

# INTRO TO SWIFT

Giorgio Natili @giorgionatili, Lead UI Engineer / Mobile Developer / Agile Coach

---

# AGENDA

---

- ▶ What's New in Xcode 6
- ▶ The Swift Language
- ▶ Key Concepts
- ▶ Generic, Extensions and Overload
- ▶ iOS Auto Layout
- ▶ User Interaction and Segues
- ▶ Remote and Local Data
- ▶ Core Data Basic
- ▶ UITableView and Custom Cells

---

# TAKEAWAY

---

**XCODE**  
usage

**SWIFT**  
fundamentals

**IOS**  
key concepts

**AUTO**  
layout

**FETCH**  
data

**DATA**  
persistence

**CLASS**  
extension

**APP**  
architecture

---

**HELLO!**

---

# **GIORGIO NATILI**



- Lead UI Engineer / Mobile Developer / Agile Coach
- Front-end Developer
- Mobile Full Stack (Cordova, iOS, Android)
- Technology Enthusiast
- Open Source Fellow
- Mentor @ [Airpair.com](http://Airpair.com)

---

**HELLO!**

---

# **MAX BLAUSHILD**

- Software Engineer
- Problem Solver
- Cutting Edge Techs Craftsman
- Mobile Developer



---

# MOBILE TEA

---



mobiletea@

<http://www.meetup.com/mobiletea>

---

## INTRO TO SWIFT

---

# XCODE 7.X

---

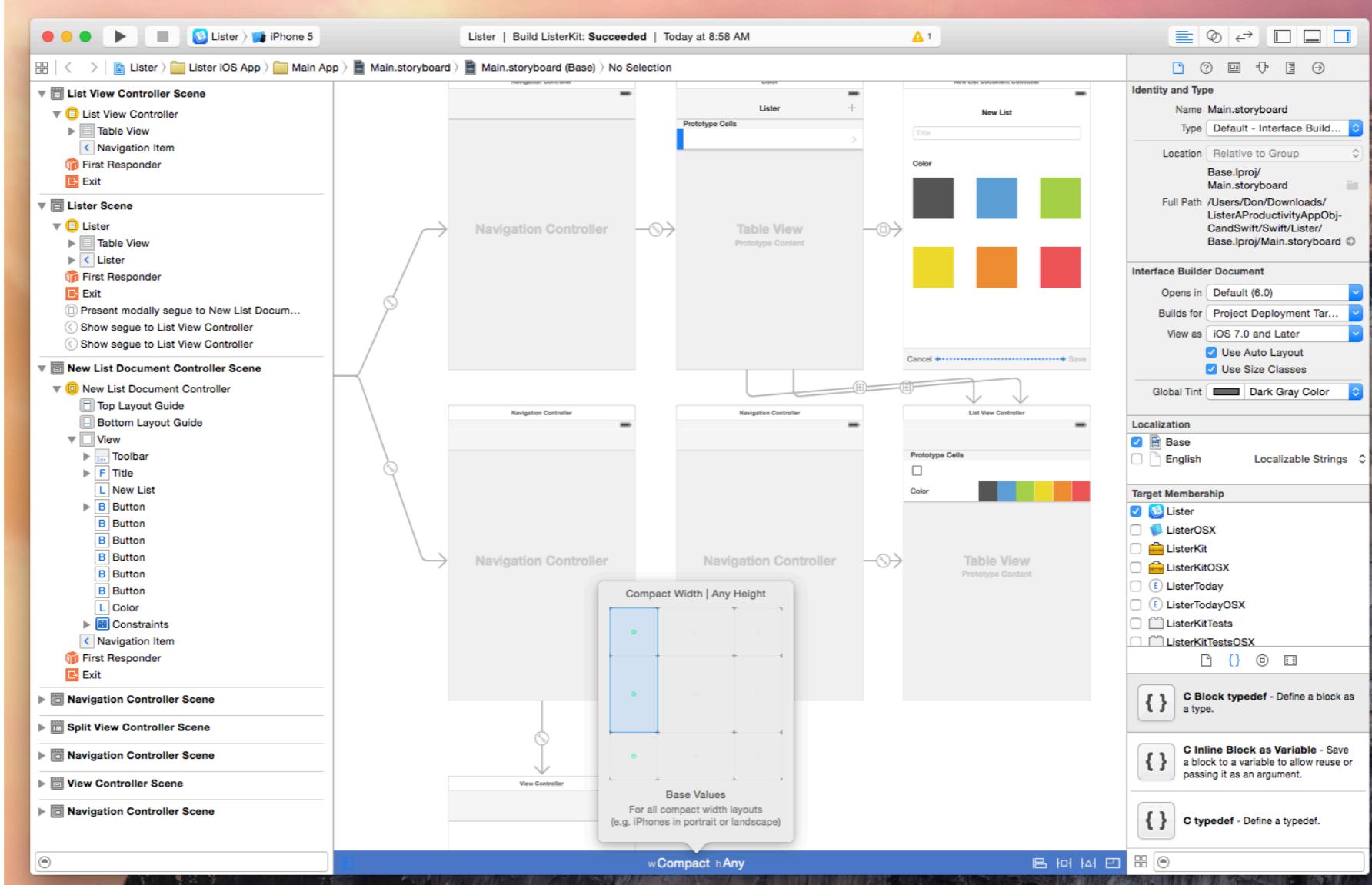
## WHAT'S NEW IN XCODE 6

---

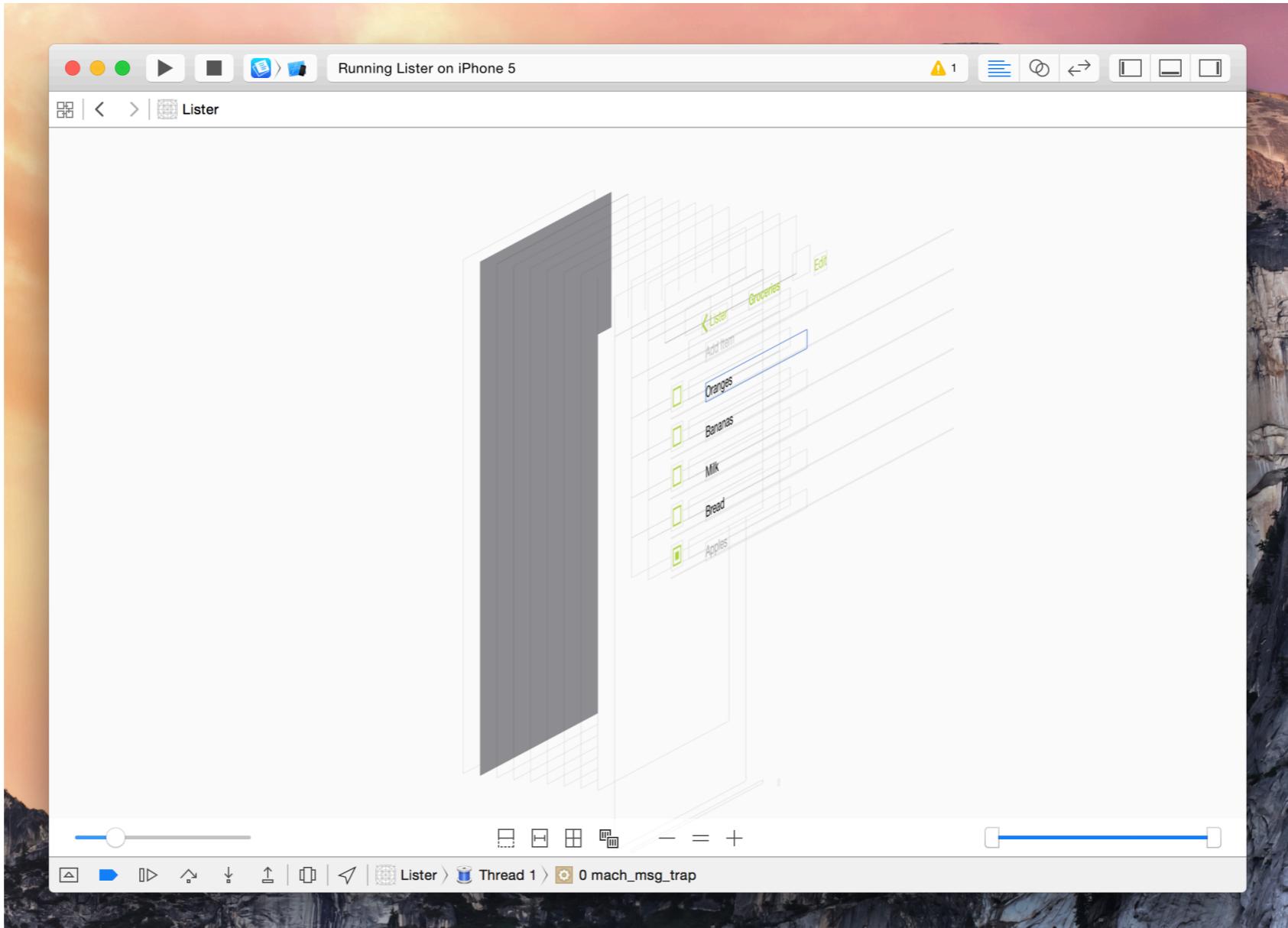
### FEATURES

- ▶ Live rendering
- ▶ Size Classes
- ▶ Visual view debugging
- ▶ Playground
- ▶ Read-eval-print loop CLI
- ▶ Swift
- ▶ Asynchronous code testing
- ▶ iOS simulator

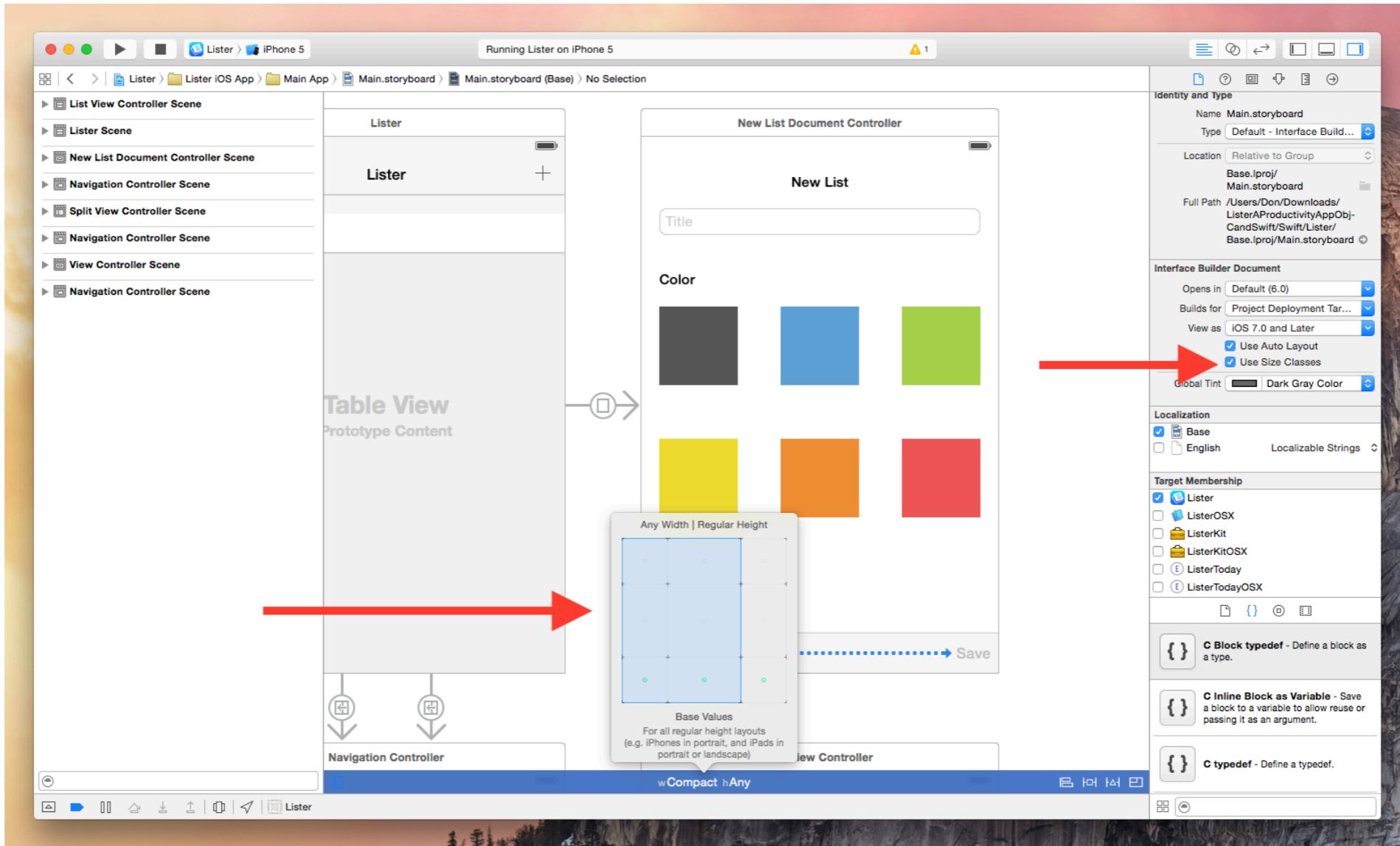
# VISUAL DESIGN



# LIVE RENDERING



# MULTI SCREEN DESIGN



# PLAYGROUND

The image shows a Xcode playground interface with three main sections: a code editor, a preview window, and a graph window.

**Code Editor:** Displays Swift code for a game scene named "Balloons". The code includes methods for initializing the scene, setting up physics, and handling collisions. It also contains a mathematical expression for calculating a sine wave.

```
func didMoveToView(scene : SKScene, delegate : SKPhysicsContactDelegate) {
    // Blimp Control
    yOffsetForTime = { i in
        return 80 * sin(i / 10.0)
    }
    // Scene Configuration
    balloonConfigurator = { b in
        b.physicsBody.categoryBitMask = CONTACT_CATEGORY
        b.physicsBody.fieldBitMask = WIND_FIELD_CATEGORY
        b.lightingBitMask = BALLOON_LIGHTING_CATEGORY
    }
    // Load images for balloon explosion.
    balloonPop = [1...4].map {
        SKTexture(imageNamed: "explode_0\($0)~")
    }
    // Install turbulent field forces.
    var turbulence = SKFieldNode.noiseFieldWithSmoothness(0.7,
        animationSpeed:0.8)
    turbulence.categoryBitMask = WIND_FIELD_CATEGORY
    turbulence.strength = 0.21
    scene.addChild(turbulence)
    cannonStrength = 210.0
    // Scene Initialization
    // Do the rest of the setup and start the scene.
    setupHero(scene, delegate)
    setupFan(scene, delegate)
    setupCannons(scene, delegate)
}

func handleContact(bodyA : SKSpriteNode, bodyB : SKSpriteNode) {
    if (bodyA == hero) {
        bodyB.normalTexture = nil
        bodyB.runAction(removeBalloonAction)
    } else if (bodyB == hero) {
        bodyA.normalTexture = nil
        bodyA.runAction(removeBalloonAction)
    }
}
```

**Preview Window:** Shows a 3D-style game scene titled "Balloons". The scene features a blue sky, a green ground with a yellow border, a large orange blimp, several colorful balloons (red, blue, green), a Ferris wheel, and two red cannons on the left.

**Graph Window:** Displays a graph of a sine wave with the equation  $y = 80 \cdot \sin(x)$ . The x-axis ranges from -30 to 30, and the y-axis ranges from -80 to 80. The graph oscillates between approximately -75 and 75.

---

## **IOS SIMULATOR**

---

- ▶ Improved SDK maintaining
- ▶ More devices
- ▶ Resizable device options

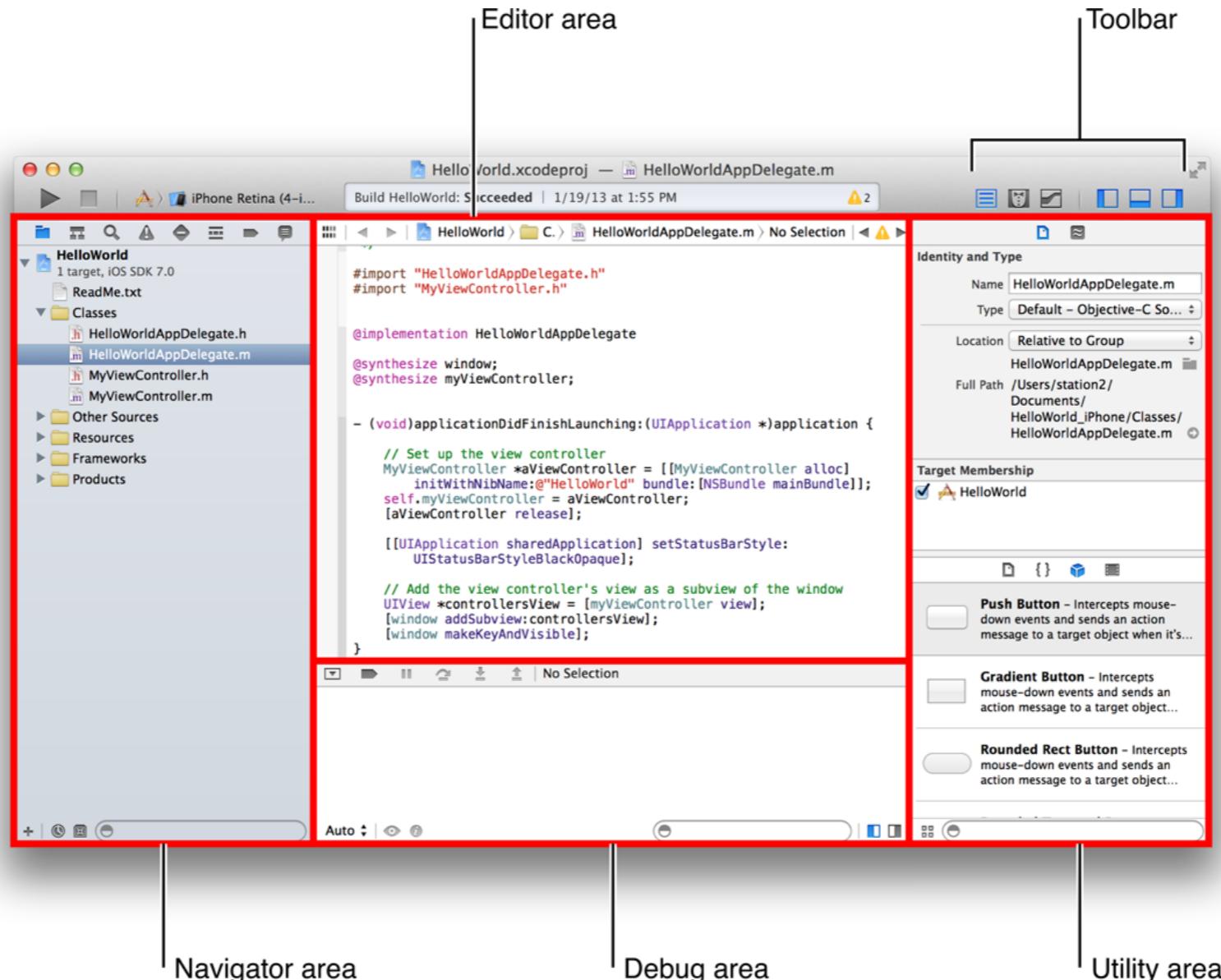
---

## ASYNCHRONOUS CODE TESTING

---

- ▶ XCTest API enhancements
- ▶ Network calls
- ▶ File IO

# XCODE AT A GLANCE



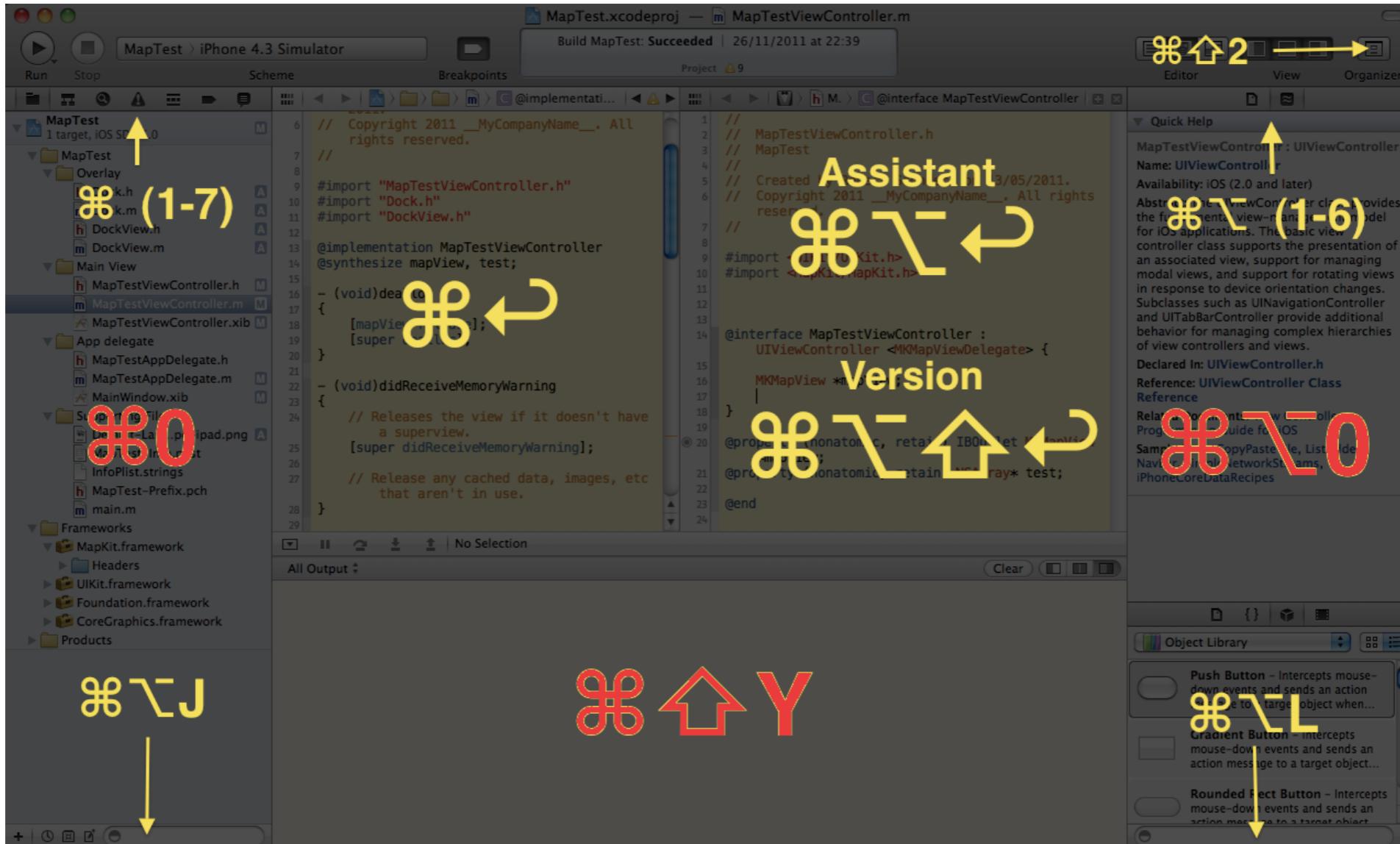
---

## SHORTCUTS – COMMON KEYS

---

- ▶ ⌘ = Command
- ▶ ⇧ = Shift
- ▶ ⇤ = Option/Alt
- ▶ ⌂ = Control
- ▶ ←→ = Left/Right Arrow Keys
- ▶ ↑↓ = Up/Down Arrow Keys

# WORKSPACE AND SHORTCUTS



# SHORTCUTS – BOOST PRODUCTIVITY

- ▶  $\hat{\wedge}$  + ⌘ + O      *Open a file*
  - ▶  $\wedge$  + ⌘ +  $\uparrow$  OR  $\downarrow$       *Switch between .m and .h files*
  - ▶ ⌘ + click on a variable/type      *Go to definition*
  - ▶  $\wedge$  + ⌘ +  $\leftarrow$  OR  $\rightarrow$       *Go forward / Go back*
  - ▶  $\wedge$  + ⌘ + E      *Edit all in scope*
  - ▶ ⌘ + T      *Open a new tab*
  - ▶ ⌘ + W      *Close the current tab*
  - ▶  $\hat{\wedge}$  + ⌘ + [ or ]      *Go forward / Go back in tabs*

# SHORTCUTS – BUILD, RUNNING AND TESTING

- ▶  + B Build
  - ▶  +  + K Clean
  - ▶  + R *Run the app (and eventually build)*
  - ▶  + U *Run the tests*

---

## SHORTCUTS – DEBUGGING

---

- ▶ F6 *Step over*
- ▶ F7 *Step into*
- ▶ ⌘ + \ *Add/remove breakpoint*
- ▶ ⌘ + Y *Disable/enable all breakpoints*
- ▶ ^ + ⌘ + Y *Pause/play the debugger*

---

## SHORTCUTS

---

A complete list of shortcuts is available here:

<http://bit.ly/xcode-short-cuts>

---

## INTRO TO SWIFT

---

# SWIFT BASICS

---

## **DATA TYPES – BASICS**

---

- ▶ Char
- ▶ Int
- ▶ Float
- ▶ Double

---

## NAMED AND COMPOUND TYPES

---

- ▶ A named type can have a name assigned

*Structures (e.g. Numbers and Strings), Classes, Enumeration, Protocols*

- ▶ A compound type has no name

*Tuples and functions (and closures)*

---

## NAMED TYPES

---

- A type identifier (i.e. name) can be defined by a custom Class, Structure, etc.
- Using the `typealias` keyword it's possible to specify a type identifier

```
typealias Point = (0, 0) // Actually this is a tuple
```

---

## COMPOUND TYPES

---

- ▶ A function definition is composed by its name and its signature representing the type definition

```
func doubleNum(val:Int) -> Int
```

- ▶ A tuple defines a type specifying the types of values it contains

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
```

---

## VARIABLES AND CONSTANTS

---

In Swift you can define a variable or a constant using different key words:

- ▶ var
- ▶ let

The general syntax to specify the type of the value stored in the var/const:

*var/let identifier: <Type>*

---

## TYPE INFERENCE

---

The variable or constant *types* are defined from the compiler

*var i:Int = 10*      *is the same of*      *var i = 10*

Type inference also happens in closures, the syntax is then very compact

```
let reversed = sorted(names) { s1, s2 in  
    return s1 > s2  
}
```

---

## ARITHMETIC EXPRESSIONS

---

- ▶ Most of the commons defined in other modern languages:

`+, -, /, =, %, +=, -=, *=, %=`

- ▶ Pre and post increment/decrement operators:

`++x, x++, --x, x--`

```
var x = 0
```

```
var y = x++ // Assign and then increment
```

```
var y = ++x // Increment and assign
```

---

## CUSTOM OPERATORS

---

- ▶ You can declare and implement your own custom operators in addition to the standard operators
- ▶ New operators are declared at a global level using the `operator` keyword, and are marked with the `prefix`, `infix` or `postfix` modifiers

```
prefix operator +++ {  
    // Do something  
}
```

---

## BOOLEAN EXPRESSIONS

---

- Relational operations as in other modern languages:

`==, <, >, <=, =>, !=`

- Ternary operator: ?

*[condition] ? [execute if true] : [execute if false]*

---

## CONDITIONAL STATEMENTS

---

```
if <condition>{
```

```
}
```

```
switch value {
```

```
    case something:
```

```
    case somethingElse:
```

```
    default:
```

```
}
```

---

## CONTROL TRANSFER STATEMENTS

---

- ▶ `return`: stops the execution and return a value
- ▶ `fallthrough`: force a C style behavior and also subsequent cases are evaluated
- ▶ `continue`: tells a loop to stop what it is doing and start again at the beginning of the next iteration
- ▶ `break`: ends execution of an entire control flow statement immediately

---

## SWITCH AND FALLTHROUGH

---

- Swift switch statements do not implicitly fall-through
- Cases can match against ranges

```
switch aNumber{  
case 0...5:  
    println("This number is between 0 and 5")  
    fallthrough  
case 6...10:  
    println("This number is between 6 and 10")  
    fallthrough  
case aNumber % 2:  
    println("it's an odd number")  
default:  
    println("I dunno in which range is this number")  
}
```

---

## LOOPS

---

Supports the most common loops: `for`, `while`, `do...while`, etc. but has a couple of nice hints for ranges and parameters:

```
// Using ranges
for i in 0...9{
    println("i = \$(i) ")
```

```
// Ignoring the variable iterator
for _ in 0..<10{
    println("I don't care about i")}
```

---

## COLLECTION TYPES – ARRAY VS DICTIONARY

---

- ▶ Arrays provide random access of a sequential set of data
- ▶ Dictionaries provide a map from a set of keys to a set of values
- ▶ Arrays are ordered, dictionaries are unordered

```
var dict: [String: Int] = ["giorgio": 74, "isaac": 82, "jonathan": 94]
for (key, value) in dict{
    println(key)
}
```

---

## COLLECTION TYPES – ARRAY

---

- ▶ There is no more distinction between NSArray and NSMutableArray in Swift
- ▶ An array can be instantiated using the literal syntax or the array type syntax

```
var languages: [String] = ["swift", "objc"]
```

is the same as

```
var languages = ["swift", "objc"]
```

---

## COLLECTION TYPES – ARRAY METHODS

---

- ▶ count
- ▶ isEmpty
- ▶ append
- ▶ insert(that, atIndex:)
- ▶ removeAtIndex(index)

---

## COLLECTION TYPES – DICTIONARY

---

A dictionary is a collection that stores key-value pairs, all keys have the same type and all values have the same type as well.

```
var states = ["NY": "New York", "MA": "Massachusetts"]
```

- ▶ count
- ▶ isEmpty
- ▶ updateValue(val, forKey:)
- ▶ removeValueForKey

---

## LAB 1

---

Create an Array with 5 numbers (e.g. [1, 2, 3, 7, 8]), and using a Range add the missing numbers to the array. As a bonus add the string "9" to the end of the array.



---

# FUNCTIONS

---

- ▶ `func` is followed by the identifier
- ▶ `identifier` is followed by a parameter list enclosed in parentheses
- ▶ `->` specifies the return type of the function, if there is one

```
func sayHello(name:String) -> String{  
    return "Hello " + name  
}
```

---

## FUNCTIONS – CONSTANT AND VARIABLE PARAMETERS

---

- By default function parameters are constant
- Optionally can be declared as variables var

```
func hello(var message:String, name:String) ->String{  
    message += name  
    return message  
}
```

---

## FUNCTIONS – NAMED ARGUMENTS

---

Sometimes giving a name to an external argument makes the code more readable when the function is called

```
func wishYou(name:String, holidayName holiday:String) {  
    println("Dear \\" + name + ", my best wishes for the next \\" + holiday + " holiday!")  
}  
wishYou("giorgio", holidayName: "xmas")
```

- ▶ Using the # sign the external argument name is the same as the internal name

```
func wishYou(name:String, #holiday:String) {  
    println("Dear \\" + name + ", my best wishes for the next \\" + holiday + " holiday!")  
}  
wishYou("giorgio", holiday: "easter")
```

---

## FUNCTIONS – SKIP NAMED ARGUMENTS

---

- By default class methods use the argument name as a named argument
- Sometimes it's useful to ignore the named argument adding an `_` in front of it

```
func hello(name: String, _ location: String) {  
    println("Hello \(name). I live in \(location) too.")  
}
```

---

## FUNCTIONS – VARIADIC PARAMETERS

---

- ▶ Functions can accept an indefinite number of parameters
- ▶ The *variadic* parameters are represented with 3 dots, every function can have **one** variadic parameter

```
func hello(people:String...) {  
    for p in people{  
        println("Hello \\" + p + "")  
    }  
}
```

---

## FUNCTIONS – INOUT PARAMETERS

---

- ▶ When a function is used to modify an existing value it's possible to pass it as an `inout` parameter
- ▶ The argument is then passed by *reference* and updated accordingly to the body of the function

```
var value
func updateValue(inout data:T) {
    // Do something
}
updateArray(&value)
```

---

## FUNCTIONS – OVERLOAD

---

- ▶ One of the most powerful features of Swift is the capability to *overload* functions
- ▶ Different implementations of the same function can live together, letting the compiler decide which one to use

```
func helloWorld() {  
    println("Hello world")  
}  
  
func helloWorld(from:String) {  
    println("Hello world by \"\$(from)\"")  
}
```

---

## FUNCTIONS – OPERATORS OVERLOAD

---

- Swift has the ability to overload operators such that existing operators, like `+`, can be made to work with additional types
- To overload an operator, simply define a new function for the operator symbol, taking the appropriate number of arguments

```
func * (left, right) -> String {  
    // Return something  
}
```

---

## LAB 2

---

Calculate the average number of a series of numbers stored in an Array using a function (it's assumed you will use the Playground) and print out the output.



---

## LAB 3

---

A common task when programming is to join strings. Handle the use case where the strings are passed as an Array or as a variable list of arguments.



---

# CLOSURES

---

Closures are expressions that resemble “anonymous” or unnamed functions: a closure is a self-contained block of functionality that can be passed around in your code.

```
{ (parameters) -> return type in  
    // Statements  
}
```

---

## CLOSURES – INFERRING FROM CONTEXT

---

When a closure is passed as an argument of another function that will use it with its other arguments, the arguments of the closure are inferred by the context (i.e. no need to declare the data type of the arguments)

```
let helloFromClosure = { (name) -> String in
    return "Hello " + name + " from your closure! "
}
```

---

## CLOSURES – SHORTHAND ARGUMENTS

---

In a closure you can refer to the arguments using the `$index` syntax (where `index` is an integer starting from 0) without specifying them at all

```
var hellos = ["giorgio", "jonathan", "isaac"].map({"Hello " + $0})
```

---

## CLOSURES + ARRAY

---

Ordering an Array with a closure makes the code more compact and easy to read

```
let animals = ["Dog", "Cat", "Fish", "Worm"]
let sortedStrings = animals.sorted(
    { (one:String, two:String) -> Bool in
        return one < two
    }
)
```

---

## LAB 4

---

Find the sum of all even numbers between 1 and 10 contained in an Array using a function or a closure.



---

# CLASSES

---

- ▶ The syntax is similar to the one of the most common languages
- ▶ A class can contains methods and stored or computed properties

```
class Person{  
    var name:String = "Giorgio"  
    var computedName:String {  
        get { return name + " " + rand().description}  
    }  
}
```

---

## CLASSES – COMPUTED PROPERTIES

---

- ▶ Do not actually store a value
- ▶ Provide a getter + an optional setter to retrieve and set other properties indirectly

```
class Teacher{  
    var name:String = "Giorgio", surname:String = "Natili"  
    var fullname:String {  
        get { return name + " " + surname}  
        set(value){  
            var data = split(value) {$0 == " "}  
            name = data[0]  
            surname = data[1]  
        }  
    }  
}
```

---

## CLASSES – INITIALIZERS

---

- ▶ Each class must have an initializer
- ▶ Strictly speaking the initializer is where the class member will be initialized (*not always true*)
- ▶ An initializer accepts parameters like a function
- ▶ A class can have multiple initializers that differ in their signature

---

## CLASSES – MULTIPLE INITIALIZERS

---

```
class Animal{  
    init() { // Some code }  
}  
  
class Animal{  
    init(specie:String) {  
        println("I am a \\" + specie + ")  
    }  
    init(specie:String, gender:String) {  
        println("I am a \\" + specie + ", my gender is \\" + gender + ")  
    }  
}
```

## CLASSES – MEMBERS SIGNATURE

---

- ▶ By default all the members have an internal access level
- ▶ Other optional access levels are public, private, final and static
- ▶ The access level of a custom type affects all members of that type

### THE INTERNAL LEVEL

This means internal to a product. A product is an app, framework, or any distributable compiled item.

---

## PROTOCOLS

---

- A *protocol* is a specification that lists the properties and methods an implementer has to support
- A property is specified by declaring a variable followed with a type and then either { get } or { get set }
- A method is specified using the func keyword followed by the function signature

```
protocol DailyGreeting{  
    var userName:String {get set}  
    func welcome() -> String  
}
```

---

## LAB 5

---

Imagine you work in the finance industry. Define a protocol to ensure that all the types of bank account are able to print out the full name of the owner, the balance, and the account ID. Once the protocol is ready, define an account class that implements the protocol.



---

## REFERENCE TYPES VS VALUE TYPES

---

A reference is usually associated with a pointer, meaning that the memory address where your variable “resides” is actually holding *another* memory address which points to the actual object in a different memory location.

class is a reference type, meaning that if you assign an instance of the class to another variable, it will hold only the reference to the instance, not a copy.

structs are value types, meaning that if you copy an instance of a structure to another variable, it is actually copied to the variable.

---

# STRUCTS

---

- A struct is a complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory (*wikipedia*)
- A struct is a value type, in general, it doesn't behave
- A typical value type has no implicit dependencies on the behavior of any external components
- Because a value type is copied every time it's assigned to a new variable, all of those copies are completely interchangeable

---

## STRUCS – MUTATING METHODS

---

A struct can have properties, methods, and initializers, and can be extended: a method that changes the value of a property is marked as mutating

```
struct Account {  
    var accountId:Int  
    var amount:UInt8 = 0  
    init(id:Int) {  
        accountId = id  
    }  
    mutating func manageAmount(value:UInt8) { amount += value}  
}
```

---

## STRUCS – IMPLEMENTING A STRINGS STACK

---

struct values can be accessed using the subscript syntax, like an Array:

```
struct StringsStack {  
    private var items = [String]()  
    mutating func push(item:String) {  
        items.append(item)  
    }  
    mutating func pop() -> String{  
        return items.removeLast()  
    }  
    subscript(i:Int) -> String{  
        return items[i]  
    }  
}
```

---

## ENUMERATIONS

---

An enum is a common type for encapsulating a group of related values which can be accessed in a type-safe way within the code:

```
enum AccountType{  
    case Savings  
    case Checking  
}
```

---

## ENUMERATIONS – WITH DATA TYPES

---

```
enum AccountType:String{
    case Savings = "savings"
    case Checking = "checking"
}

// Emun as a place holder for future values
enum AccountID{
    case inString (String)
    case inNumber (Int)
}
```

---

## ENUMERATIONS – MUTATING FUNCTIONS

---

An enumeration can define a *mutating function* that changes its state:

```
enum TwoStateSwitch {  
    case Off, On  
    mutating func next() {  
        switch self {  
            case Off:  
                self = On  
            case On:  
                self = Off  
        }  
    }  
}
```

---

## SWIFT OPTIONAL VALUES

---

- ▶ Using optional values it's possible to assign a `nil` value to a variable
- ▶ An optional is defined using the `?` sign

```
var age:Int? = nil
```

- ▶ Optional is an enumeration defined in the Swift language protocol

```
enum Optional : NilLiteralConvertible {  
    case None  
    case Some(T)  
}
```

---

## SWIFT OPTIONALS – UNWRAPPING

---

- ▶ An optional value needs to be unwrapped before using it: you can force the unwrap using !
- ▶ It's safer to unwrap an optional using a kind of bind technique

```
if let value = possibleValue {  
    // Something happens  
}
```

---

## FUNC OPTIONAL RETURN VALUES

---

- If it's possible that a function won't return a value, an optional can also be used in the function return type

```
func maxValue(nums: [Int]) -> Int? {  
    return nums.reduce(Int.min, { max($0, $1) })  
}
```

---

## TUPLES

---

Tuples enable you to create and pass around groupings of values

```
let http404Error = (404, "Not Found")
// http404Error.0 or http404Error.1
```

Tuples can also define named group of values

```
var namedHttp404Error: (statusCode:Int, statusText:String)
namedHttp404Error.statusCode = 404
namedHttp404Error.statusText = "Not Found"
```

---

## LAB 6

---

Implement a function able to return a tuple containing the max, min, and average value of a sequence of numbers. The returned tuple values should be accessed by name.



---

## EXTENSIONS

---

- Extensions add new functionality to an existing class, structure, or enumeration type
- This includes the ability to extend types for which you do not have access to the original source code (known as retroactive modeling)
- Explore the syntax <https://github.com/pNre/ExSwift>

---

## LAB 7

---

Create an extension for the `Int` class in order to determine if a number is odd or even.



---

## GENERICs

---

Generics allow a programmer to tell their functions and classes, "I am going to give you a type later, and I want you to enforce that type everywhere I specify"

```
// Function definition
func areValuesEqual<T: Equatable>(a: T, b: T) -> Bool {
    return a == b
}
```

---

# WE LOVE SWIFT!

---



<https://github.com/raywenderlich/swift-style-guide>

---

## INTRO TO SWIFT

---

# HELLO WORLD!

---

# HELLO WORLD APP

---

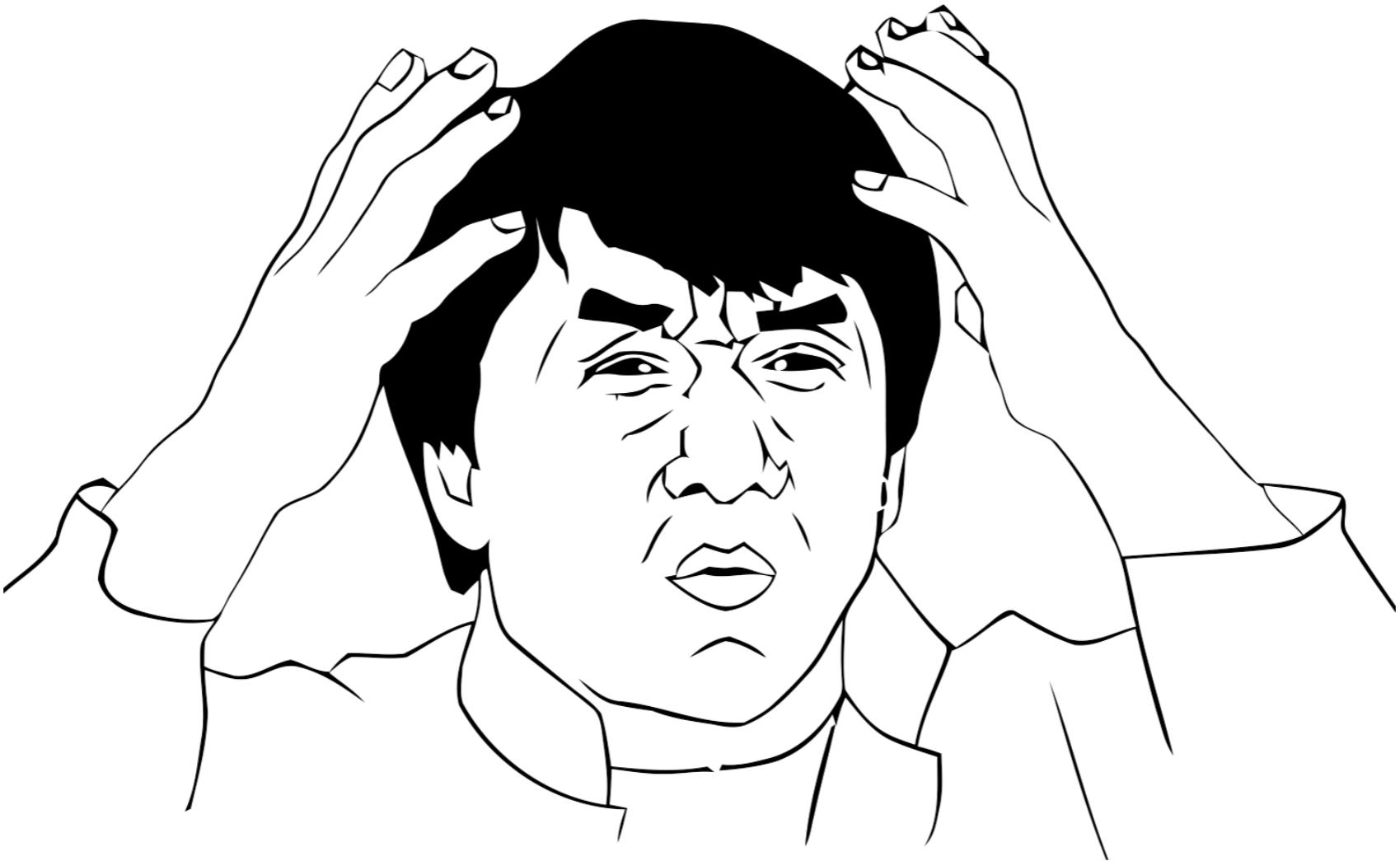
- ▶ Open Xcode
- ▶ Create a single view app
- ▶ Drag a button to the main view in the storyboard
- ▶ Right click on the button and create a binding for the touch up inside event
- ▶ Open an alert from the event handler

```
let alertController = UIAlertController(title: "Welcome to My First App", message:  
"Hello World", preferredStyle: UIAlertControllerStyle.Alert)  
  
alertController.addAction(UIAlertAction(title: "OK", style:  
UIAlertActionStyle.Default, handler: nil))  
  
self.presentViewController(alertController, animated: true, completion: nil)
```

---

# WHAT'S HAPPENED?

---



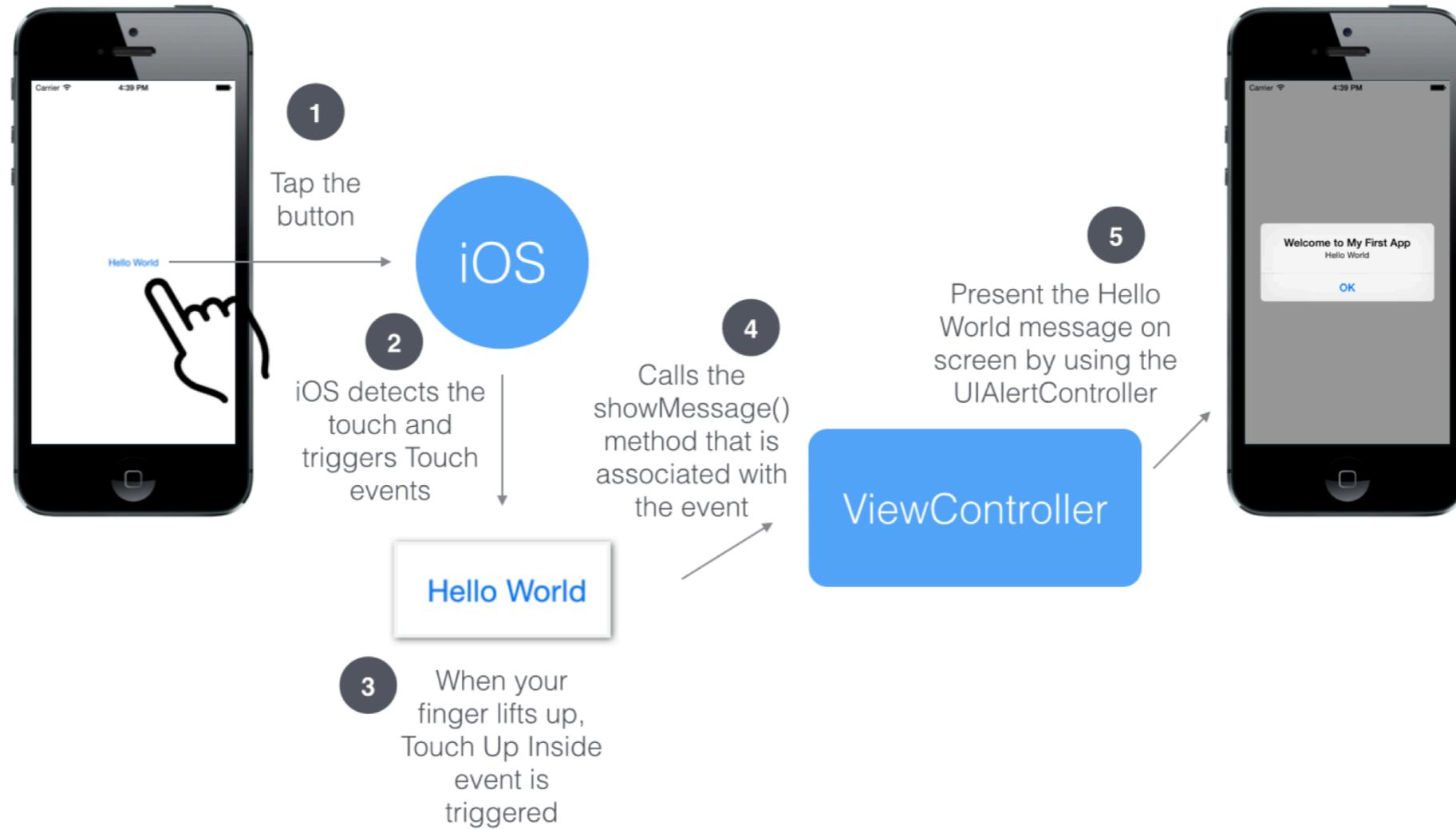
---

## BASIC CONCEPTS

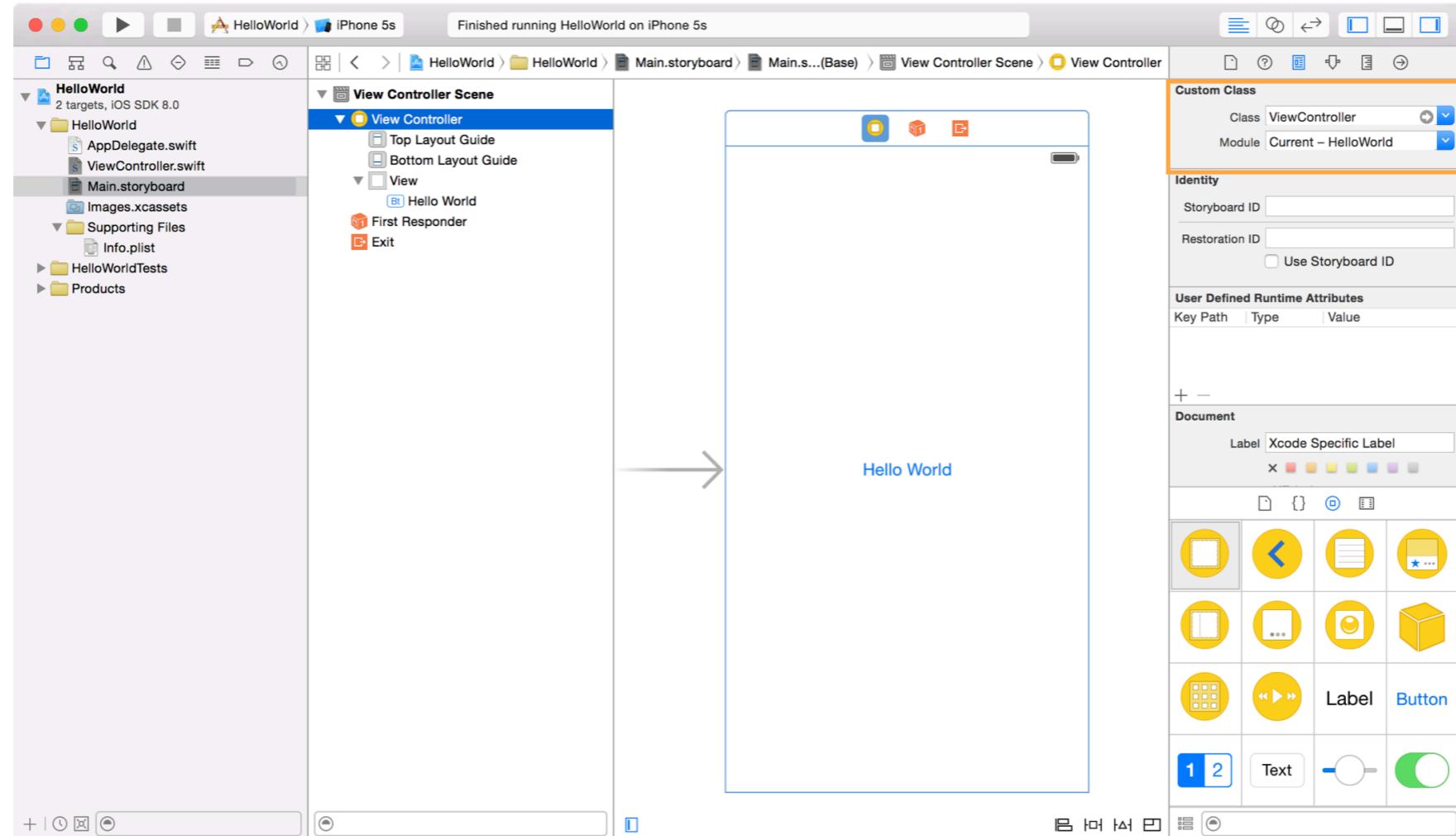
---

- ▶ The user interface in storyboard is the interface, while the code is the implementation
- ▶ The user interface elements (e.g. buttons) communicate with the code via messages
- ▶ The `@IBAction` keyword allows to connect your source code to user interface objects in Interface Builder

# UNDER THE HOOD



# UI AND VIEW CONTROLLERS



---

## OWNER, RESPONDER AND EXIT

---

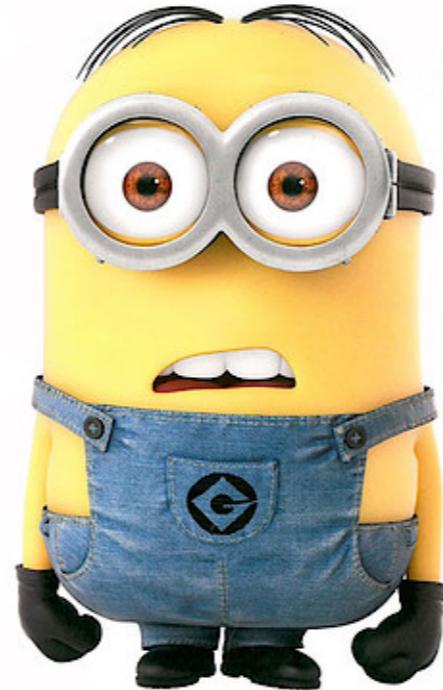
- ▶ The *File Owner* is an instantiated runtime object that owns the contents of your .xib and its outlets/actions when the .xib is loaded
- ▶ The *First Responder* is simply the first object in the responder chain that can respond to events (i.e. a proxy)
- ▶ The *Exit* is the method to be executed when .xib or the View Controller are unloaded from the view

---

## LAB 9

---

Create the Hello World application, and add another View Controller to the storyboard. The view controller should be rendered when a navigation bar button is touched, and should render something on the screen.



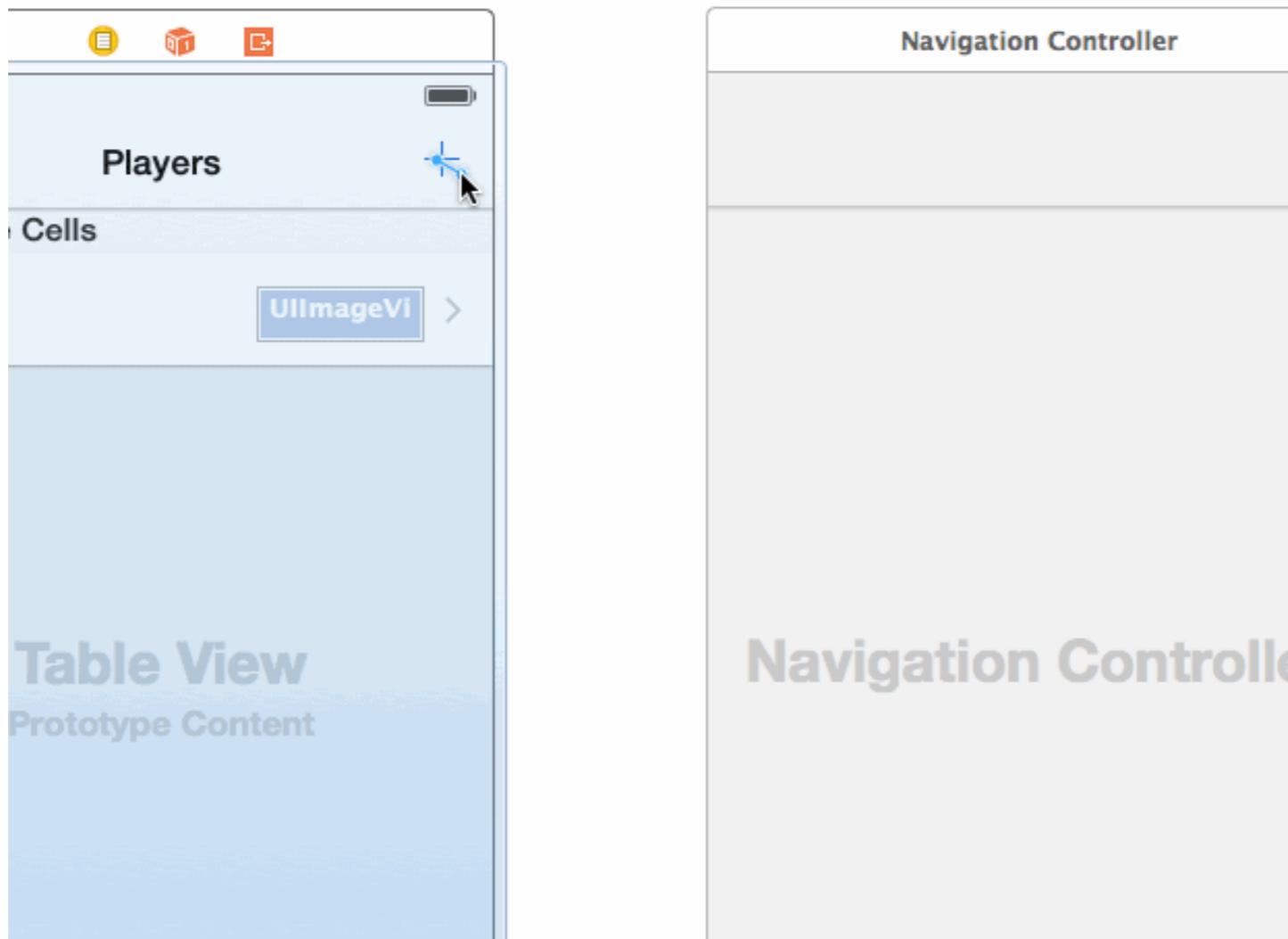
---

## MULTIPLE VIEWS CONTROLLER AND SEGUES

---

- ▶ An easy way to start to implement the navigation is to embed view controllers into navigation controllers (i.e. Editor > Embed In... > Navigation Controller)
- ▶ Then you can add “Segues” from a button using Control-drag
- ▶ A segue (pronounce: seg-way) is a type of connection from View Controllers and represents a transition from one screen to another

## ADDING A SEGUE – VISUALLY



---

## ADDING A SEGUE – PROGRAMMATICALLY

---

- ▶ The View Controller has to be instantiated accordingly to its identifier
- ▶ Using the navigationController, it has to be pushed into the view stack

```
self.storyboard?.instantiateViewController(withIdentifier: "sceene2")
self.navigationController?.pushViewController(view, animated: true)
```

- ▶ To come back to the previous segue, just pop the root View Controller

```
self.navigationController?.popToRootViewControllerAnimated(true)
```

---

## LAB 10

---

Reopen the Hello World app, add one or more View Controllers, and implement a visual and a programmatic navigation between them.



---

## INTRO TO SWIFT

---

# AUTO LAYOUT

---

## DIFFERENT PLATFORMS BUT SAME LAYOUT ISSUES

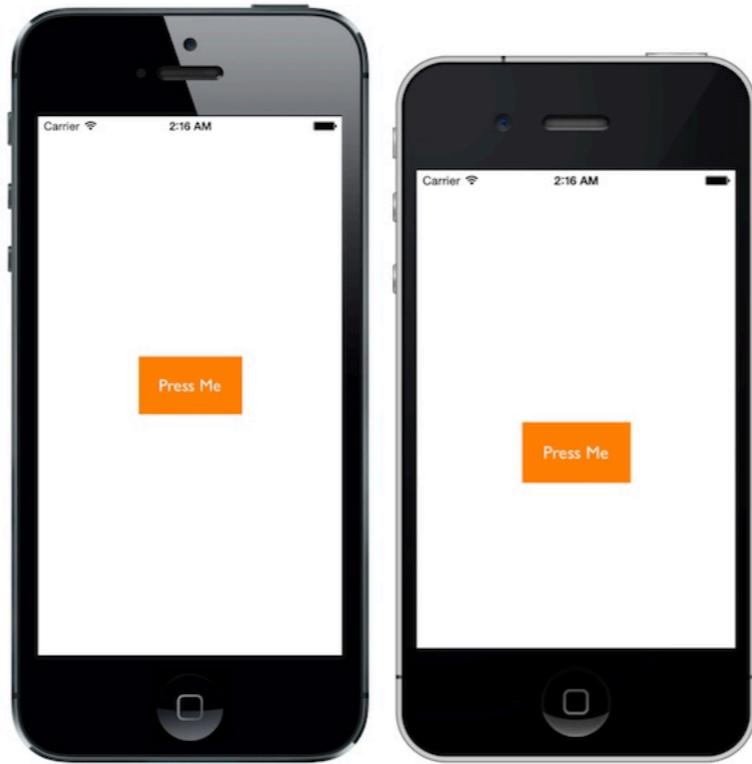
---



---

# WHAT'S WRONG IN THIS IMAGE?

---



---

## UI DESIGN, RETINA AND 3X

---

- In the storyboard or in a .xib file the UI elements are positioned using a point based coordinate system
- Until retina displays, a point was the equivalent of a pixel
- With retina displays, resolution is doubled and a point corresponds to two pixels
- iPhone 6 Plus introduced another complication, a point now corresponds to three pixels (and is then slightly downsampled when rendered...)

---

## AUTO LAYOUT

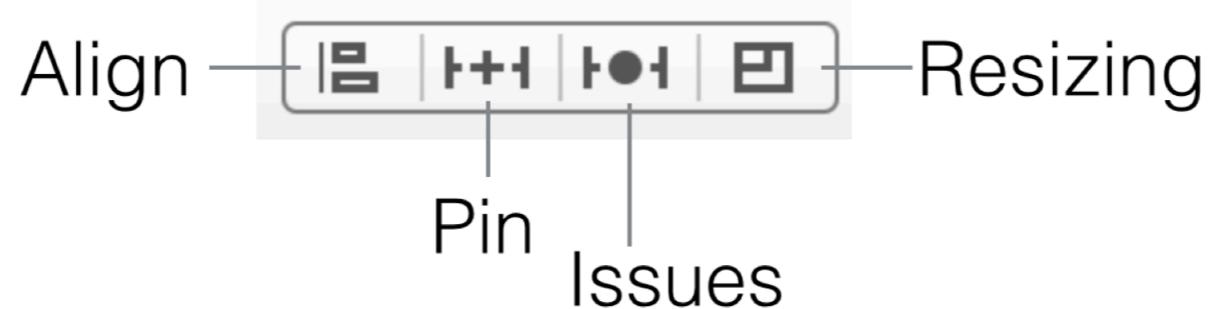
---

- ▶ Auto Layout is a constraint-based layout system
- ▶ It allows developers to create an adaptive UI that responds appropriately to changes in screen size and device orientation
- ▶ Xcode 6.1 allows a live preview in the storyboard itself

## DEFINING CONSTRAINTS

---

- ▶ Align: create alignment constraints, such as aligning the left edges of two views
- ▶ Pin: create spacing constraints, such as defining the width of a UI control
- ▶ Issues: resolve layout issues
- ▶ Resizing: specify how resizing affects constraints



---

## LAB 11

---

Again, open the Hello World app and use Auto Layout make the UI elements render properly on various Simulators. In addition show an alert when opening the second view controller.



---

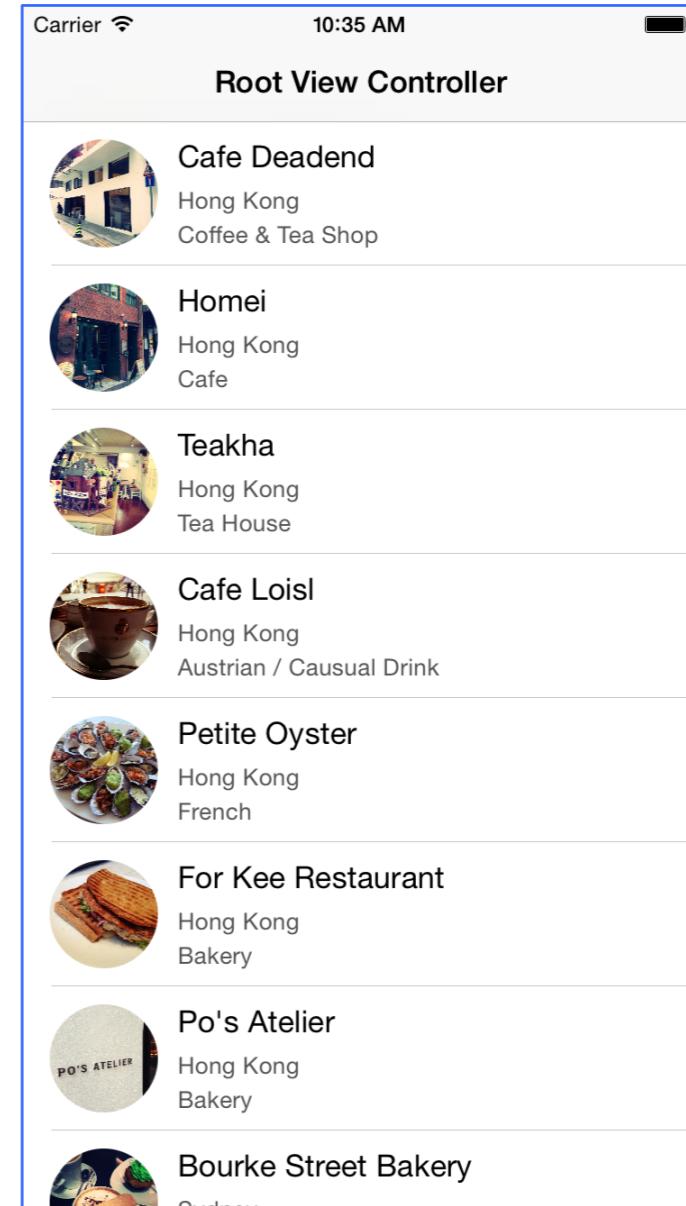
## INTRO TO SWIFT

---

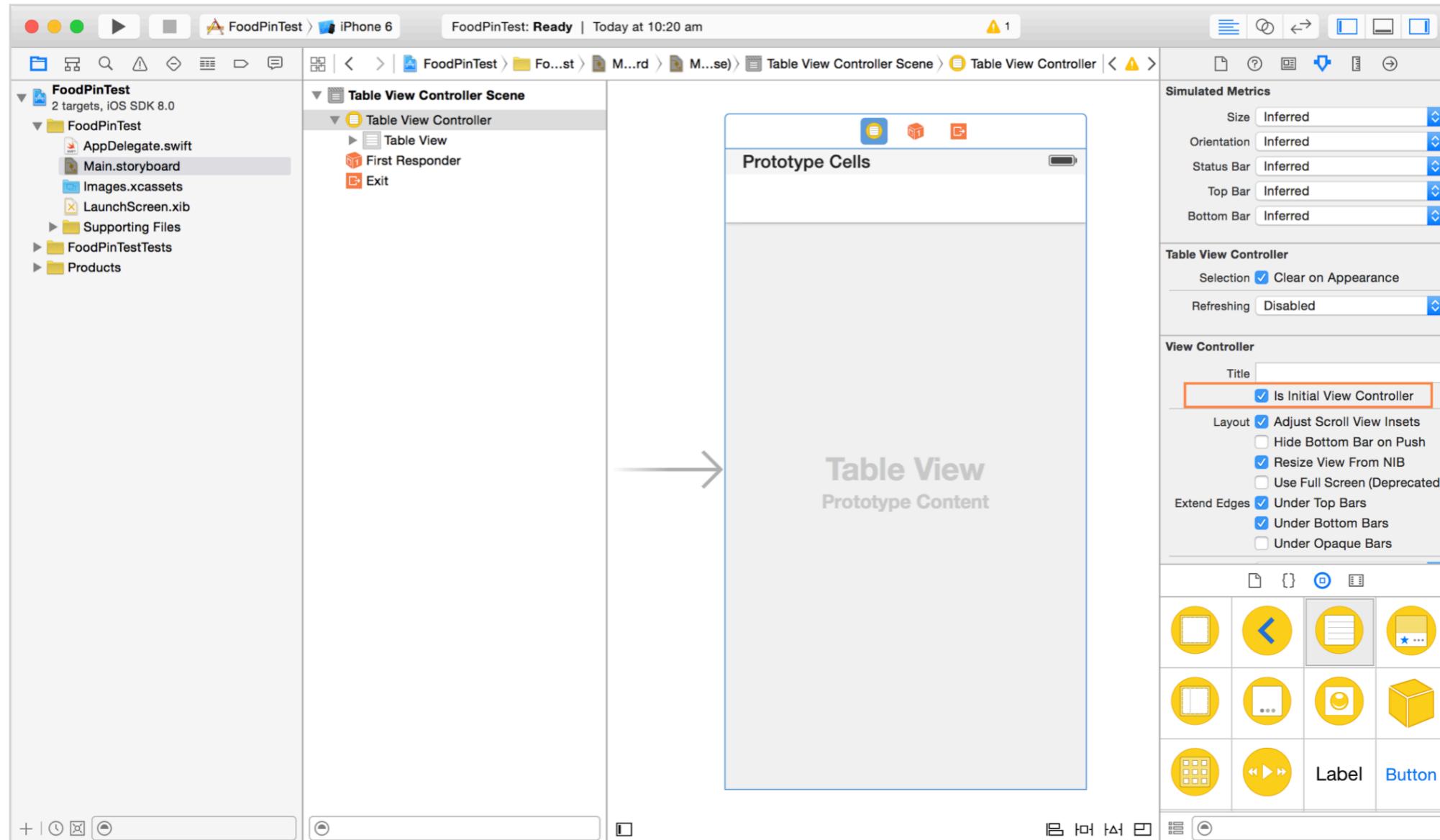
# TABLE VIEW

# TABLE VIEW APP

- ▶ Start from a Single View App
- ▶ Need a Table View Controller
- ▶ Uses Custom Cell to render
- ▶ Need some data
- ▶ Etc.



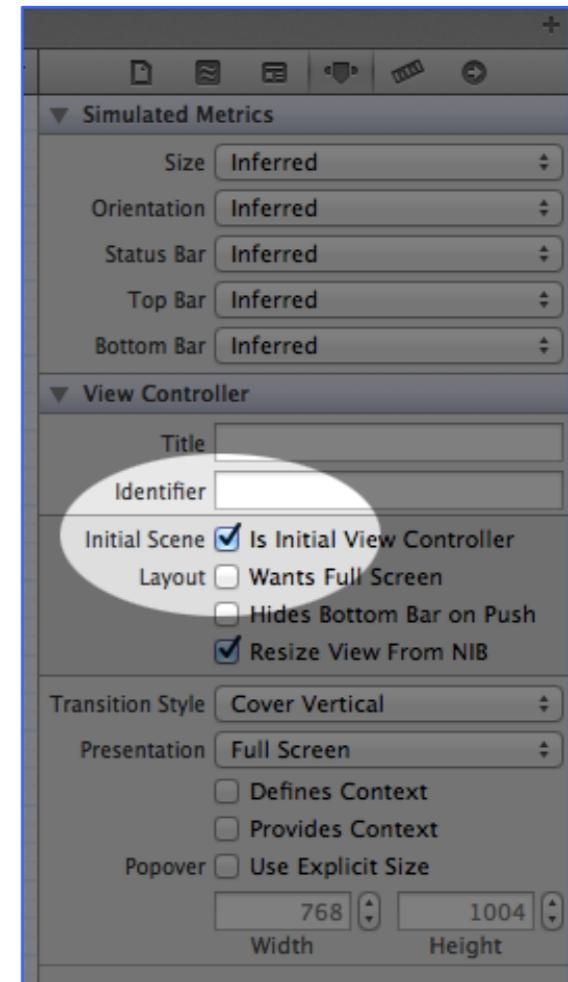
# TABLE VIEW STORYBOARD



# STEP 1 – PREPARE THE TABLE VIEW CONTROLLER

---

- ▶ Delete the default View Controller from the storyboard
- ▶ Add a new Table View Controller
- ▶ Set the new View Controller as the initial one



---

## **STEP 2 – PREPARE A CUSTOM CELL**

---

- ▶ Assign an identifier to the cell
- ▶ Assign a custom class to the cell (extending the `UITableViewCell`)
- ▶ Add the needed controls (`UIImage`, `Label`, etc.)
- ▶ Create the needed outlet

---

## STEP 3 – RENDER DATA

---

- ▶ Override the `tableView` function of the Table View Controller
- ▶ Get the cell using the identifier and populate it

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {

    let cellIdentifier = "Cell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath) as CustomTableViewCell

    // Configure the cell...
    cell.nameLabel.text = restaurantNames[indexPath.row]

    return cell
}
```

---

## TABLE VIEW APPS

---



---

## INTRO TO SWIFT

---

# FETCH + PERSIST DATA

---

## REMOTE DATA – JSON

---

- ▶ `NSURL`: represents a URL that can potentially contain the location of a resource on a remote server
- ▶ `NSURLSession`: provide an API for downloading content via HTTP
- ▶ `NSURLSession.dataTaskWithURL`: creates an HTTP GET request for the specified URL
- ▶ `NSJSONSerialization`: convert JSON to Foundation objects and vice-versa

**XCPSETEXECUTIONSHOULDCONTINUEINDEFINITELY**

It forces the playground to keep continuing executing the code after the end of the code block.

---

## LAB 7

---

Use the `NSURL`, `NSURLSession`, and `NSJSONSerialization` APIs load a remote JSON file and display the contents of its properties with multiple labels.



---

## CORE DATA

---

- ▶ Data held in memory (e.g. an array) is volatile
- ▶ To save the data permanently a persistent storage (file, database, ...) is required
- ▶ Core Data is not a database, also considering that SQLite database is the default persistent store for Core Data on iOS

---

## CORE DATA – THE STACK

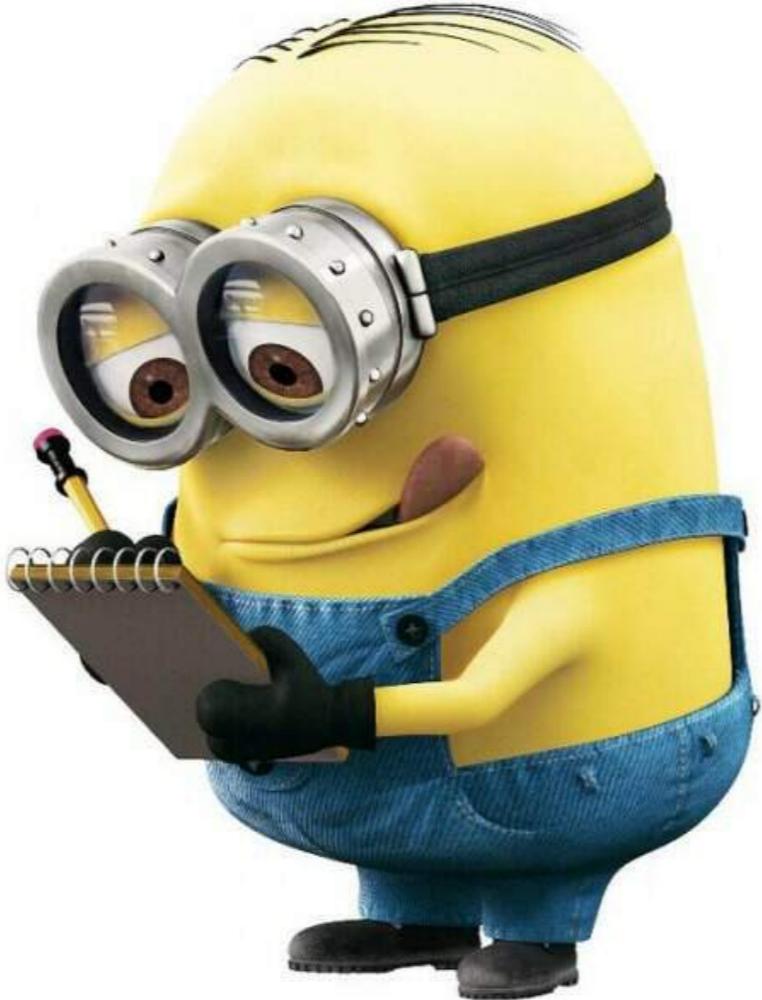
---

- ▶ Managed Object Context: a temporary memory area containing objects that interact with data in persistent store. Its job is to manage objects created and returned using Core Data framework.
- ▶ Persistent Store Coordinator: SQLite is the default persistent store in iOS
- ▶ Managed Object Model: describes the schema that you use in the app (kind of database schema)
- ▶ Persistent Store: the repository in which your data is actually stored (a SQLite database or a binary or XML file)

---

# EXPLORING CORE DATA

---



---

## INTRO TO SWIFT

---

# Q&A

---

**WE ARE DONE!**

---



---

**THANKS!**

---

**GIORGIO NATILI**

- ▶ Optional Information:
- ▶ E-mail: [me@webplatform.io](mailto:me@webplatform.io)
- ▶ Source: [github.com/giorgionatili/swift](https://github.com/giorgionatili/swift)
- ▶ Twitter: [@giorgionatili](https://twitter.com/giorgionatili)