Developing iOS 11 Apps with Swift

# Introduction

**What’s in iOS?**

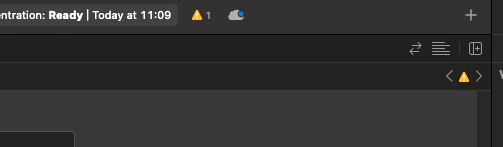
* Cocoa Touch (UI Layer, Multi-Touch, Alerts, Web View, Core Mtion, Map Kit, Camera, etc)
* Media (Core Audio, OpenAL, PDF, JPEG, PNG, etc)
* Core Services (Collections, Core Location, Address Book, Net Services, Networking, Threading, File Access, Preferences, SQLite, URL Utilities)
* Core OS (OSX)

When you create an app, you will have the navigation pane (at the left, with the app files and other option like breakpoints), the utilities pane (right pane), the debugger pane (bottom pane) and the options to run the app and the selected device to run the app, at the top of the screen.

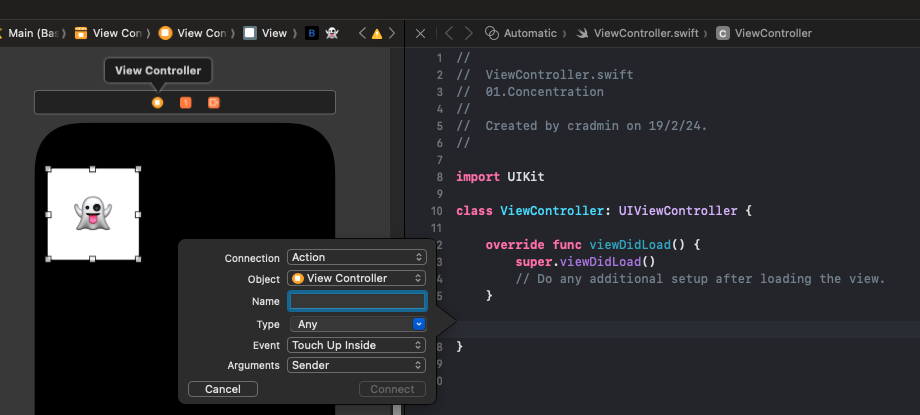
To add objects to the Layout, you need to open the storyboard, for example the Main and click on the + button at the top right corner. It will display a popup with different views you can add.

## Creating a Connection - Using @IBAction

To create a method for an object of the view, for example, the method when the user clicks on a button, the simplest way is to open the storyboard screen and the view controller class of the view. You just need to click on the icon of several lines at the top right corner, the one for the Assistant option:



Then, click on the view or the object you want to create the method or event, press Ctrl and drag and drop to the Code screen. That should show a popup like this one:



With this you can create a connection between the UI and the controller. In this popup you can choose the type of connection (Action for method), the name of the action, the type, for example, UIButton, the type of event and the arguments you want for the method.

It will create something like this:

**@IBAction** **func** touchCard(\_ sender: UIButton) {

}

In Swift, each parameter has two names. An internal (first name) and an external name (second name), which is used when you call the function, so you don’t have to remember what parameter represents the value you are sending. If you don’t want to have two different names for the parameters, you can use underscore like in the example above, this way, both internal and external names are going to be the same.

Tip:

To see the documentation about a method or value, you can hold the key Option and click on it (in Window’s keyboard it is the Alt key).

Example:

**@IBAction** **func** touchCard(\_ sender: UIButton) {

flipCard(withEmoji: "👻", on: sender)

}

**func** flipCard(withEmoji emoji: String, on button: UIButton){

print("flipCard(withEmoji: \(emoji))")

**if** button.currentTitle == emoji {

button.setTitle("", for: UIControl.State.normal)

button.backgroundColor = UIColor.orange

} **else** {

button.setTitle(emoji, for: UIControl.State.normal)

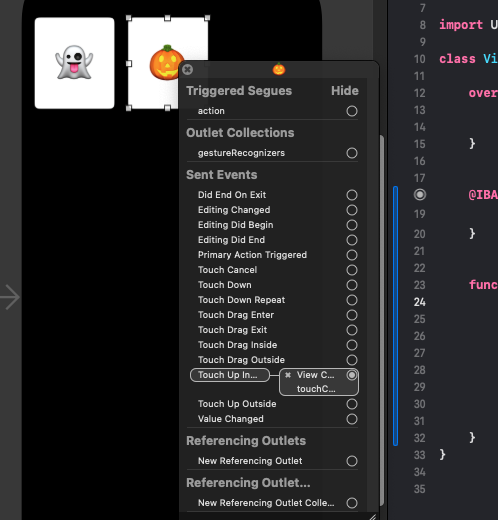
button.backgroundColor = UIColor.white

}

}

To interpolate values on a string message, like in the print line, in Swift you use \(value) where value can be a variable name.

If you right-click on something in the UI, you will see everything that thing is connected to. For example:



Swift is strongly typed language, however, you can initialize a variable and Swift can infer the type of it with the value, for example:

**var** flipCount = 0

Swift knows this is an Int variable.

## Creating an Instance Variable – Using @IBOutlet

Just as you created a connection for an UI element to create a method, to create an instance variable for an UI element, you just need to press Ctrl over the element and drag it to the code view and this time, you chose Outlet as the Connection type.

Example:

**@IBOutlet** **weak** **var** flipCountLabel: UILabel!

Swift offers the property observer, using didSet, that means that every time the value of property changes, the code inside the didSet will be executed. For Example:

**var** flipCount = 0 {

**didSet** {

flipCountLabel.text = "Flips: \(flipCount)"

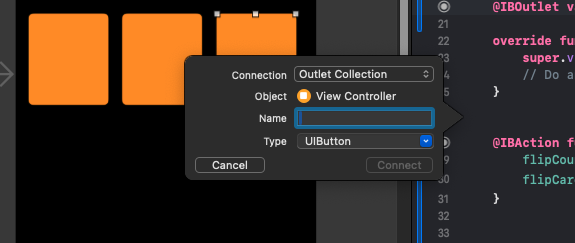
}

}

Every time we update the flipCount value, the flipCountLabel will be updated too to show the text with the counter updated.

## Creating a Collection of UI elements – Using @IBOutlet

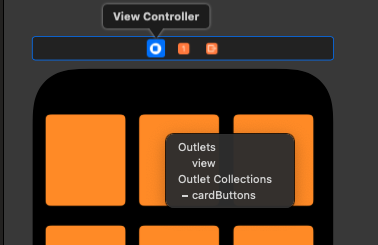
When you need to create a collection, aka an Array in swift for UI Elements, you do the same as to create an instance or a method, but chose the Outlet Collection option:



Tip:

If you need to rename a variable name or instance, you can press Ctrl and click over the variable name, it will display a menu with the option Refactor > Rename to change all the instances.

Another way to hook up elements with the code, is to select the yellow button in the top bar of the UI, press Ctrl and drag it to the UI element and select the outlets to relate it to.



In Swift, when you declare a value and won’t change it later, in other words, when you use a constant, you use **let** instead of var.

**@IBAction** **func** touchCard(\_ sender: UIButton) {

flipCount += 1

**if** **let** cardNumber = cardButtons.firstIndex(of: sender) {

print("cardNumber = \(cardNumber)")

} **else** {

print("Chosen card was not in the cardButtons")

}

}

The if let… allows you to handle optionals, for example, in the previous code the method firstIndex from cardButtons returns an optional, that means, it could be an Int or a nil value, so, to avoid the app crashing for unwrapping a nil value, you can use this syntaxis to validate this value.

# MVC

MVC is the architectural pattern that tries to separate concerns of the application in 3 parts:

* **Model**: “What” your app does, but not how it is displayed.
* **Controller**: “How” your Model is presented to the user, the UI logic. Controllers can always talk directly to their models and their views.
* **View**: The UI to present to the user. The View can communicate with the Controller, but it has to be a “blind” and structured communication. It is like if the View sends an action to the controller when things happen in the UI, sometimes with the use of a delegate and the controller processes this with a protocol.

The Model and View don’t have a communication between each other.

The controller can drop a target on itself.

The view sends the action when things happen in the UI.

Sometimes the View need to synchronize with the Controller and this can be done with a delegate. The controller sets itself as the View’s delegate. The delegate is set via a protocol.

Controllers are almost always the data source, not model. Controllers interpret/format Model’s information for the View.

The Model doesn’t communicate directly with the controller. The way to communicate any update from the Model to the controller is with a broadcast mechanism.

## Struct vs Class

There are 2 major differences between Struct and Classes in Swift:

* Structs don’t allow inheritance.
* Structs are value types and Classes are reference types. Value types are copied when you pass them on some methods, reference types pass the reference of the object.

MVC 20