inout Int, second: inout Int) {
    let temp = first

    first = second
    second = temp
}

var a = 10
var b = 5

// NB: You have to call with `&` before the parameter
swapInts(first: &a, second: &b)
// a = 5
// b = 10
```

Functions that return a function

```
func createGreetingFunction(coolness: Int) -> (String) -> String {
    func normalGreeting(_ name: String) -> String {
        return "Hi, \(name)."
    }

    func veryCoolGreeting(_ name: String) -> String {
        return "Sup, \(name)."
    }

    return coolness > 8 ? veryCoolGreeting : normalGreeting
}

let fn = createGreetingFunction(coolness: -1)
print(fn("Anderson")) // Hi, Anderson.
```

Functions that accept a function as a parameter

```
func helloWorld() -> String {
    return "Hello world"
}

func invokeFunction(fn: () -> String, times: Int) {
    for _ in 0 ..< times {
        print(fn())
    }
}

invokeFunction(fn: helloWorld, times: 3)

/*
"Hello world"
"Hello world"
"Hello world"
*/
```

Closures

Aka blocks (obj-c), lamdas, anonymous features

Closures (cont'd)

```
/*
Syntax:
{ (parameters) -> returnType in
    expression
}
*/

let greetingClosure =  { (greeting : String) -> Void in
    print(greeting)
}

greetingClosure("Hello") // "Hello"
```

Closures (cont'd)

```swift
/*
Swift has an Array function called sorted

public func sorted(by: (Int, Int) -> Bool) -> [Int]
*/

var numbers = [43, 2, 1, 90]

numbers.sorted(by: { x, y in
    if y > x {
        return true
    } else {
        return false
    }
}) // 1, 2, 43, 90
```

Closures (cont'd)

```
// A shorthand looks like this
var numbers = [43, 2, 1, 90]

numbers.sorted(by: { x, y in y > x })  // 1, 2, 43, 90

// Single line expressions are implicitly returned
// Also note how the type omitted. It is inferred from context.
```

Closures (cont'd)

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

numbers.sort{ $0 < $1 }

/*
We can drop the parentheses and parameters if the closure is the last argument.
This is called a trailing closure. Parameters are accessible via $0..$n
*/
```

Enums

```
enum WashMode { // NB: Enum names start with a capital
    case unknown
    case cottom
    case wool
    case silk
}

var mode = WashMode.unknown

// Since the type is known we don't have to use it to access the value
mode = .cotton
```

Classes and structs

```
struct Coordinate {
    //...
}

class Person {
    //...
}
```

By default use structs

Structs

```
struct PointOfInterest {
    var latitude: Double = 0
    var longitude: Double = 0
    var name : String
}

let poi1 = PointOfInterest(latitude: 59.91126, longitude: 10.76046, name: "Kristiania")
print("\(poi1.name)  - \(poi1.latitude),\(poi1.longitude)")

var poi2 = poi1
poi2.name = "Høyskolen Kristiania"

// What is poi1.name?
// What is poi2.name?
```

Pass by value vs. reference

- Structs (including Strings, Arrays and Dictionaries), Int, and Enums are data types that pass by value, and are copied when passed around
- It's not as scary as it sounds, Swift is optimised so that copying only happens when it's absolutely necessary
- Classes, functions and closures are sent by reference and are not copied

Classes

```
class Server {
    var ip: String
    var startTime : Date?
    var running = false

    // Constructor
    init(ip: String) {
        self.ip = ip
    }
}

let server = Server(ip: "127.0.0.1")

// `self` is used to refer to the instance, same as `this` in some languages.
// Not to be confused with `Self`, which refers to the type of the instance.
```

Methods

```swift
class Server {
    // ...
    func boot() {
        startTime = Date()
    }
}

let server = Server(ip: "127.0.0.1")
server.boot()
```

Properties

- Stored properties (classes, structs)
- Computed properties (classes, structs, enums, and extensions)

Computed properties

```
class Server {3
    // ...
    // computed properties
    var uptime : Int {
        get {
            if let start = startTime {
                return Int(Date().timeIntervalSinceDate(start))
            } else {
                return 0
            }
        }
        /*
        Also possible to implement set. Otherwise read-only.
        set(newValue) {...}
        */
    }
}

let server = Server(ip: "127.0.0.1")
server.boot()
Thread.sleep(forTimeInterval: 5)
print("Up for \(server.uptime) seconds")
```

Property observers

```
class Server {
    var ip: String {
        willSet(newIp) {
            print("Will register a new IP: \(newIp)")
        }
        didSet {
            print("Ip \(ip) has been registered.")
            print("Old ip was \(oldValue)")
        }
    }
}
// If we don't include a parameter in our willSet and didSet implementations,
// the parameter names default to "newValue" and "oldValue" respectively.
```

Type properties/type methods - (aka static)

Operates at type level (class / struct), without the need for an instance.

Example of a class

```swift
class ClassUtils {
    class var typeProperty: Int {
        get {
            return 1
        }
    }
    class func typeMethod() {}
}

ClassUtils.typeProperty
ClassUtils.typeMethod()
```

Example of a struct

```
struct StructUtils {
    static var typeProperty: Int = 0
    static func typeMethod() {}
}

StructUtils.typeProperty
StructUtils.typeMethod()
```

Access control

- Swift defaults to sensible access control, so it is not always necessary to think about this
- Becomes very important when creating frameworks
- By default, the access is internal
- Pro Tip: set methods private by default, and change the access as needed

```
class SomeInternalClass {}                  // implicit internal
var someInternalConstant = 0                // implicit internal
```

Access levels

The short version

- `private` - restricted to the enclosing declaration
- `fileprivate` - restricted to the enclosing file
- `internal` - restricted to the module
- `public` - Accessible everywhere.
- `open` - Accessible, subclassable, overridable everywhere

Further reading

- pages 12-29 TSPL
- http://goshdarnclosuresyntax.com/

Tasks

See Exercises on GitHub