1st Day - iOS Swift Training in Barcelona

INSTRUCTOR: Geppy Parziale: geppy@invasivecode.com
Alex Martinez: amartinezmes@icloud.com
Tim Nowaczynski: TimNowaczynski@googlemail.com
Romina Felahi: r.felahi@gmail.com
Gabriele Filosofi: g.filosofi@me.com

For a Swift program the first line of executable code in “main.swift” is implicitly defined as the main entry-point. By adding @UIApplicationMain to a regular Swift file causes the compiler to synthesize a main entry point, and eliminates the need for a “main.swift” file. This happens for all template projects in Xcode.

At app launch time a main is provided, where the UIApplicationMain is called.

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}
```

In obj-C the code is

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

The NSAutoreleasePool class is used to support Cocoa’s reference-counted memory management system (ARC).
The main function will never return.
UIApplicationMain:

- creates an object of type UIApplication (the principal class name)
- creates an object of type AppDelegate (the delegate subclass of UIApplication)
- set up the event cycle

You don’t need to subclass UIApplication class, instead you designate another object, AppDelegate, which conforms to the UIApplicationDelegate protocol. AppDelegate will receive a number of application-delegate messages throughout the app life cycle, where you have the chance to interact with the app

When the app gets started:

applicationWillFinishLaunchingWithOptions()
applicationDidFinishLaunchingWithOptions() **..**this is the last delegate method called before to enter the ACTIVE state

The APP is ACTIVE

applicationDidBecomeActive()
The APP enter the Event Loop (main queue) **..**Implement your logic for handling iOS and user events

When the app is moved in background (by pressing the Home button):

applicationWillResignActive() **..**pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks.

The APP is INACTIVE

applicationDidEnterBackground() **..**release shared resources, save user data, invalidate timers

When the app is resumed in foreground:

applicationWillEnterForeground() **..**undo many of the changes made on entering the background

The APP is ACTIVE

applicationDidBecomeActive() **..**Restart any tasks that were paused (or not yet started) while the application was inactive

When the app is terminated because user or iOS quitted it:
applicationWillTerminate() **..**Save data if appropriate
If the app supports background mode execution, applicationDidEnterBackground is called instead of applicationWillTerminate.

An app has a UIWindow which is very stupid (black screen).
A UIWindow object does not have any visual appearance of their own.
Typically we add one or more UIView on top of the window. The first one is for free
Every view that appears onscreen is enclosed by a window. Events received by your app are initially routed to the appropriate window object, which in turn forwards those events to the appropriate view inside the window.

Objective-C

For example, in Objective-C you would do this:

*UITableView *myTableView = [[UITableView alloc] initWithFrame:CGRectZero style:UITableViewStyleGrouped];*

In Swift, you do this:

let myTableView: UITableView = UITableView(frame: CGRectZero, style: .grouped)

You don't need to call alloc; Swift correctly handles this for you. Notice that "init" doesn't appear anywhere when calling the Swift-style initializer.
You can be explicit in typing the object during initialization, or you can omit the type. Swift's type inference correctly determines the type of the object.

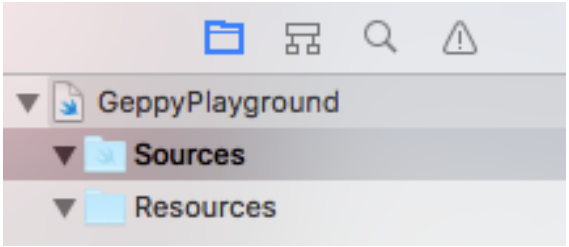
Swift 3.0.1

Swift language was introduced in 2014.
Online documentation is still a lot in obj-C.
Obj-C deals with memory management, pointers, .. Swift instead looks more like JS.
On Dec 2015 Swift became open source.
For ver 4.0 expected on June 2017 they promise to not do huge changes.

Tip: Do not use Obj-C for new projects.

Swift and C++ they do not work well together.
You cannot import C++ code directly into Swift.
Instead, create an Obj-C or C wrapper for C++ code.
Swift works good with C.
There is a version for iPad Pro of the playground.

Let's do some experiment in a playground.



In the Source directory you can drag class implementation files from Xcode and use within playground.
The folder Resources is for audio files, etc.

Constant and Variables

You can declare constants and variables with let and var

```
1  //Constants                                                    "Geppy"
2  let name: String = "Geppy"
3  //and variables
4  var age: Int = 40                                            40
5  age = age + 1                                                41
6  age += 1                                                     42
```

In Obj-C you can define the type later.
In a Swift playground file this is not allowed for safety reasons.
It is strict on the type. You have to specify it soon.
This allows the compiler to do a lot of optimisations.

Swift can infer the type by the value.

```
8  //Swift is able to infer the type                              "Gabriele"
9  let aname = "Gabriele"                                       40.5
10 var t = 40.5 //by default it is Double
```

Optionals

in Obj-C nil is a pointer to 0, or a is a pointer to a nonexistent object. In Swift it is not a pointer, it means “no value”.
In Swift you use *optionals* in situations where a value may be absent. An optional represents two possibilities: Either there *is* a value, and you can unwrap the optional to access that value, or there *isn’t* a value at all.

Tip: Every type can be optional

An optional String

```
19 //An optional can either be nil or some value                 nil
20 var bname: String?      //this is nil
```

You set an optional variable to a valueless state by assigning it the special value nil.
Before using an optional check if it has a value (unwrapping)

```
21 var bname: String? = "Jenny"                                  "Jenny"
22
23 //we can use the optional binding to unwrap
24 if var n = bname {
25     n.append("Y")                                             "JennyY"
26 } else {
27     print("error")
28 }
```

This technique is called *optional binding*.

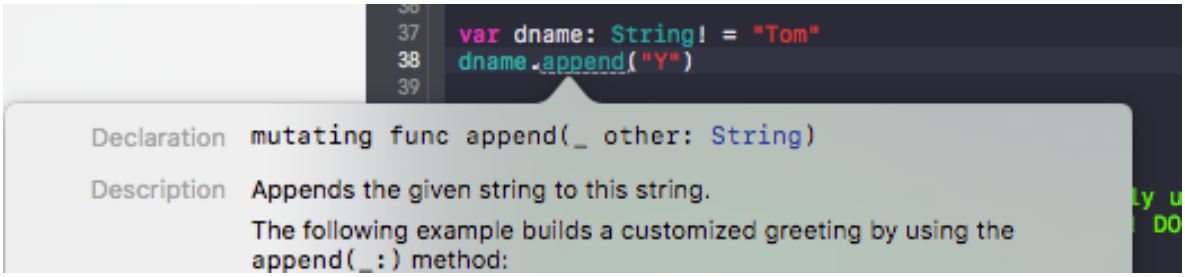
Once you’re sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional’s name. The exclamation mark effectively says, “I know that this optional

definitely has a value; please use it." This is known as *forced unwrapping* of the optional's value.

Implicitly unwrapped optionals are automatically unwrapped for you. But it is dangerous! You write an implicitly unwrapped optional by placing an exclamation mark (String!) rather than a question mark (String?) after the type that you want to make optional.

Use unwrapped optionals only if you are absolutely sure it has a value when you use it.

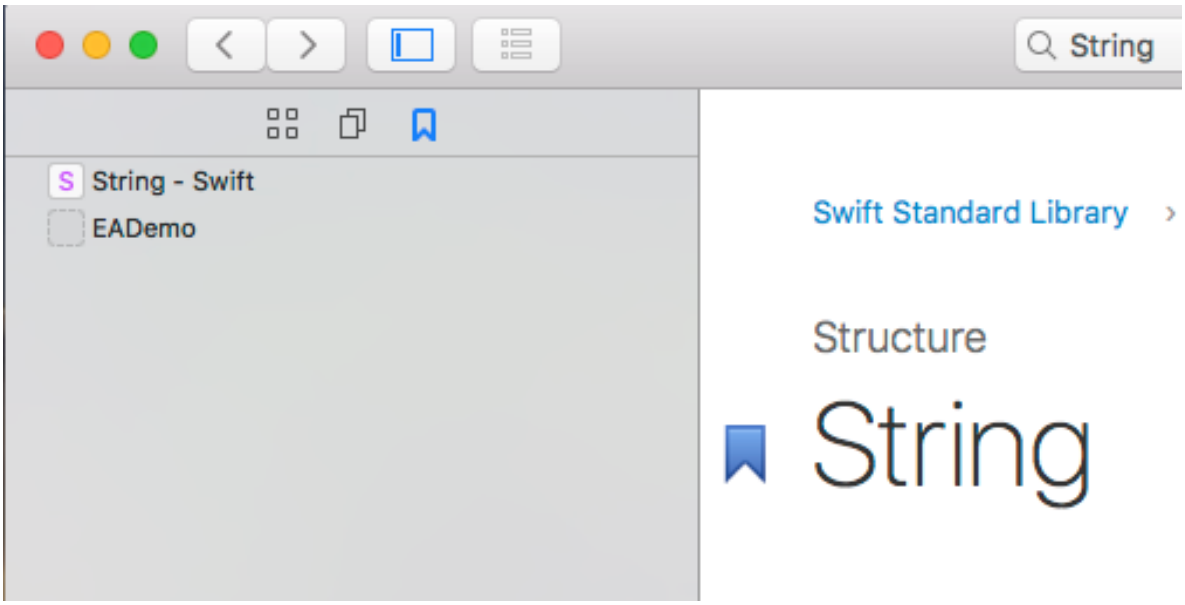
Tip: type Option key while click on a keyword to have Documentation



Swift Standard Library

The Swift Standard Library implements the basic data types, algorithms, and protocols you use to write apps in Swift. It includes high-performance fundamental data types such as String and Array, along with generic algorithms such as sort() and filter(_:).
The Swift Standard Library is written in C and the compiler translates it into C++
Note that the Swift Standard Library has nothing to do with iOS and UI.

Use bookmark!



String

A sequence of characters (or Unicode scalars) you can see on screen.
You can add special characters by placing your cursor inside the string and going to Edit > Emoji & Symbols

Array

An ordered, random-access collection. All elements have a single type

```
//ARRAY. Ordered collection of items. All of the same type
var ashoppingList: Array<String> = ["Milk","Pasta","Coffe"]
let bshoppingList: [String] = ["Milk", "Pasta", "Coffy"]
let cshoppingList = ["Milk","Pasta","Coffy"]

print(ashoppingList[1])
ashoppingList.count
ashoppingList.append("Tomatoes")
ashoppingList.insert("Soup", at: 2)
ashoppingList.remove(at: 3)
print(ashoppingList)
ashoppingList.isEmpty
```

```
["Milk", "Pasta", "Coffe"]
["Milk", "Pasta", "Coffy"]
["Milk", "Pasta", "Coffy"]

"Pasta\n"
3
["Milk", "Pasta", "Coffe", "Tomatoes"]
["Milk", "Pasta", "Soup", "Coffe", "Tomatoes"]
"Coffe"
"["Milk", "Pasta", "Soup", "Tomatoes"]\n"
false
```

There are a couple of ways to define an empty array

```
var aemptyArry = [Int]()
var bemptyArray: [Int] = []
```

```
[]
[]
```

Dictionary

A collection whose elements are key-value pairs.

```
//DICTIONARY. Collection of key:value objects. The order is undefined
var acar: Dictionary<String, String> = ["Color":"red", "MaxSpeed":"200", "Brand":"Ferrari"]
let bcar: Dictionary = ["Color":"red", "MaxSpeed":"200", "Brand":"Ferrari"]
//we can use the sugar notation:
let ccar: [String : String] = ["Color":"red", "MaxSpeed":"200", "Brand":"Ferrari"]
//we can exploit the type inference of the language:
let dcar = ["Color":"red", "MaxSpeed":"200", "Brand":"Ferrari"]

let speed = acar["MaxSpeed"]
acar.count
acar.isEmpty

acar["Model"] = "F40" //adds a new key and value
acar["Brand"] = "BMW" //replaces the value for the existing key
acar
```

```
["Brand": "Ferrari", "Color": "red", "MaxSpeed": "200"]
["Brand": "Ferrari", "Color": "red", "MaxSpeed": "200"]

["Brand": "Ferrari", "Color": "red", "MaxSpeed": "200"]

["Brand": "Ferrari", "Color": "red", "MaxSpeed": "200"]

"200"
3
false

"F40"
"BMW"
["Color": "red", "Model": "F40", "Brand": "BMW", "MaxSpeed": "200"]
```

Set

To do

Structure

```
struct Item {
	var price: Int
	var count: Int
}
```

Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and do not have to be of the same type as each other.
Example of a couple of type (Int, String)

Value and reference types

There are two kinds of types in Swift:

- Valued types: e.g. structures (90% of Swift Standard Library)
- Reference (or object) types: e.g. classes, functions (I do not pass a copy of the value, but a reference)

What's the difference ? When you pass a structure to a function as a parameter, that structure is duplicated in the function. When you pass a class instance to a function, only the reference is duplicated in the function.

Closures

Closure is a specific type of function. Sometimes it is called anonymous function
Except the mutating concept, everything can be applied

Class

You can subclass a class, not a structure.
For example, you don't want subclass a point, thus use a structure to represent it. For a person we would prefer a class, because we would like to create different categories of persons.

For Classes you NEED to create the init function! There is no one for free, as for structures.

Classes have properties and methods.
There are 3 kinds of properties:

- Stored properties
- Computed properties

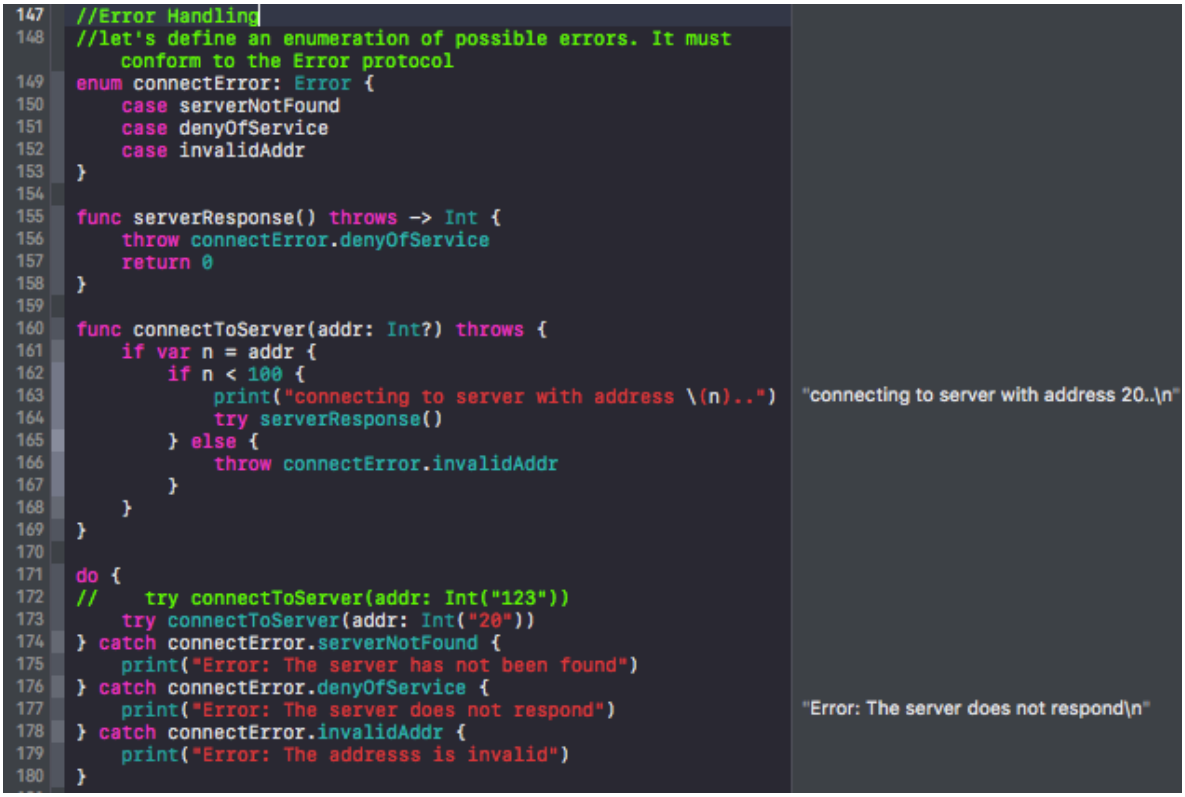
- property Observers

Range Operators

a...b from a to b including a and b
a..<b a to b without b

Error handling with Throws

You can use the *throws* keyword in the declaration of a function which can generate an error condition. Then you prepend *key* to the function call.
Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.
The standard way to handle errors is with the *do-try-catch* construction.
First of all, declare a enum with all handled errors. This enum must conform to the empty protocol Error. As you can see in the following example, errors can propagate through nested throwing functions.



Memory Management

It is easy for structures. Swift will handle the memory for valued types.
For classes and reference types it is different.
If you delete a reference you just have not access to the memory, but the object is still there, the memory is not freed and it becomes a memory leak.

MRR and ARC

In the past, with the Manual Retain Release (MRR), the developer would declare that an object must be kept in memory by claiming ownership on every object created, and relinquishing it when the object was not needed anymore.
MRR implemented this ownership scheme by a reference counting system.
When I create an instance of a reference type object, a retain counter is created. It is increased for every time I create a new reference to that object.
The counter indicates how many times it has been "owned", increasing the counter by one, and decreasing it by one each time it was released.
The object would then cease to exist in memory when its reference counter reached zero.

Before destroy a reference PLEASE decrease the retain counter!
When the counter is 0 Swift will destroy the RAM area.
Methods to be used are

objname.retain() (increases)
Use this right after the new reference has been created
objname.release() (decreases)
Use this right after the new reference has been destroyed

Having to take care of reference counting manually was really an annoyance for the developer. Now there is a compiler technology ARC (Automated Reference Counting) which reads the code and inserts the methods at compile time for you.
This is not a garbage collector. A garbage collector (only for the Mac) works at runtime.

There are two keywords to specify the type of reference when we create it:

- strong
- weak
- unowned

Under ARC a strong declaration will retain the object and increment its reference count.
Example:
let firstEmployee = Employee(age:21) (the strong keyword is omitted)

Under ARC a weak variable will not be retained by the object declaring it.
Example:
weak var secondEmployee = firstEmployee

"unowned" means that we don't claim ownership or memory management on the variable.

Retain cycles

A retain cycle happens when object A has a strong reference to B (A —> B), and B also has a strong reference to A (B —> A). This cycle will prevent the memory from releasing, when it should, thus becoming a memory leak.
The solution is to mark B —> A as weak.
ARC unfortunately doesn't include a circular reference detector, so it's prone to retain cycles, thus forcing the developers to take special precautions when writing code.

Unowned case Example:
Person -> Credit Card (use strong specifier)
Credit Card -> Person (this ref cannot be nil)
Use the unowned specifier in this case
So, When you destroy the Person also the Credit Card became nil.

Whenever you have a property that is a closure which refers to the class then there is a chance for a retain cycle.
You have to add [unowned self] in the definition of the closure.

lazy
The *lazy* keyword tells to initialise a property later than init.

Enumerations
It is more powerful than in other languages.

```
enum Direction {
	case north
	case south
	case west
	case est
}
```

In Swift you don't need to assign numeric values.

var direction = Direction.west (Direction is the type of the var)
or
direction = .north

Raw Values:

```
enum semaphore: String {
	case red = "red"
	case green = "green"
	case bleu = "blue"
}
```

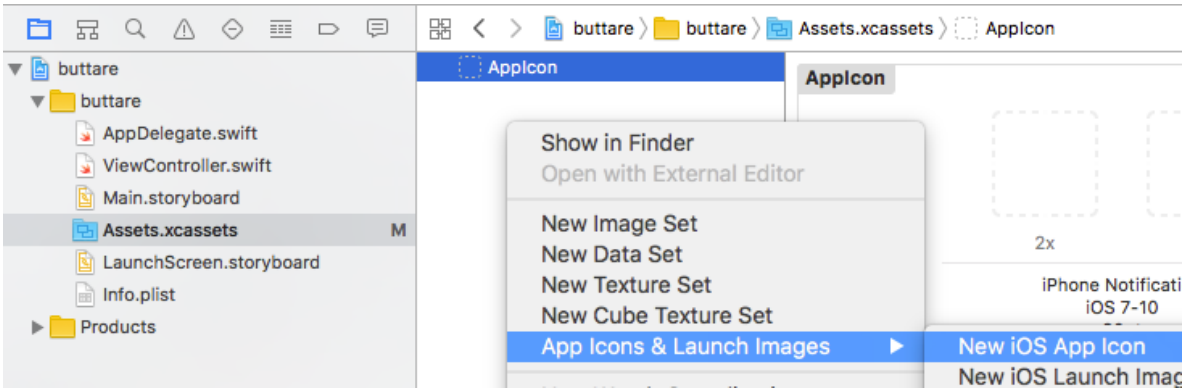
let red = Semaphore.red.rawValue

NOW LET'S MOVE TO iOS

Foundation is the basic framework. Not related to the UI, only to the model. The prefix of all class is NS. For some of them the prefix disappeared.
On top of it there is UIKit. The prefix is UI. This level is related to UI.
Cocoa Touch is the collection of all the frameworks for iOS.
AppKit is like UIKit but for Mac. It is back to 1996. UIKit is more recent.
CA QuarkCore is also important for any app.

Create App Icons

When you create an app in Xcode, you have to prepare a set of PNG images for the app icon. The procedure is very simple.
Create a 200x200 px image for your app, "Icon-200.png"
To do that you can the online tool
www.canva.com
Select the AppIcon under Assets.xcassets, or create it if not available by right clicking in the xcassets area

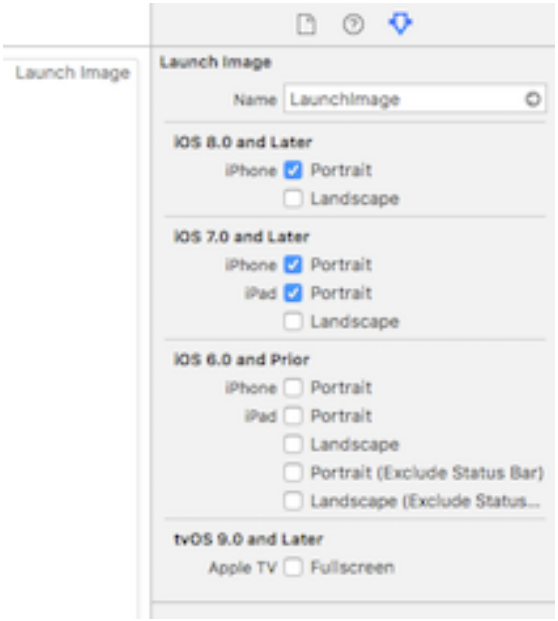


For each requested image "Icon-N[Mx].png":

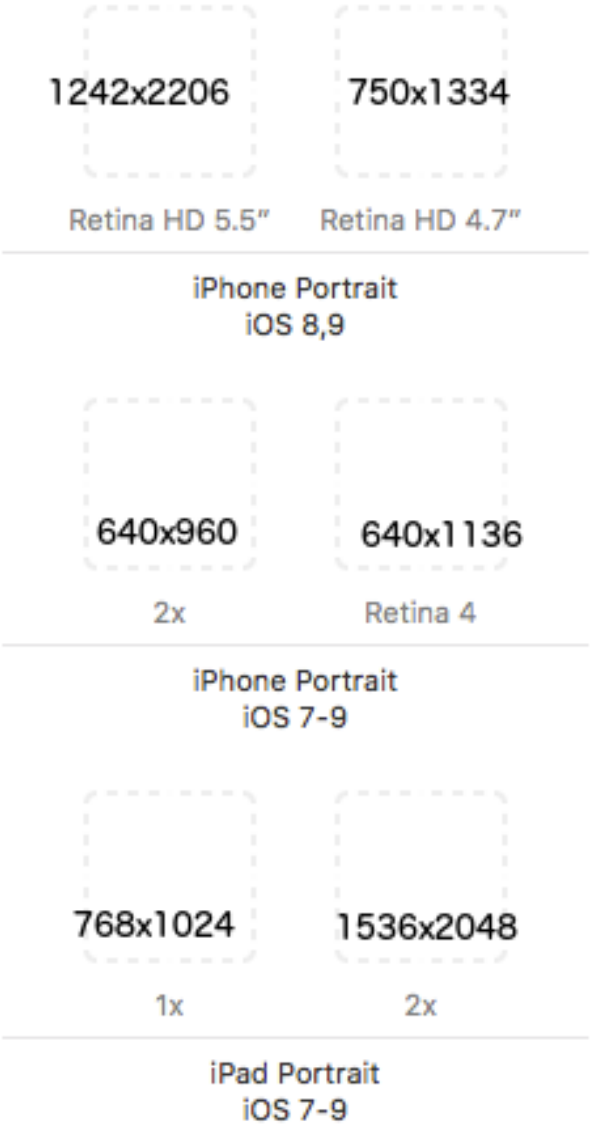
1. duplicate the "Icon-200.png" file
2. rename it as "Icon-N@Mx.png"
3. open it with GIMP (or any equivalent image editor)
4. resize the image to NMxNM px
5. save with the same format PNG
6. drag the image just created onto the appropriate placeholder

Create a Launch Image

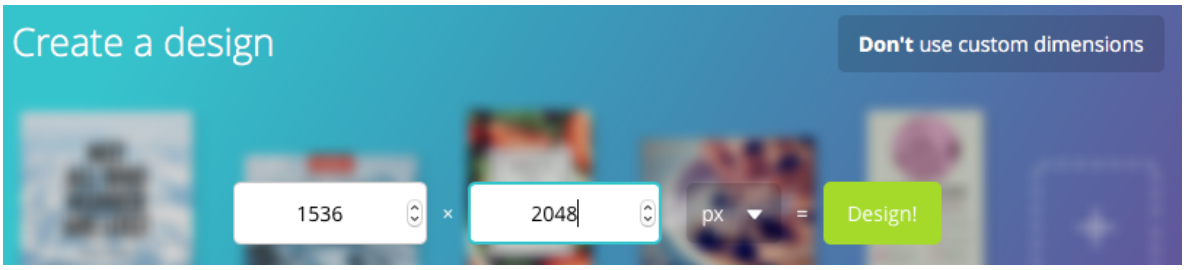
The launch image is visible for a while at app startup. As a launch image Apple recommends to use a blank version of your UI.
Select the LaunchImage under Assets.xcassets, or create it if not available by right clicking in the xcassets area.
In the attribute inspector uncheck unsupported iOS versions and devices.



Drag a copy of the launch image with the appropriate resolutions in each placeholder.



To do that you can the online tool
www.canva.com



View Controllers

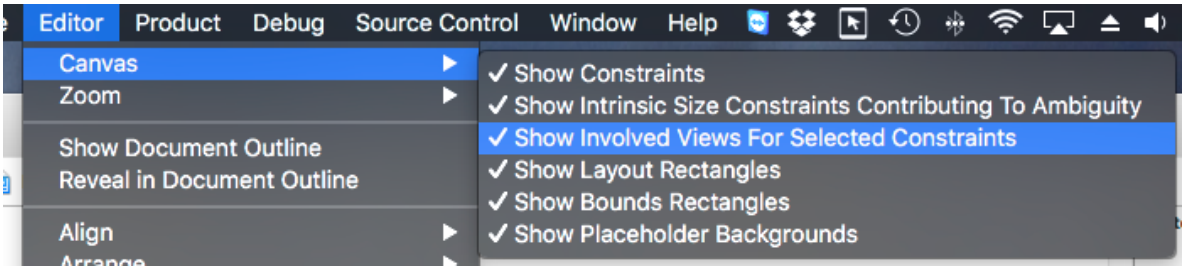
An App it is a collection of screens.
Each screen has to be mapped to a class that is a ViewController (VC), which contains the logic.
A VC is a subclass of UIViewController.
It has a strong reference to a UIView, on top of which there are objects to be controlled (buttons, sliders, etc.)

Examples:
UIViewController
UIImagePickerController which is subclass of UIImageNavigationController
...
In general they have also a predefined screen (not much customisable)

The Interface Builder allows you to draw the View.
The View comes from a nib file (Next Interface Builder), which is a binary file, compiled by IB from a xib file, which is a xml plist file.
The runtime gets the nib binary file from the disc and converts it to the view.

go to FirstApp..

Tags: it is a string assigned to a view
Select all Editor->Canvas options



Each VC controls one nib file.

A storyboard is a collection of nib files. It puts and connects all those nib files in a single file. The disadvantage is when you work with other people. Committing different contributions may result in a lot of conflicts. Then we prefer to use nib files instead.

The nib file will initialise all the class in the VC for you.

There are object properties that you can only change in source VC, not by Attribute inspector.
Typically the Interface Builder defines the static properties of the view objects.
The swift code is used for runtime behaviour.

When you launch a VC iOS will call viewDidLoad.
Just before the rendering iOS will call viewWillAppear
viewWillLayoutSubviews
viewDidlLayoutSubviews
viewWillDisappear
...
All those method need super. as prefix

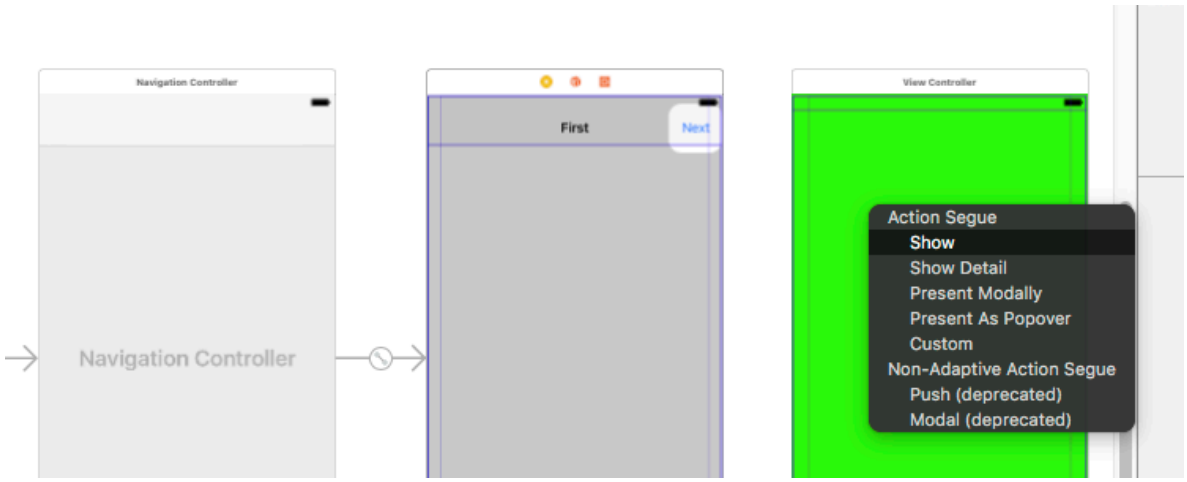
UINavigationController

Helps to navigate through a a sequence of VC.
It generates a navigation bar.
It needs to know which is the root VC (the view of the first VC).
It is like a stack.
When you press the back button the view is popped from the stack (DESTROYED).
On the bottom there is a Toolbar that is hidden by default.
You need to add a Bar Button Item to make transition to the next VC.

UITabBarController

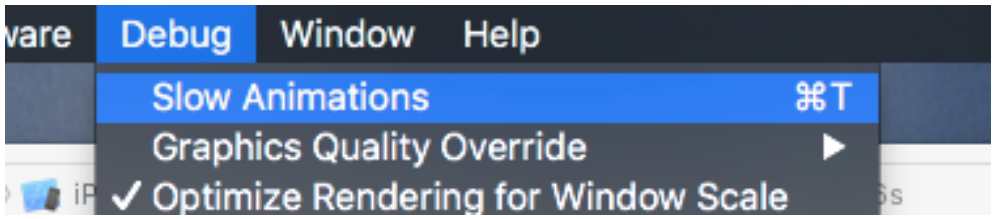
Use the TabBar with buttons to switch in any order between ViewControllers.
A tab has a small image and text.
It is like a container. You can embed a NavigationController inside a tab of a TabBar Controller. An example is the Music App.
Technically you can also do the reverse, but it is not effective.
If you have more than 5 tabs you can use the last one as a pointer to the remaining tabs (see iTunes for example)

go to Navigation app..



ADD A NAVIGATION ITEM OBJECT IN THE LAST VIEW

A trick to slow the animation



Let's add the third

They will be associated automatically

The third button must be added programmatically in the SecondViewController

In Obj-C you can call a method a selector

```
    //We add the third button to navigate to the third VC
    let nextButton = UIBarButtonItem(title: "Next", style: UIBarButtonItemStyle., target: Any? , 
        Selector? )
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recrea
}
```

```
UIBarButtonItemStyle bordered
UIBarButtonItemStyle done
M UIBarButtonItemStyle? init(rawValue: Int
UIBarButtonItemStyle plain
V Int hashValue
```

```
13  override func viewDidLoad() {
14      super.viewDidLoad()
15
16      //We add the third button to navigate to the third VC
17      let nextButton = UIBarButtonItem(title: "Next", style: .plain, target: self, action: #selector(goNext))
18
19      navigationItem.rightBarButtonItem = nextButton
20
21  }
22
23  func goNext() {
24      //here we are going to load the third VC
25      let vc = ThirdViewController()
26      show(vc, sender: nil)
27  }
```

```swift
class ThirdViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        title = "Third"
        view.backgroundColor = UIColor.red
    }

```

go to TabBar app..

Use lldb command prompt