iOS

Some first elements
Development Environment

• Platforms
  — iOS
  — WatchOS
  — TvOS

• Tools
  — Xcode 7.1 (emulator)

• Language
  — Swift 2.1

• Frameworks
  — MVC
Four layers

- The Cocoa Touch layer contains key frameworks for building iOS apps
  - These frameworks define the appearance of your app
  - They also provide the basic app infrastructure and support for key technologies such as multitasking, touch-based input, push notifications, and many high-level system services
- The Media layer contains the graphics, audio, and video technologies you use to implement multimedia experiences in your apps
- The Core Services layer contains fundamental system services for apps
  - This layer also contains individual technologies to support features such as location, iCloud, social media, and networking
- The Core OS layer contains the low-level features that most other technologies are built upon
Model View Controller

Model

Controller

View

notification

data source

delegate (protocol)

target

outlet

action

User action

Update

Notify

View

Update

Model
Simple App

Single View Application

main()

http://petermcintyre.com/topics/ios-app-structure-and-startup/
UIApplicationDelegate

- The interface contains a single property: window, where all of our app content is drawn
- The implementation contains “skeletons” of important methods that allow the application object to talk to the app delegate
  - During runtime events (e.g., app launch, low-memory warnings, and app termination) the application object calls the corresponding method in the app delegate, giving it an opportunity to respond appropriately
- Delegate design pattern
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }

    func applicationWillResignActive(application: UIApplication) {
        // Sent when the application is about to move from active
        // to inactive state
    }

    func applicationDidEnterBackground(application: UIApplication) {
        // Use this method to release shared resources, save user data,
        // invalidate timers, and store application state
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Called as part of the transition from the background to the inactive
        // state
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Restart any tasks that were paused while the application was inactive
    }

    func applicationWillTerminate(application: UIApplication) {
        // Called when the application is about to terminate
    }
}
Views

• Represent user interface elements that display contents or respond to user events
  — Can be nested in a view hierarchy
  — Can animate their property values (animation is critically important to iOS apps)

• Views do not know the role they play
  — For example, clicking a button is always the same, it does not know what it controls
View controllers

• Provide the infrastructure for managing content and for coordinating the showing and hiding of it
  — Manage the views used to display content
  — Communicate and coordinate with other view controllers when transitions occur
• Different view controllers can control separate portions of your user interface
• May also communicate with other controllers, such as data controllers or document objects
View Controllers

• You use custom subclasses of UIViewController to present your app’s content
  — Content view controllers own all of their views and are responsible for the data in those views
  — Container view controllers do not own all of their views; some of its views are managed by other view controllers

• The most common parent classes of Content View Controllers are
  — UITableViewController when your view controller’s main view is a table
  — UICollectionViewController specifically when your view controller’s main view is a collection view
  — UIViewController for all other view controllers
Views and view controllers

• Every view is controlled by only one view controller
  — When a view is assigned to the view controller’s view property, the view controller owns it
• If the view is a subview, it might be controlled by the same view controller or a different view controller
Graphical elements (I)

- UIWindow
- ViewController
- View

rootViewController
View Controller Hierarchy (II)
Overall organization
Storyboards

• A Storyboard is a visual representation of the app’s user interface
  – Shows screens of content and the transitions between them
• A Scene Corresponds to a Single View Controller and Its Views
  – On iPhone, each scene corresponds to a full screen’s worth of content
  – On iPad, multiple scenes can appear on screen at once
  – Scenes are linked together with Segues, these define the transitions between one scene to another
Unified Storyboards for Universal Apps

• Create a single interface for your app that works well on both iPad and iPhone, adjusting to orientation changes and different screen sizes as needed
• Design apps with a common interface and then customize them for different size classes
• You no longer need to create a specific iPad storyboard; instead target the appropriate size classes and tune your interface for the best experience
How to start

Choose a template for your new project:

iOS
Application
Framework & Library
watchOS
Application
Framework & Library
tvOS
Application
Framework & Library
OS X
Application
Framework & Library
System Plug-in
Other

Master-Detail Application
Page-Based Application
Single View Application
Tabbed Application

Game

Single View Application
This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.
Two choices

Choose options for your new project:

- Product Name: MyApp
- Organization Name: Luciano Baresi
- Organization Identifier: it.polimi
- Bundle Identifier: it.polimi.MyApp
- Language: Swift
- Devices: Universal
- Use Core Data
- Include Unit Tests
- Include UI Tests
iOS Simulator

• Easy to start and try apps
• Known problems
  – Different devices
  – Different orientations
• Better positioning of elements
Interface Builder
How to add graphical elements

Do you want to send a message?

Send

Button

UIButton

Implements a button that intercepts touch events and sends an action message to a target object when it's tapped. You can set the title, image, and other appearance properties of a button. In addition, you can specify a different appearance for each button state.
Views (I)

- Action sheet
- Activity indicator
- Alert view
- Collection view
- Image view
- Label
- Navigation bar and items

The volume of the ringer and alerts can be adjusted using the volume buttons.
Views (II)

Progress view

Search bar

Tab bar

Sent from my iPhone

Text view

Tool bar

Table view

Web view
Properties

- **UIColor**
  - `UIColor.greenColor()`
  - `UIColor.lightTextColor()`

- **UIFont**
  - `UIFont.preferredFontForTextStyle()`
  - You should never use fonts directly
UILabel

- Implements a read-only text view
- Set label content using fields text or attributedText
- By default, a label is a single line
  - To create a multiline label, we must increase the value of field numberOfLines

- Special-purpose fields can be used for setting the look and feel of all characters
UITextView

• Implements the behavior of a scrollable, multiline text region
• Supports the display of text using custom style information and also supports text editing
• The appearance of the keyboard itself can be customized using the properties provided by protocol UITextInputTraits
Autolayout

• Determines where objects should go and how big they should be based on constraints we set on them
  — This allows interfaces to adapt to being rotated between portrait and landscape, and to handle differing screen sizes
• Constraints allow us to express what matters to us and to let other factors vary as needed
  — We can specify the size of components, their alignment with or distance from other components, etc.
Four buttons

- Stack layouts views in either a column or a row. It applies Auto Layout to the views without you having to add the constraints.
- Align creates constraints that align a view’s edges or horizontal or vertical center with another (containing) view.
- Pin creates constraints that specify a fixed value for spacing views and/or define a fixed width or height.
- Resolve adjusts a view position or size so it matches its constraints, or do the opposite and create constraints based on its current position and size.
Example

Do you want to send a message?

Center X Alignment Constraint
- First Item: Superview.Center X
- Relation: Equal
- Second Item: Do you want to send a message?.Center X
- Constant: 0
- Priority: 1000
- Multiplier: 1
- Placeholder: [ ] Remove at build time
- [ ] Installed

Add New Alignment Constraints
- Leading Edges
- Trailing Edges
- Top Edges
- Bottom Edges
- Horizontal Centers
- Vertical Centers
- Baselines
- Horizontal Center in Container
- Vertical Center in Container

Update Frames: None
- Add 1 Constraint

View Controller Scene
- View Controller
  - Top Layout Guide
  - Bottom Layout Guide
- View
  - B Send
  - L Do you want to send a message?
- Constraints
  - Center X Alignment - View - Do you want to send a message?
  - Center X Alignment - View - Send

First Responder
- Exit
Constraints and colors

- Orange means that there are not enough constraints
- Blue means that there are enough constraints for an unambiguous auto-layout
  - Constraints must be added explicitly: definitions are not enough
- For complex layouts auto-layout is easier, not trivial, and more dependable
App content

• Xcode provides a library of objects
  – Some of these are user interface elements that belong to a view, such as buttons and text fields
  – Others define the behavior of our app, such as view controllers and gesture recognizers
• A view controller manages a corresponding view and its sub-views
Interface Builder and code

• An IBOutlet connects a variable or property in code to an object in a storyboard
  - This lets us read and write objects’ properties, like reading the value of a slider or setting the initial contents of a text field

• An IBAction connects an event generated by a storyboard object to a method in the code
  - This lets us respond to a button being tapped or a slider’s value changing
Generated code

```swift
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```
Control-click, drag
... and the result is

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view,
    // typically from a nib.
}

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

@IBAction func sendMessage(sender: UIButton) {
    // Code to handle the send message action
}
```
Logging

• On iOS, we can use function print() to write a string out to the system’s log file
  — For example, we can implement our action to just log a message every time the button is tapped

```swift
@IBAction func sendMessage(sender: UIButton) {
    print("Button pressed")
}
```
A first complete example

@IBAction func sendMessage(sender: UIButton) {
  var alert = UIAlertController(title: "Sent",
                               message: "First App Done",
                               preferredStyle: UIAlertControllerStyle.Alert)

  alert.addAction(UIAlertAction(title: "Continue",
                              style: UIAlertActionStyle.Default, handler: nil))

  self.presentViewController(alert, animated: true, completion: nil)
}
Outlets

- Preceding a property with the @IBOutlet modifier tells Interface Builder that a property can serve as an outlet.

- A stored property implies a variable to store the value, but it is managed by Swift and we cannot access it directly.
  - Swift sets up getter and setter methods for us.

- A computed property does not imply a backing variable, but instead has get() and set() methods that we provide to compute the property’s value on the fly.
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

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

    @IBAction func increment(sender: UIButton) {
    }
}
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var counterLabel: UILabel!

    var counter = 0

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

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

    @IBAction func increment(sender: UIButton) {
        counterLabel.text = String(counter++)
    }
}
Weak attribute

• Automatic Reference Counting (ARC) solves almost all memory problems, but it cannot solve retain cycles
• Our ViewController knows about the UILabel, so ARC cannot free the label from memory so long as the view controller exists
• But if the UILabel also requires the ViewController, then neither can ever be freed from memory (cycle)
• The way to break this is to declare one side of the arrangement as weak, meaning that we do not require an object to be in memory if ours is the only one that knows about it
Why it works

• The top-level view has strong references to all its children (including the label)
• The view controller has a strong reference to the view, so having an additional strong reference from the view controller to the label would be overkill
• The rule of thumb is that
  — Only “top-level” objects in a storyboard scene (like the view) need strong references, and everything else can be weak
  — Xcode defaults to this behavior when we made the connection
Lifecycle

1. **Appearing**
   - `viewWillAppear`
   - `viewDidAppear`

2. **Appeared**
   - `viewWillDisappear`
   - `viewDidDisappear`

3. **Disappearing**
   - `viewWillAppear`
   - `viewWillDisappear`

4. **Disappeared**
   - `viewDidAppear`
   - `viewDidDisappear`