Interactive Graphics

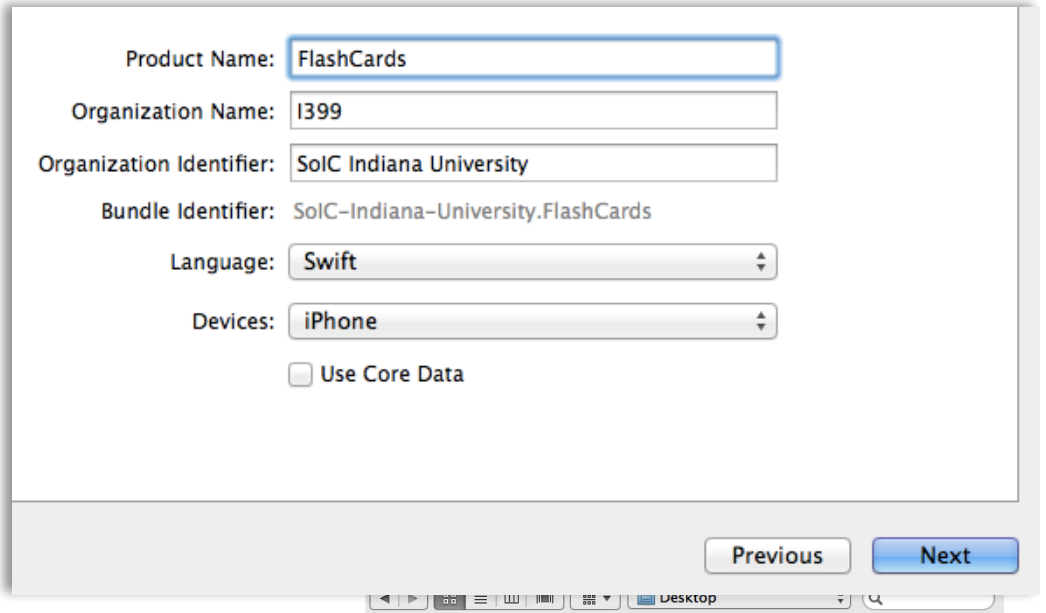
CSCI B481 – Spring 2018

Lab 06 – practice with apps in Swift

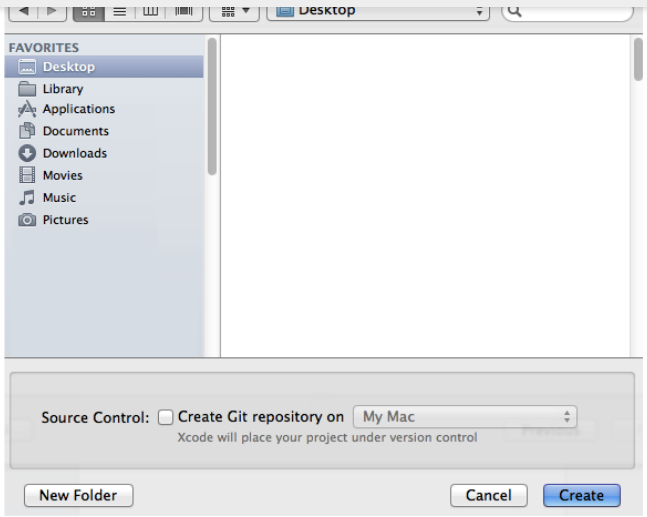
Instructor:
Mitja Hmeljak,
http://pages.iu.edu/~mitja
mitja@indiana.edu

An iOS App: "*FlashCards*"

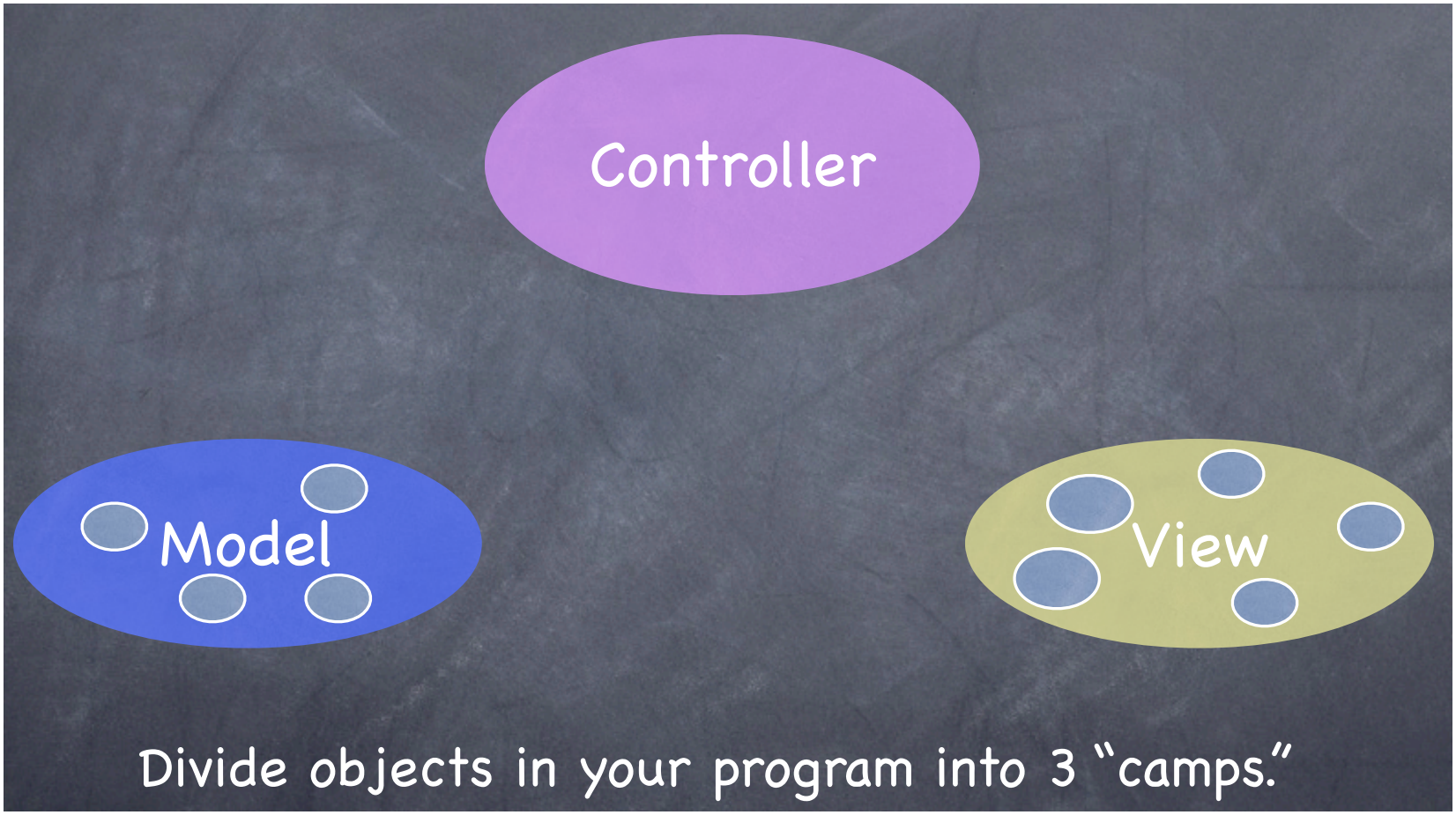
name it "FlashCards",
set *Language* to *Swift*,
set *Devices* to *iPhone*,
do *not* select "~~use Core Data~~",
click *Next* ...



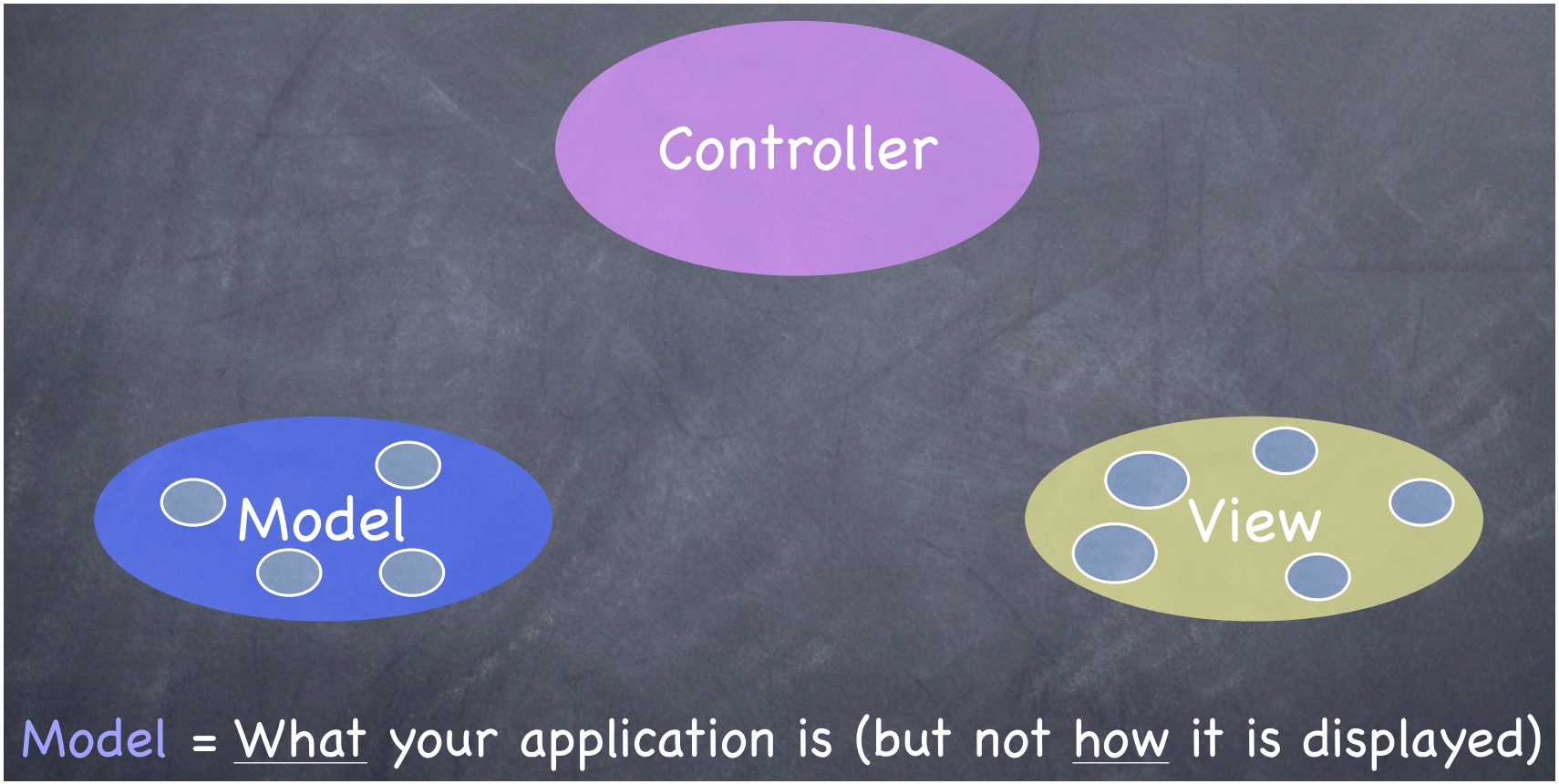
then save the project to the Mac's *Desktop*:
... select "Desktop" either from the sidebar
or from the filesystem navigation
pop-up menu, do *not* select
"~~Source Control~~", and click *Create*.



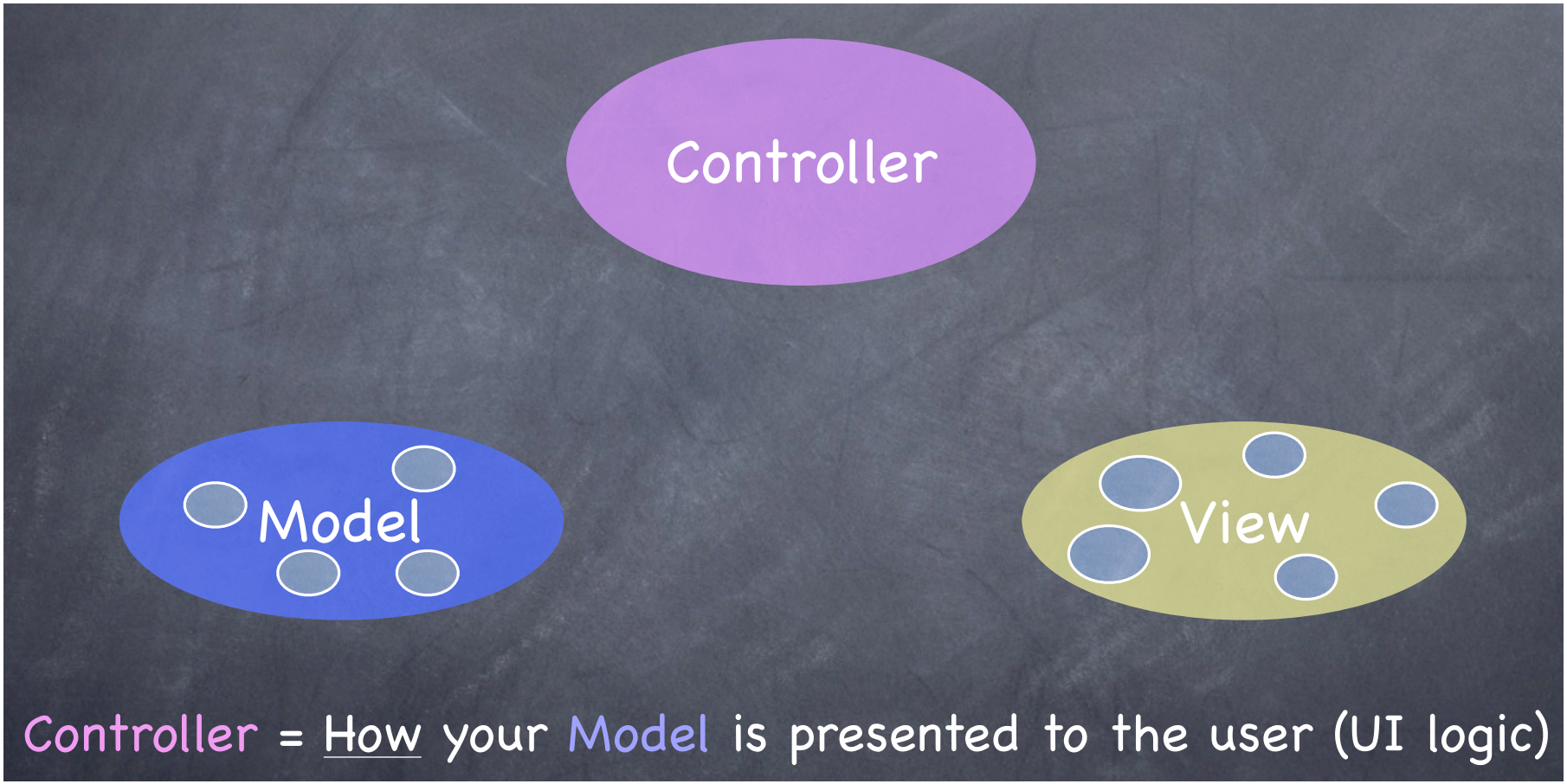
iOS Programming: Model-View-Controller



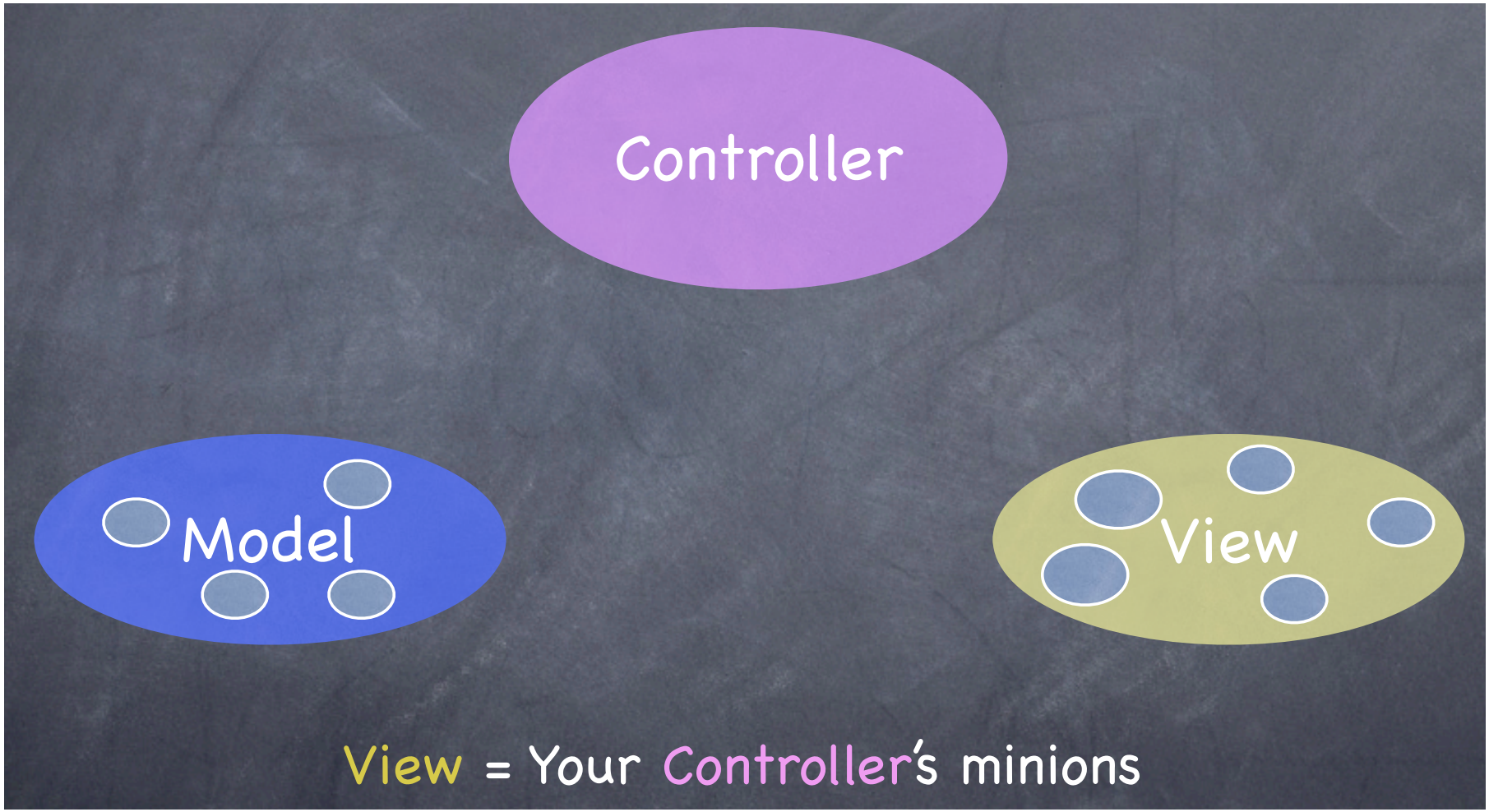
iOS Programming: M-V-C



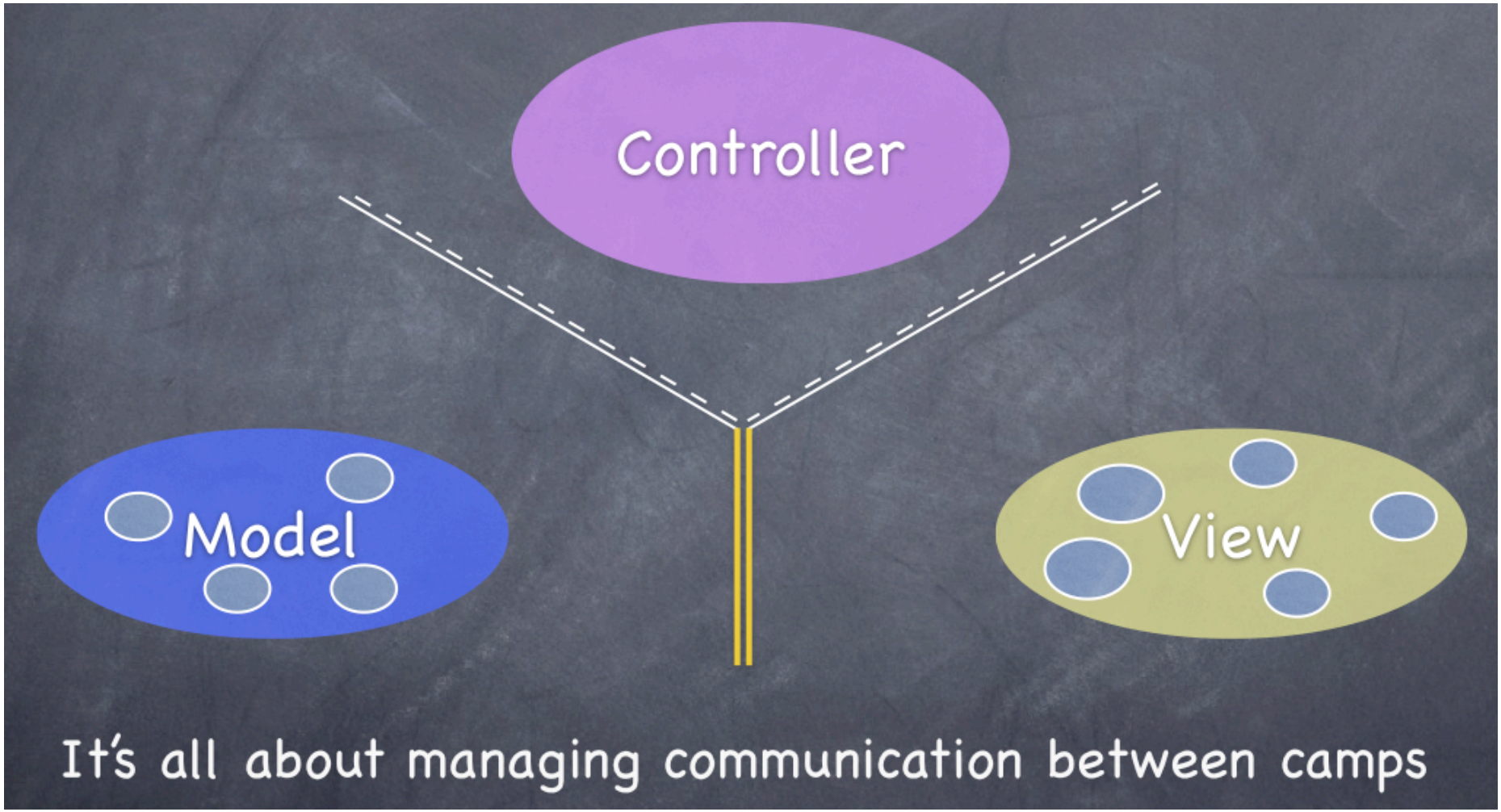
iOS Programming: M-V-C



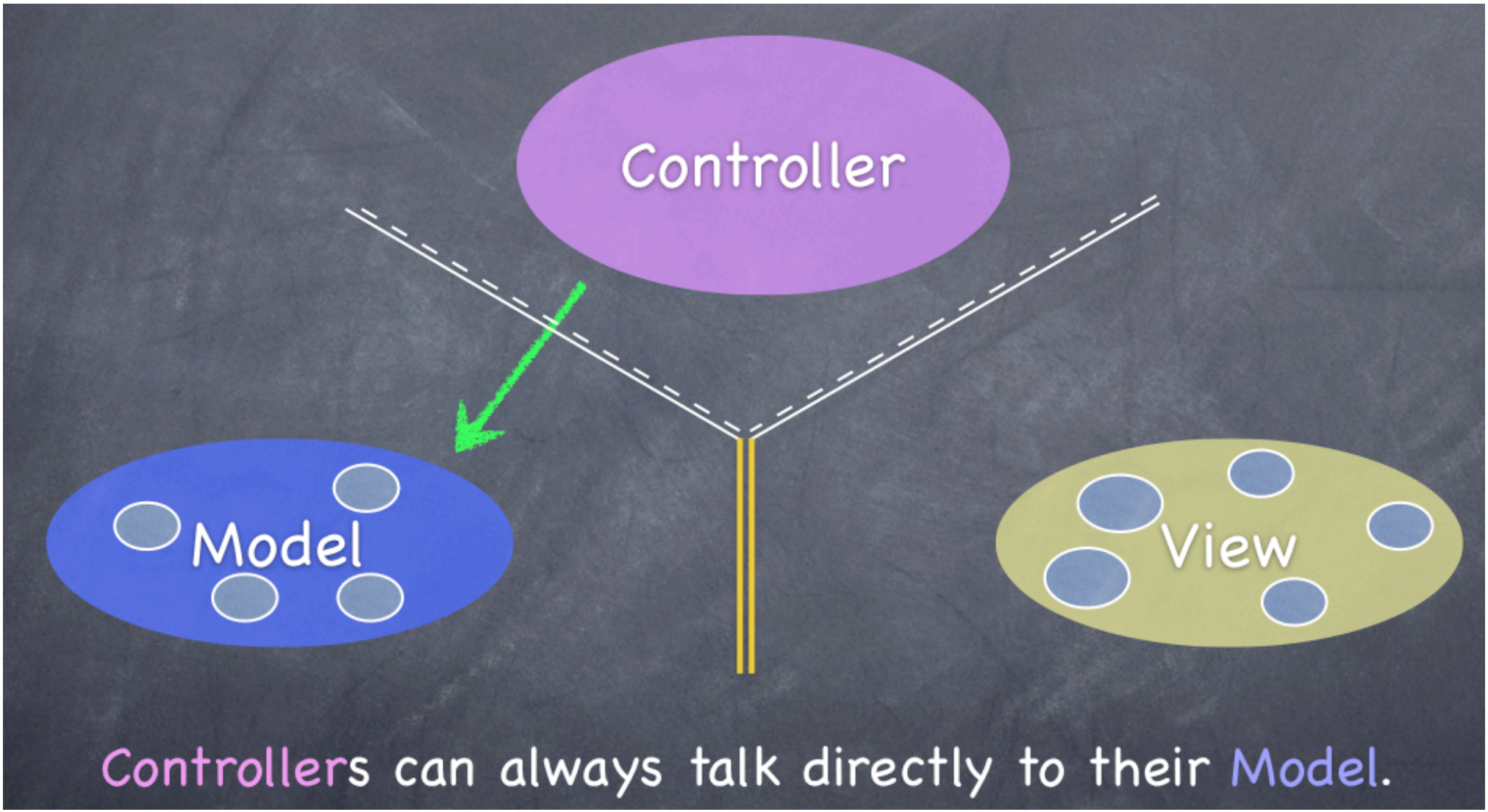
iOS Programming: M-V-C



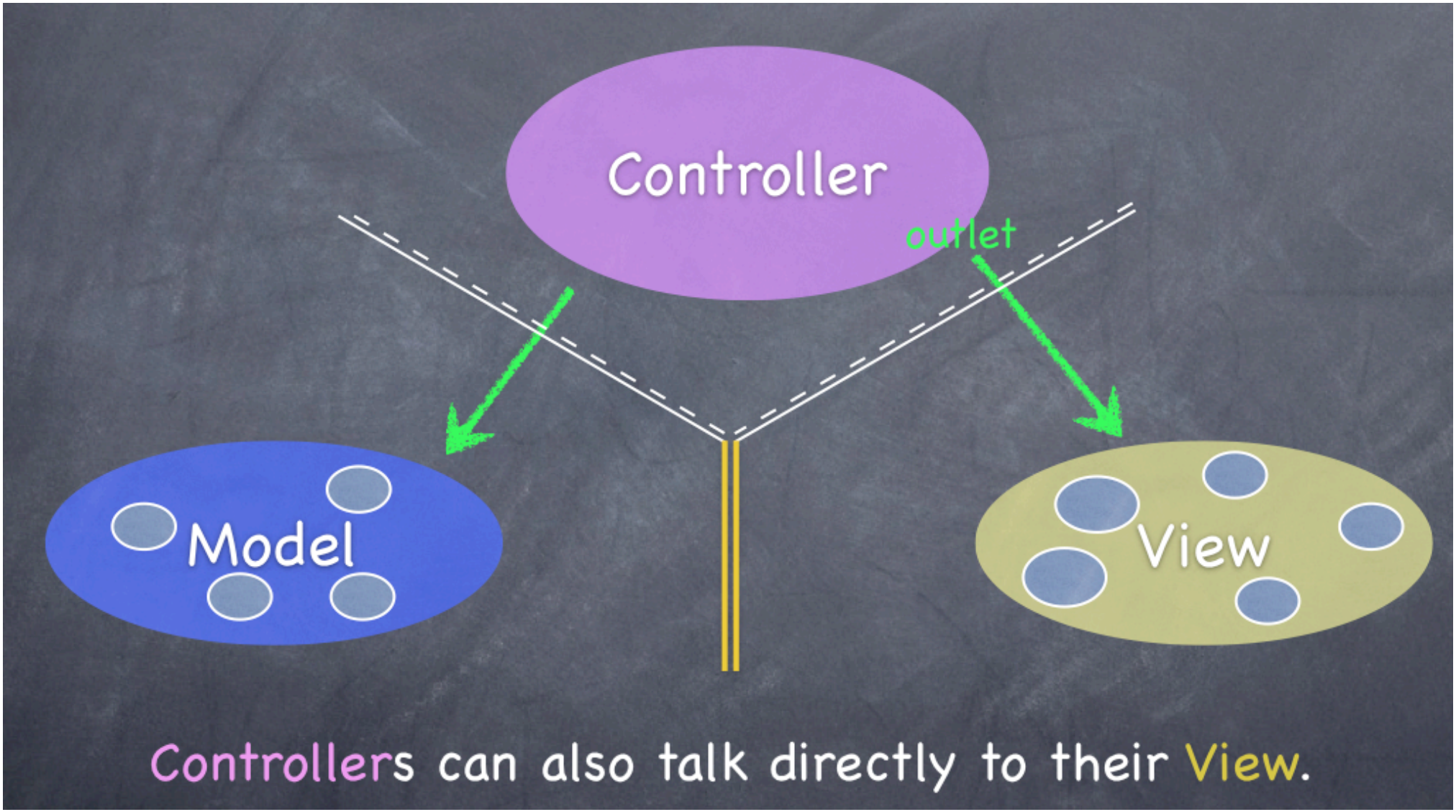
iOS Programming: M-V-C



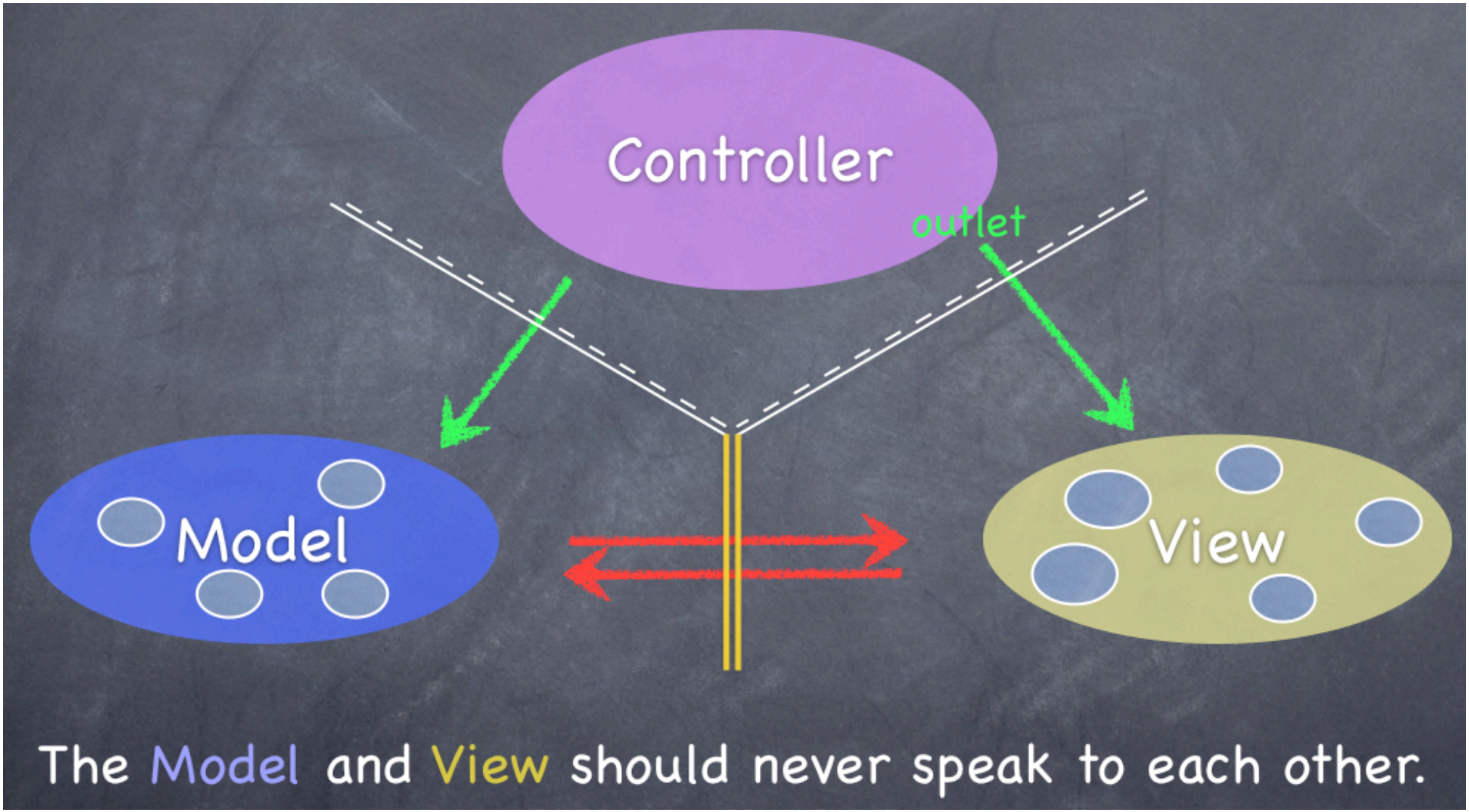
MVC: Model ←→ Controller



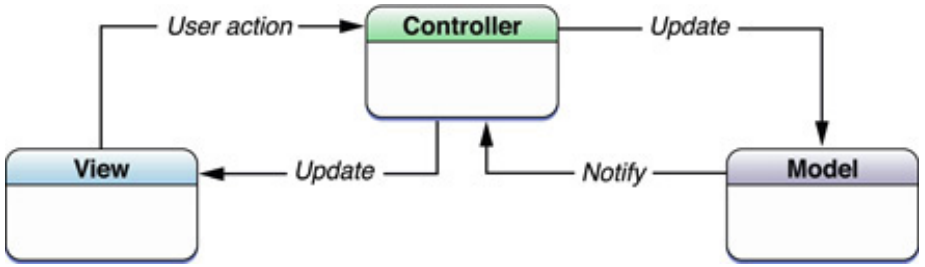
MVC: View ←→ Controller



MVC: View ←≠≠≠≠→ Model



the M-V-C design for the *FlashCards* app



View Objects: we'll need

- two UILabel instances
- two UIButton instances
- (four view objects total)

Model Objects: we'll need

- two *Array* object instances
  we'll use the Array type, which is a "collection type" in Swift
  - (note → we are going to build this part in a *subsequent lecture*)

Controller Objects: we'll need

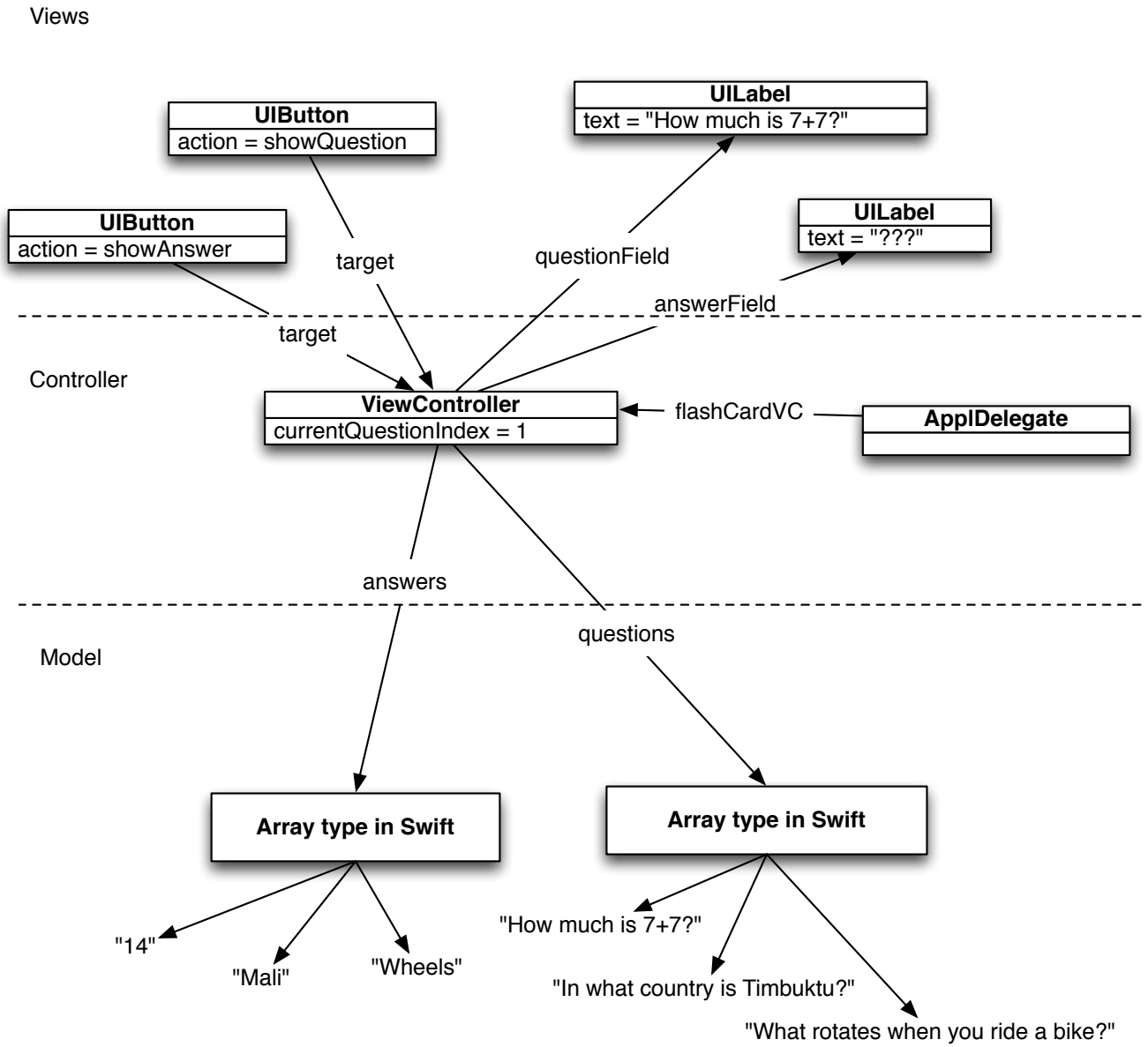
- one AppDelegate object instance
- one FlashCardController object instance

the M-V-C design for the *FlashCards* app

View Objects:

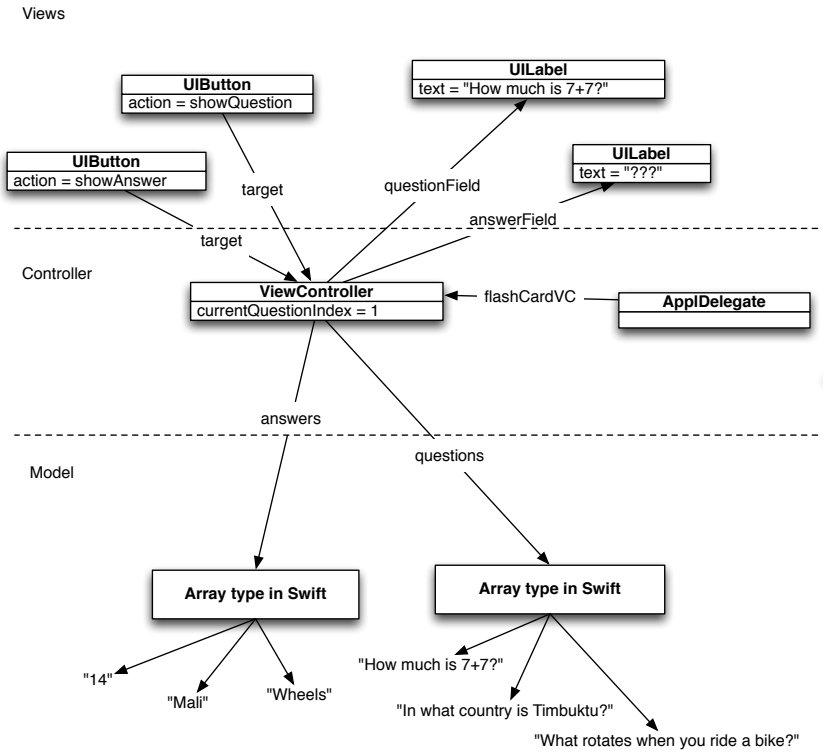
Controller Objects:

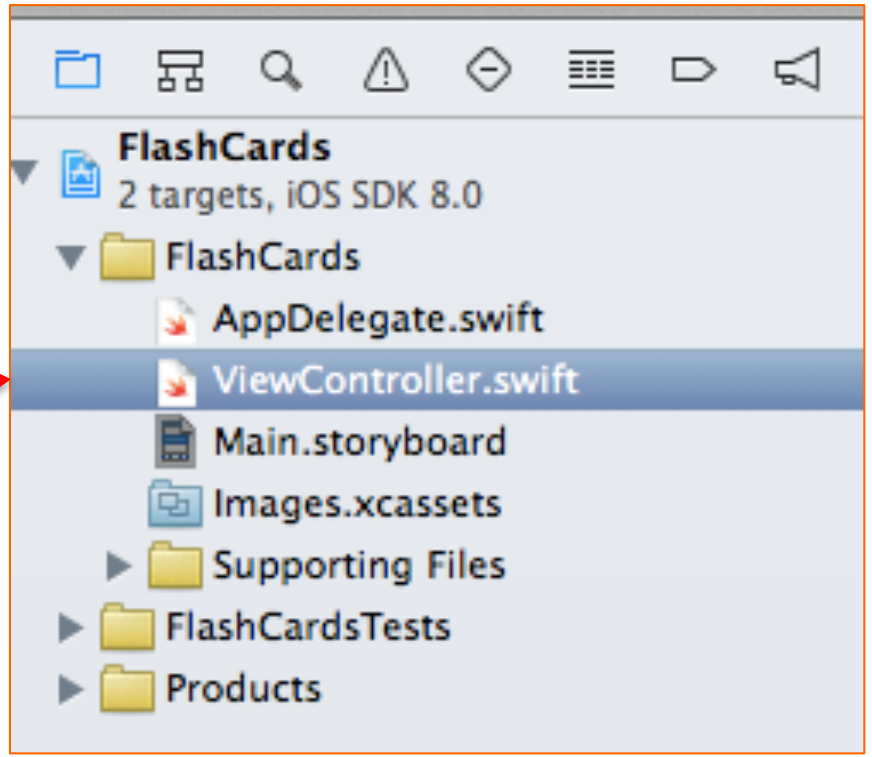
Model Objects:



the M-V-C design for the *FlashCards* app

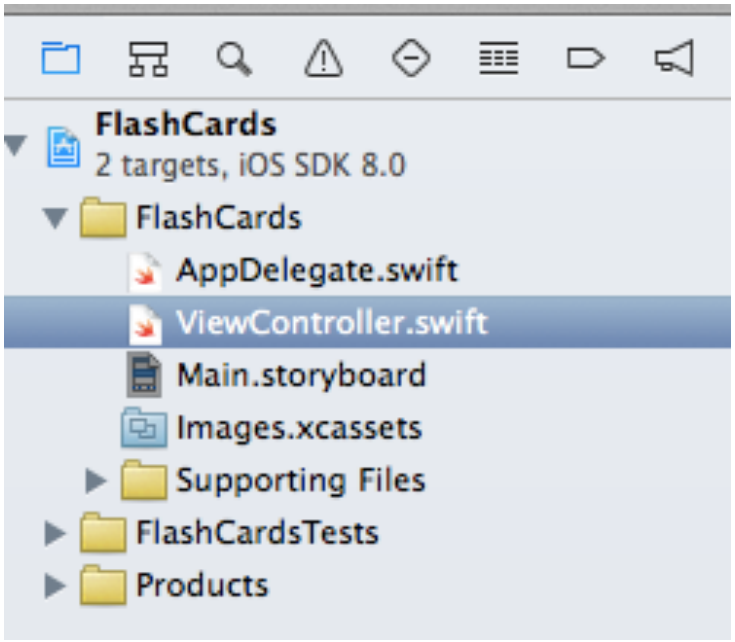
with this overview of the objects we're going to use in our application, we can start building them in our Xcode project...





the View Controller for the *FlashCards* app

In *Xcode*, you'll find *two* new files in the *project navigator (compared to the empty project we used in Lab 01)* :



- a new *source code* file named "ViewController.swift"
- and a *Storyboard* type of file named "Main.storyboard"

In the filesystem, these two files are located thus:

- the *.swift* file is stored inside the "FlashCards" *source code folder* (which is inside the "FlashCards" *project folder*)
- and the *.storyboard* file is inside a folder named "Base.lproj" stored inside the *source code folder*.

the View Controller for the *FlashCards* app

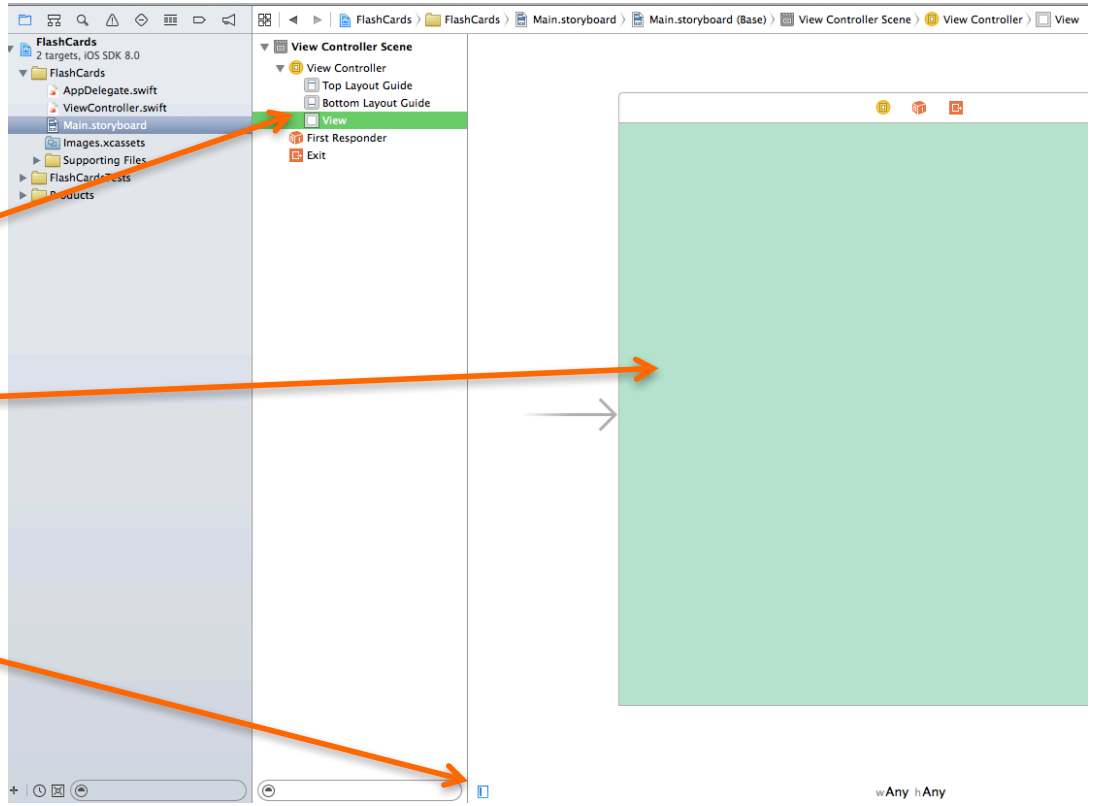
if you select any *.storyboard* file in the *project navigator*, it will open inside the main editor area, i.e. the *Interface Builder* tool in Xcode, where you can add, edit, arrange GUI objects.

There are two parts

in the *editor* area:

- the *dock* on the left
- the *canvas* on the right

(*to reveal the dock*, you may have to *click* on the "disclosure button" on the bottom left of the canvas)



building the View for the *FlashCards* app

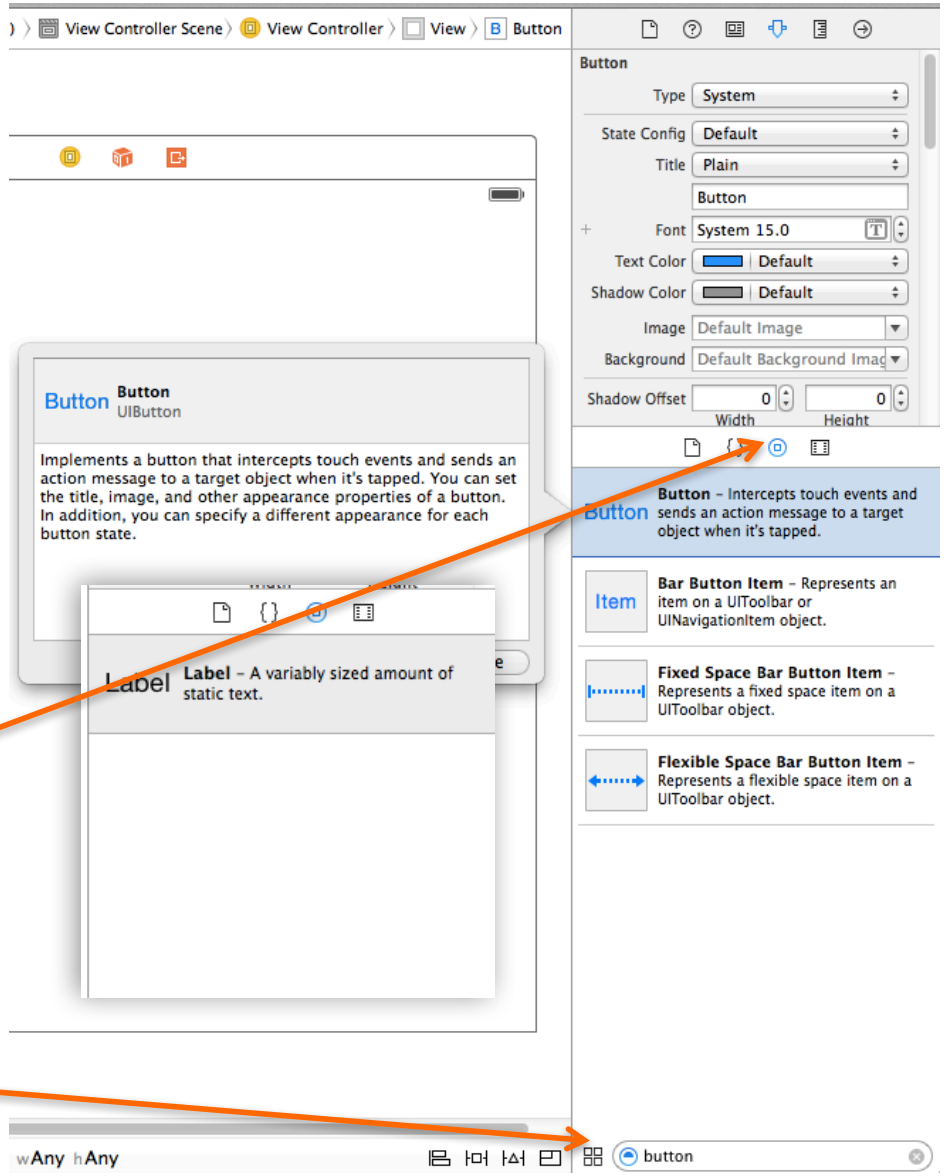
select the View object (listed in the InterfaceBuilder *dock* as part of the "View Controller"), which is an instance of UIView, the foundation of your app's user interface.

We'll need two additional objects in this view:

- two *labels* and
- two *buttons*.

find these in the *utility area* on the left side of Xcode's window, in the third *tab* of the *library* (the *library* is the *bottom section* of the utility area).

You may use the search entry to type "button" and "label".

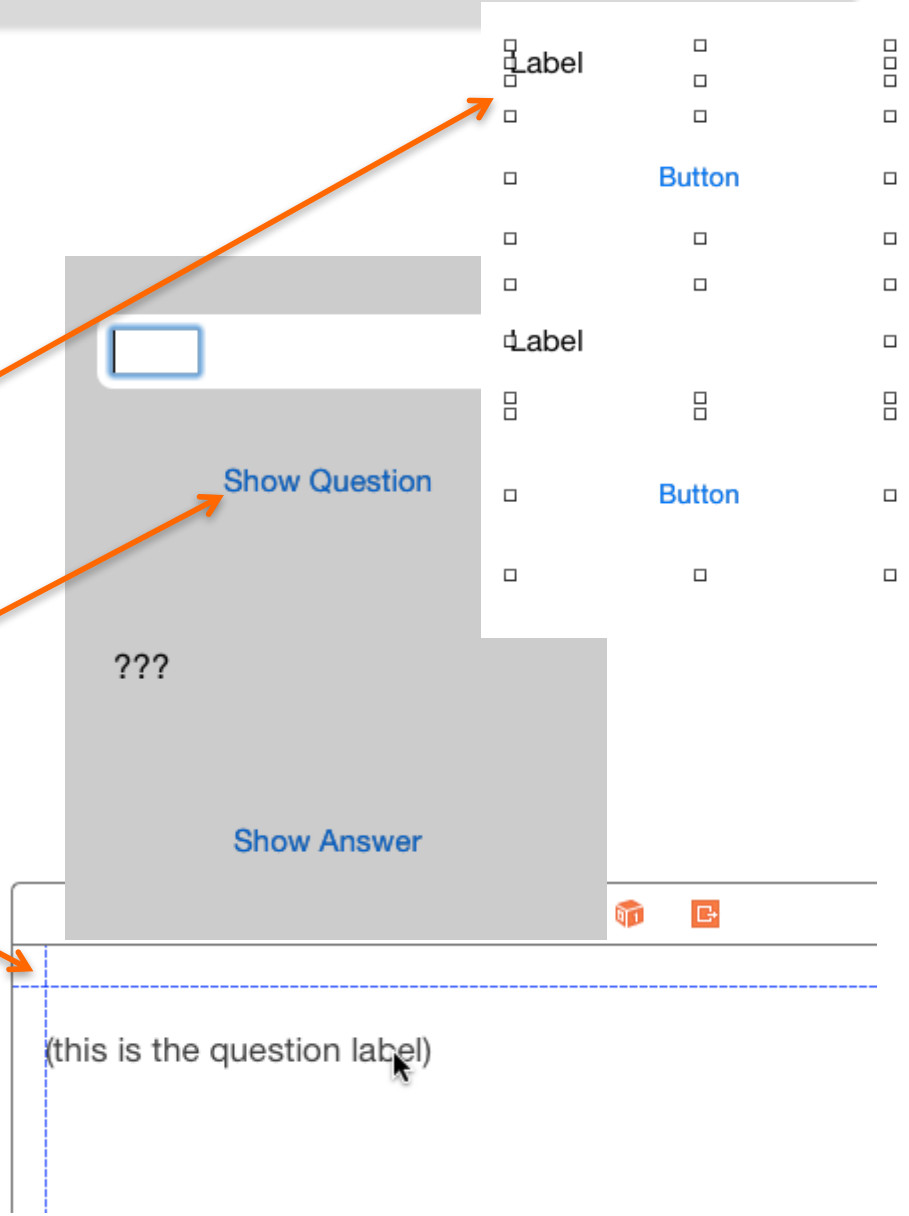


the View Controller for the *FlashCards* app

- ***place*** these two *labels* and two *buttons* in the main View, then edit some of their properties directly on the canvas
- ***resize*** the *buttons* and the *labels* so that they cover most of the area
- ***edit*** the text of the two *buttons* as shown, leaving the *top label blank* – this label will be used to display a question to the user.

Ensure that ***all*** UI elements ***align*** to the "blue dashed lines" ...this will help with *autolayout* for screen UI elements.

<u>*Note*</u>: remember to save your work in Xcode often. For example, to save the *Main.storyboard* changes, select *Main.storyboard* in the *project navigator*,then press *[command ⌘]-S* on the keyboard (or select the *File→Save* menu).



the View Controller for the *FlashCards* app

Using Xcode's *project navigator* (the leftmost pane in Xcode's window) let's go back to the *ViewController.swift* file and ***add*** two ***properties***, i.e. variables that will be used together with the Label view objects:

Notice two differences from variables we've used in Swift so far.

```
// framework we import:
import UIKit

// a subclass of the UIViewController class:
class SecondViewController: UIViewController {
21      // Add two new properties to the ViewController:
        @IBOutlet weak var questionTextField: UITextField!
23
        @IBOutlet weak var answerTextField: UITextField!
```

First difference: these variables are *prefixed* with "@IBOutlet"

Here's why: Interface Builder scans your source code looking for any properties in your view controller prefixed with the "@IBOutlet" keyword. Xcode exposes to Interface Builder any "@IBOutlet" properties it discovers, so you can ***connect*** them to views.

the View Controller for the *FlashCards* app

Second difference: these variables are defined with an exclamation mark (!) appended to their type.

```
// framework we import:
import UIKit

// a subclass of the UIViewController class:
class SecondViewController: UIViewController {

    // Add two new properties to the ViewController:
    @IBOutlet weak var questionTextField: UITextField!

    @IBOutlet weak var answerTextField: UITextField!
```

Here's why: the (!) sign *appended to the variable type* indicates that these variables have *optional values*, but they are *implicitly unwrapped* ...and what does this actually mean?!?

Practically speaking, this means that you can write Swift code *assuming* that these variables will have a set value by the time you use them .... but your app will *crash* if you try using such a variable without a value!

In Swift these types are called *implicitly unwrapped optionals* (i.e. those having an (!) sign at the end of their type definition), and are used to define variables that will be certainly set up before being used, for example any user interface elements we create in the Storyboard.

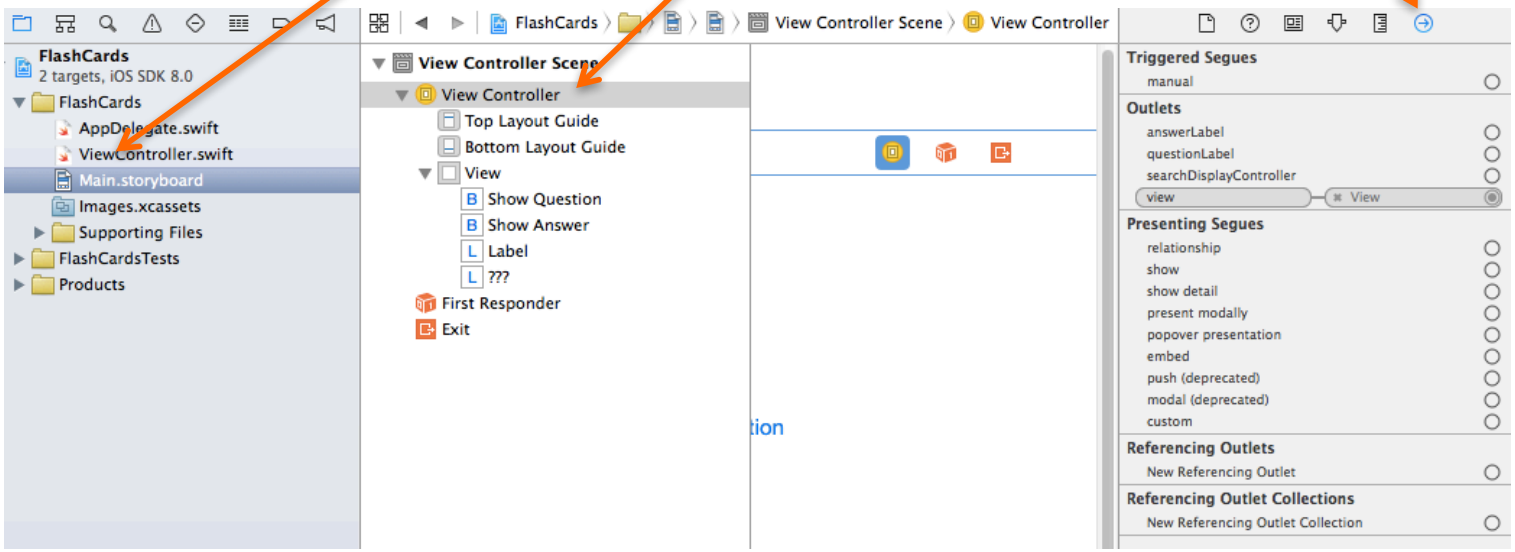
(for a bit more details, an example about "optionals" in Swift is available here: http://www.drewag.me/posts/what-is-an-optional-in-swift)

the View Controller for the *FlashCards* app

Now to ***connect*** *properties* (i.e. the two variables we just defined in ViewController.swift) *to the user interface elements*

1. in Xcode's *project navigator* select Main.storyboard, then
2. in Interface Builder's *dock* select View Controller, then
3. in the Inspector, select Connections (the 6th tab).

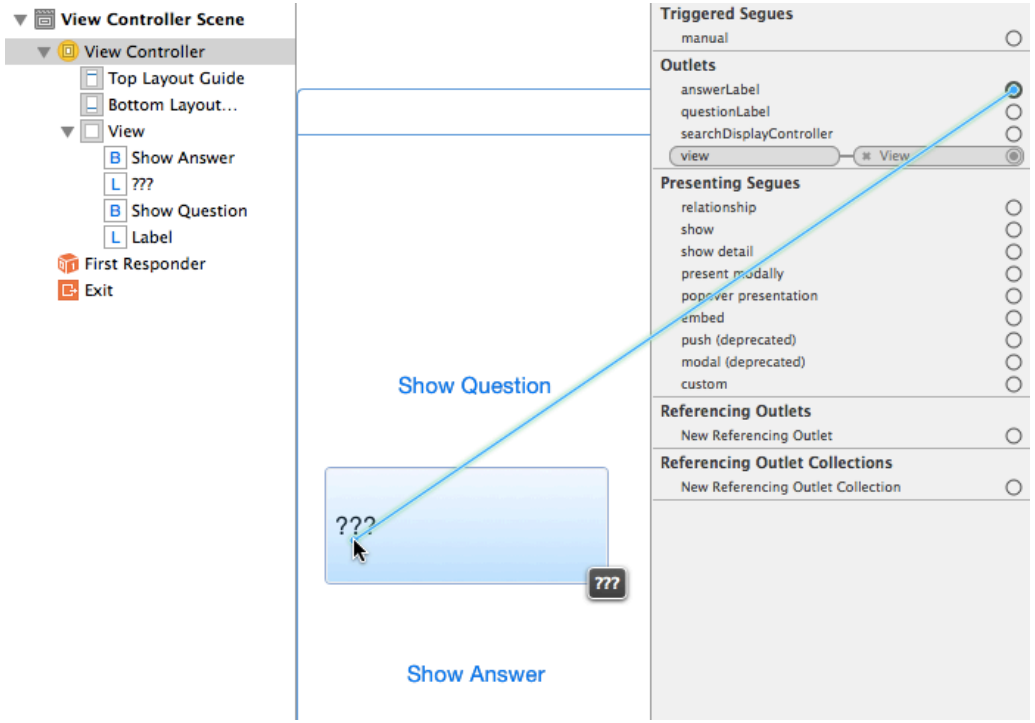
You'll see that all *the properties you created in ViewController.swift* are now listed in the Outlets section:



the View Controller for the *FlashCards* app

In the "outlets" listed in the Connections tab, there is a *circle* on the right of every outlet. For example, there is a circle on the right of *answerLabel*.

Using the mouse, ***control-drag*** (i.e. ***hold*** the "control" key on the keyboard while ***clicking*** the mouse button, then ***dragging*** the mouse) from the circle on the right of *answerLabel* in the inspector, to the "???" *label* above the Show Answer button in the Canvas, then release the mouse button to ***connect*** the "*answerLabel*" Swift property to the "???" *label*.



Complete the same control-drag connection for the *questionLabel*:
connect it to the *label* above the Show Question button in the Canvas.

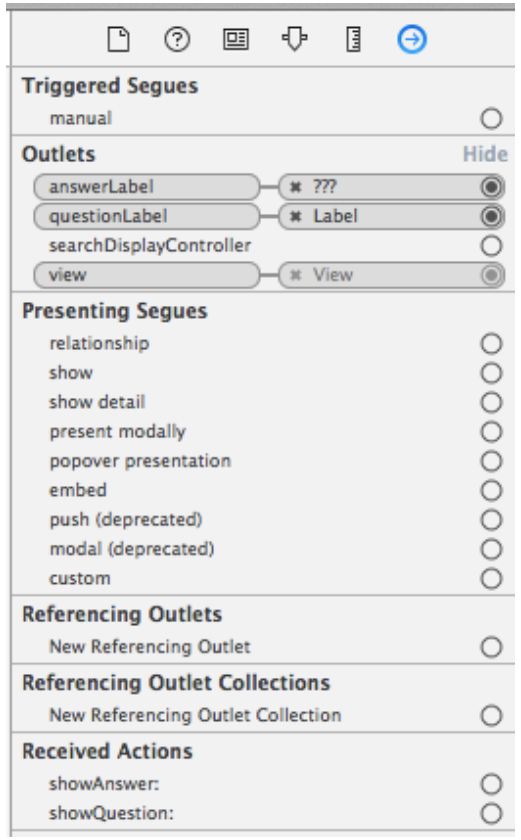
the View Controller for the *FlashCards* app

To connect the two buttons in the view to your *controller*,

type the following code inside your ViewController.swift file, anywhere in your ViewController class:

```
26      // Methods to handle buttons
        @IBAction func buttonOKAction(sender: AnyObject) {
```

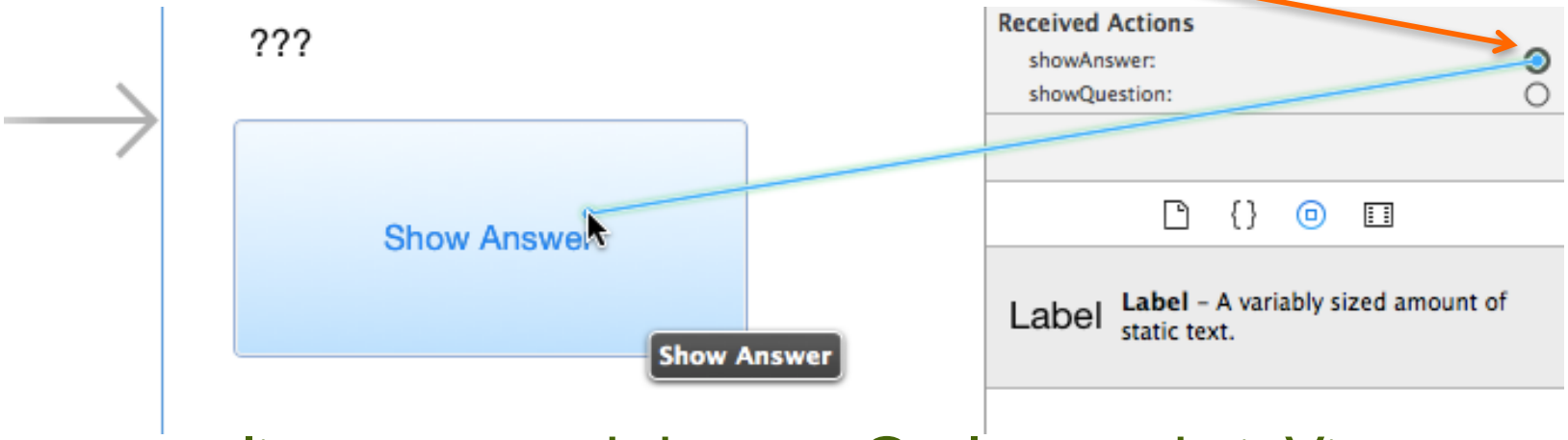
the "@IBAction" keyword is used by Xcode to list these methods in the *Received Actions* section in the *Connections* tab of the *Inspector*:



the View Controller for the *FlashCards* app

The *ViewController.swift* file now contains two new methods to interface to the buttons in the View.

Connect (*) these methods from the *Received Actions* section of the *inspector* to the View elements in the Canvas, i.e. the two buttons "Show Answer" and "Show Question":

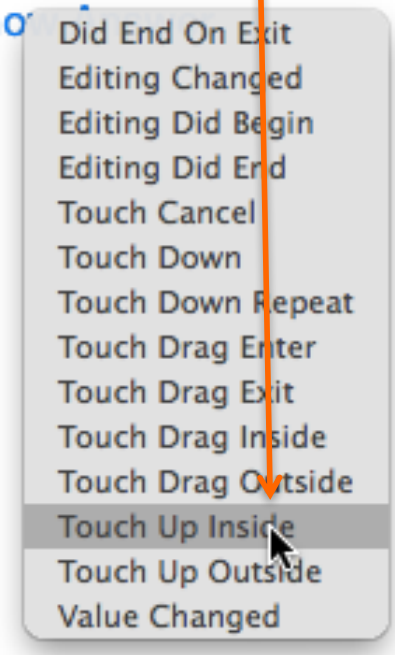


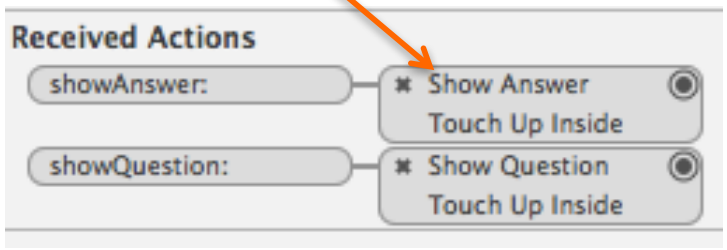
(*) just as you earlier connected the two Outlets to their View elements, but this time...

the View Controller for the *FlashCards* app

... you will also need to ***select*** "Touch Up Inside" in the pop-up menu that shows up when you connect the Received Actions to their respective Views.

Select this for both methods/buttons, so that the connection will show up as "Show Answer" and "Touch Up Inside" in the inspector





testing your FlashCards app so far

to make sure that everything is updated in your App executable, while in Xcode

press *[command ⌘]-[Clean ⇧]-K* on the keyboard (or select the *Product→Save* menu), to remove compiled/executable code from your project, then

press *[command ⌘]-R* on the keyboard (or select the *Product→Run* menu), to run your freshly compiled app in the iOS simulator.

While the app has no functionality yet, the buttons will show that they "work" by changing their appearance when you *tap* (or while in the iOS Simulator, when you *click*) them.

adding some "behavior"

to add some visible "behavior" to our app,
(even before writing any Model code)
we'll add some code to the two "@IBAction" functions that were left empty before:

```
26      // Methods to handle buttons
        @IBAction func buttonOKAction(sender: AnyObject) {
28          print ("self.questionTextField.text = \(self.questionTextField.text)")
29          print ("self.answerTextField.text = \(self.answerTextField.text)")
```

compile and run the app again, and test it by pressing on either one of the two buttons in the User Interface.

Let's look at... running an iOS app

In general, the main entry point for an application at launch time is usually the *main()* function.

For an iOS app, the main function's job is to hand control over to the UIKit framework.

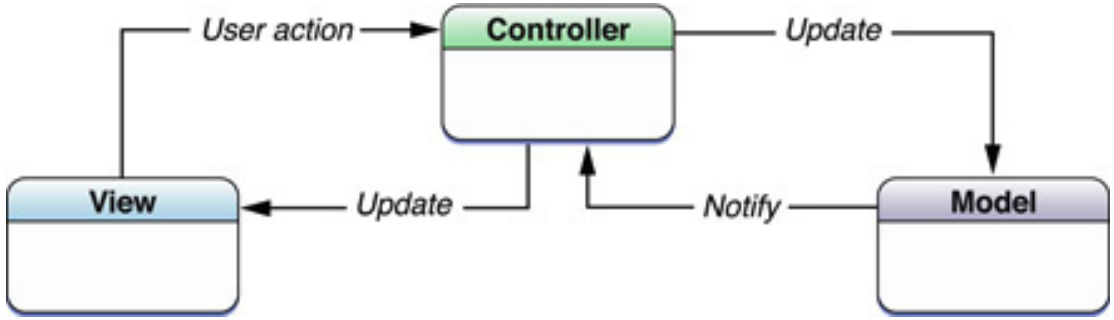
However, in the Swift programming language there is no need for a *main()* function (although it's possible to have one) since the Swift "code written at *global scope* is used as the entry point for the program, so you don't need a *main()* function."

…what is being run, actually?

iOS apps generally follow the

Model-View-Controller

design pattern.



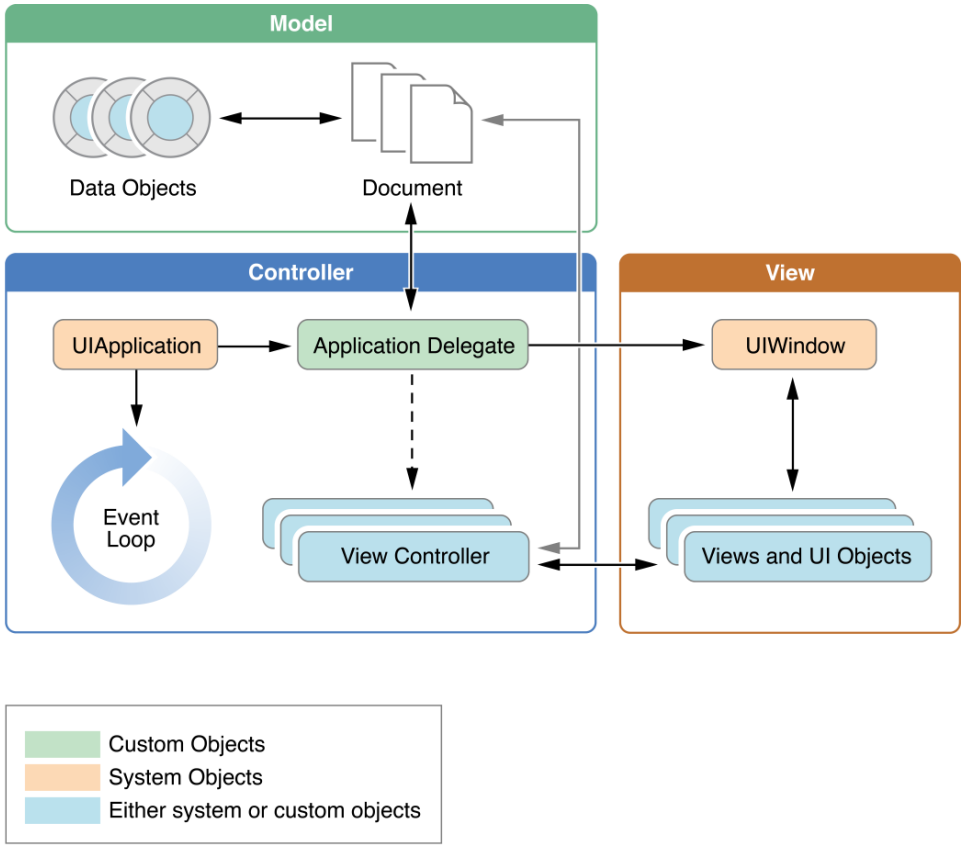
The M-V-C pattern divides your program into three blocks:

- Model
- View
- Controller

the *Model* in M-V-C

the *Model* block of the M-V-C pattern corresponds to "what your application *is* or *does*",

(the Model is *not* "*how* does it look" nor is it "how do I interact with it".)



the *Model* in M-V-C

Model objects

encapsulate the data

specific to an application

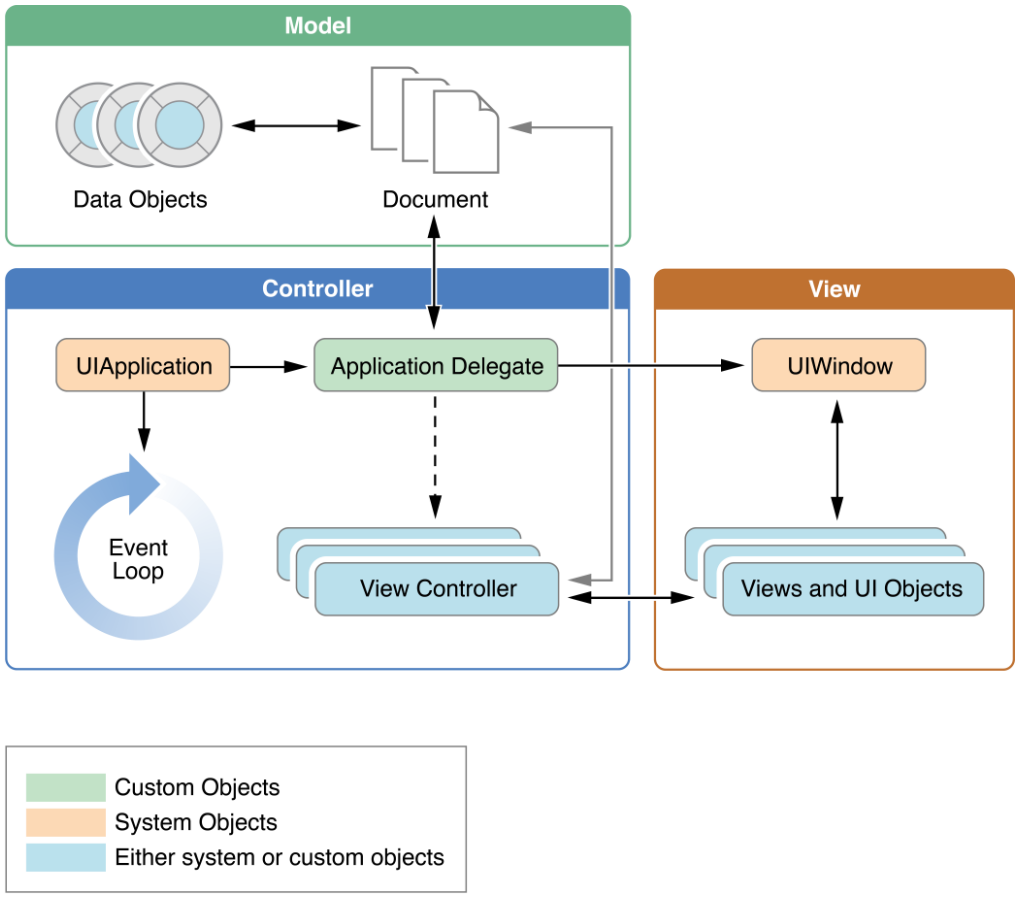
and define

the *logic*

and *computation*

that *manipulate*

and *process* that data.

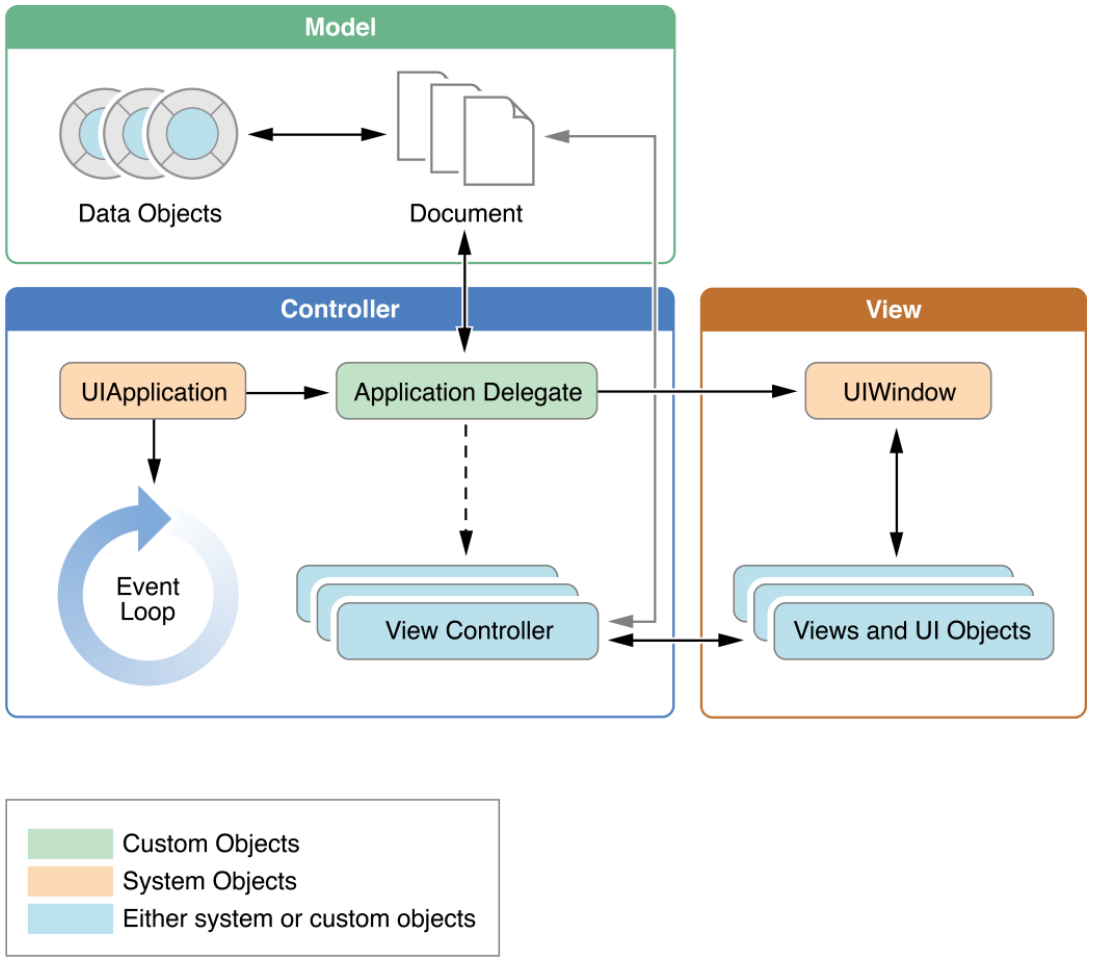


the *Controller* in M-V-C

the *Controller* is

"how is your *Model* *presented* to the user",
and it defines
"how does the user *interact* with it".

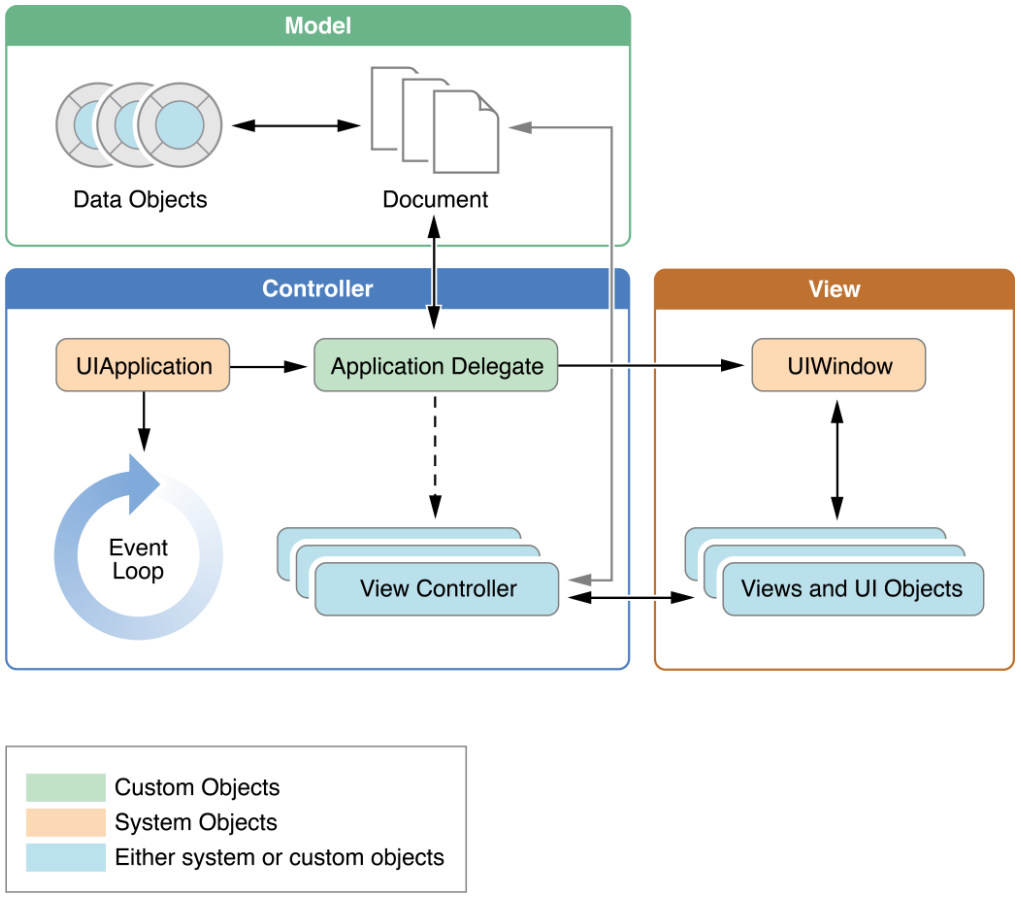
the *Controller* contains the *User Interface logic*.



the *Controller* in M-V-C

Controller objects are
conduits through which
view objects
learn about changes in
model objects
and vice versa.

Controller objects also
deal with: app *setup*,
coordinating *tasks*,
managing *lifecycles* of
other objects in the app.



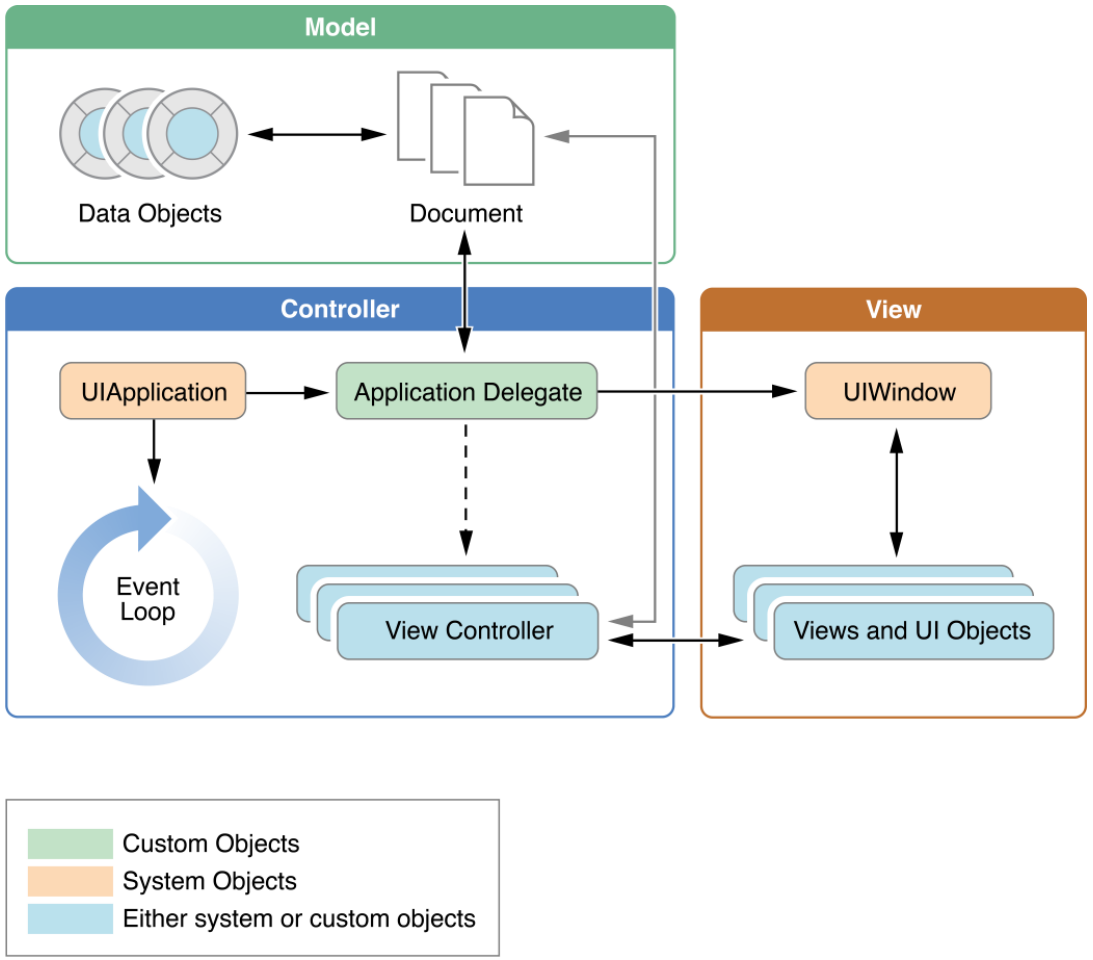
the *View* in Model-View-Controller

the *View* is:

"how the *Model* data is shown to the user".

The *View* is controlled...

...by the *Controller*.

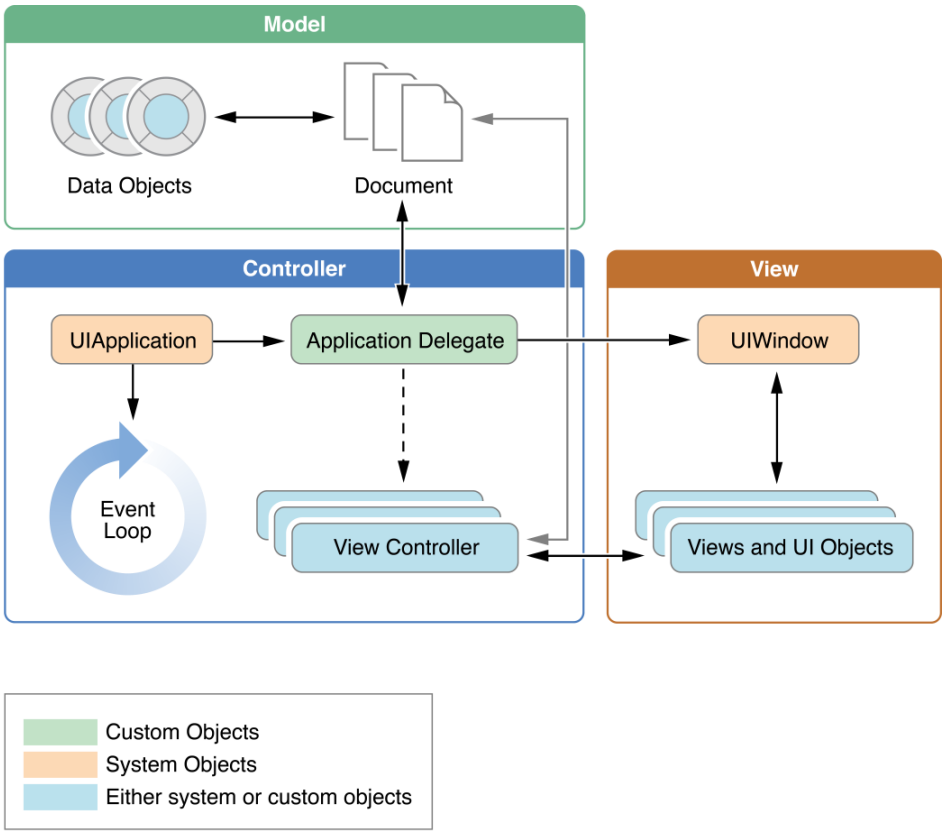


the *View* in Model-View-Controller

A *view object* is an object
(in an application)
that users can see.

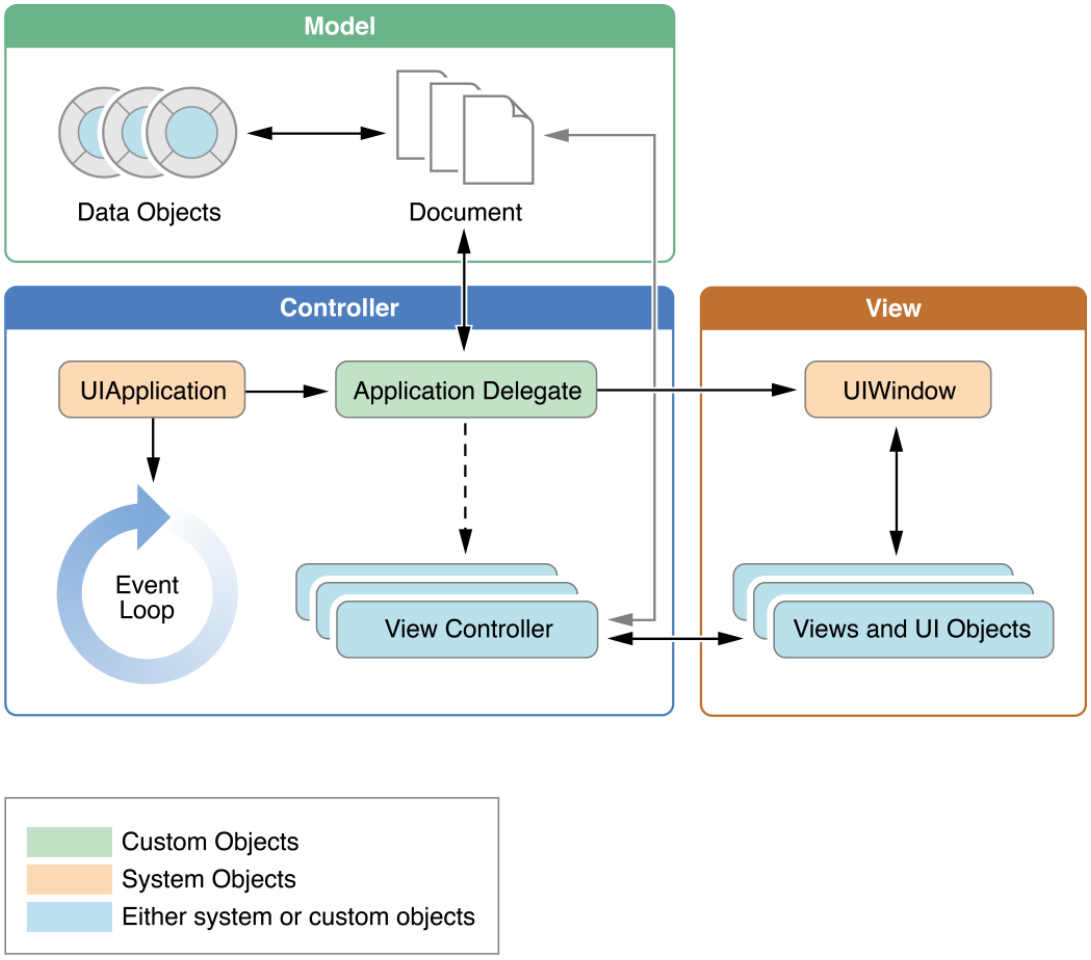
View objects know:

how to *draw themselves*
view objects also can
respond to user action.



communicating between M and V and C

communication
between M-V-C
i.e. the three blocks
needs to be managed
by design:
Controllers talk directly
to their *Model*.
Controllers talk directly
to their *View*.
Model and *View*
never communicate directly.

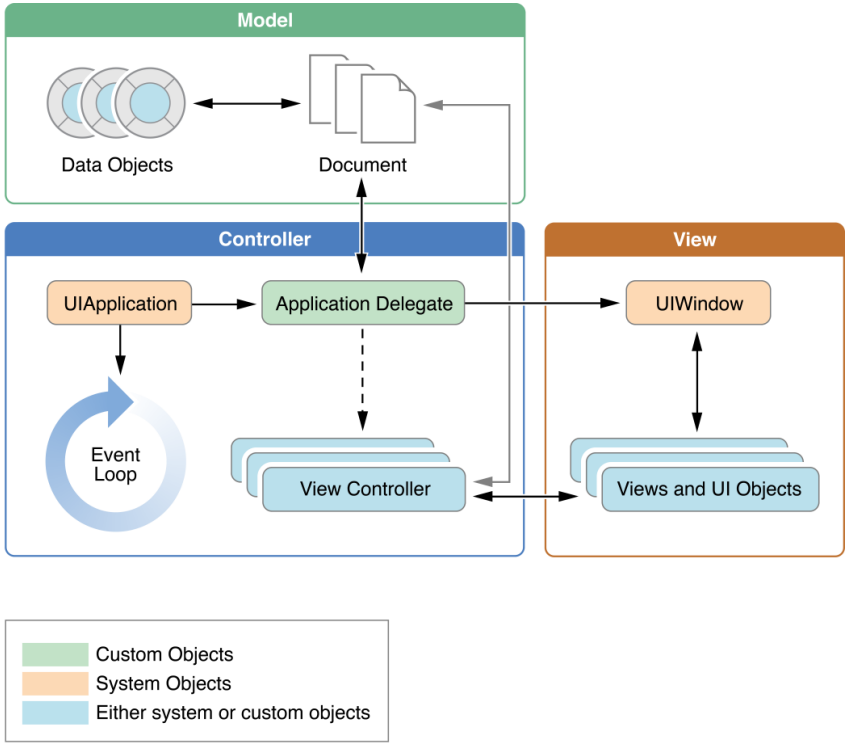


the essential objects in an iOS App

"From the time your app is launched by the user, to the time it exits, the UIKit framework manages much of the app's core behavior. "

At the heart of the app is the **UIApplication object**, which receives *events* from the system,

and dispatches them to your app's custom code for handling.



"The UIApplication class provides a centralized point of control and coordination for apps running on iOS. Every app must have exactly *one instance* of UIApplication (or a subclass of UIApplication)."

"When an app is launched, the UIApplicationMain function is called; among its other tasks, this function creates a *singleton* UIApplication *object*. Thereafter you access this object by invoking the sharedApplication class method."

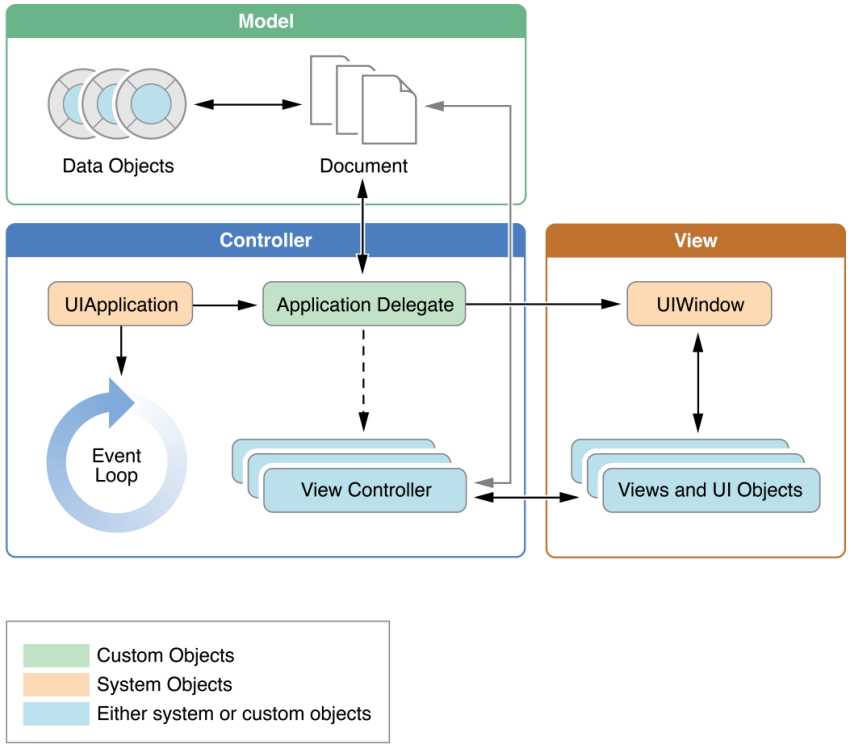
Other UIKit classes play a part in managing your app's behavior too: all of these classes have similar ways of calling your app's custom code to handle details.

the essential objects in an iOS App

The **application delegate** (or app delegate) is a custom object created at app launch time, usually by *UIApplicationMain*.

The primary job of this object is to handle state transitions within the app.

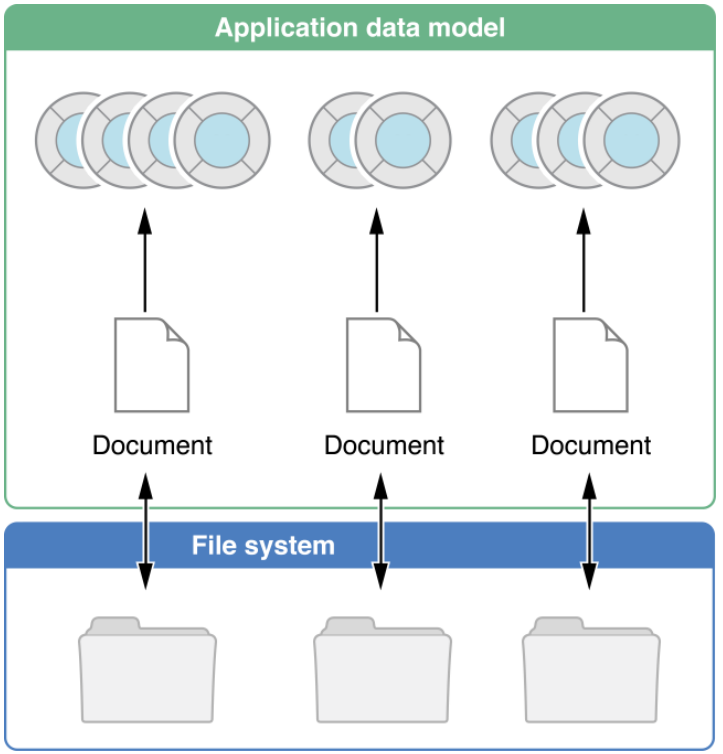
For example, this object is responsible for launch-time initialization and for handling transitions *to* and *from* the background.



the Model and *document objects*

Data model objects store your app's content, and are *specific* to your app.

E.g.: in a *banking app*, you may have a database containing financial transactions,
in a *painting app*, you may need an image object, or the sequence of drawing commands that led to the image creation, etc.



Apps can also use **document objects** (custom subclasses of UIDocument) to manage data model objects.

Document objects are not required, but offer a convenient way to group data that belongs in a single file or file package.

Completing our "*FlashCards*" iOS App

we'll now complete the FlashCards iOS app – so that it actually implements the "flash cards" functionality.

The app currently has:

1. two buttons
2. two methods that respond to buttons (but don't ***do*** anything)
3. two labels

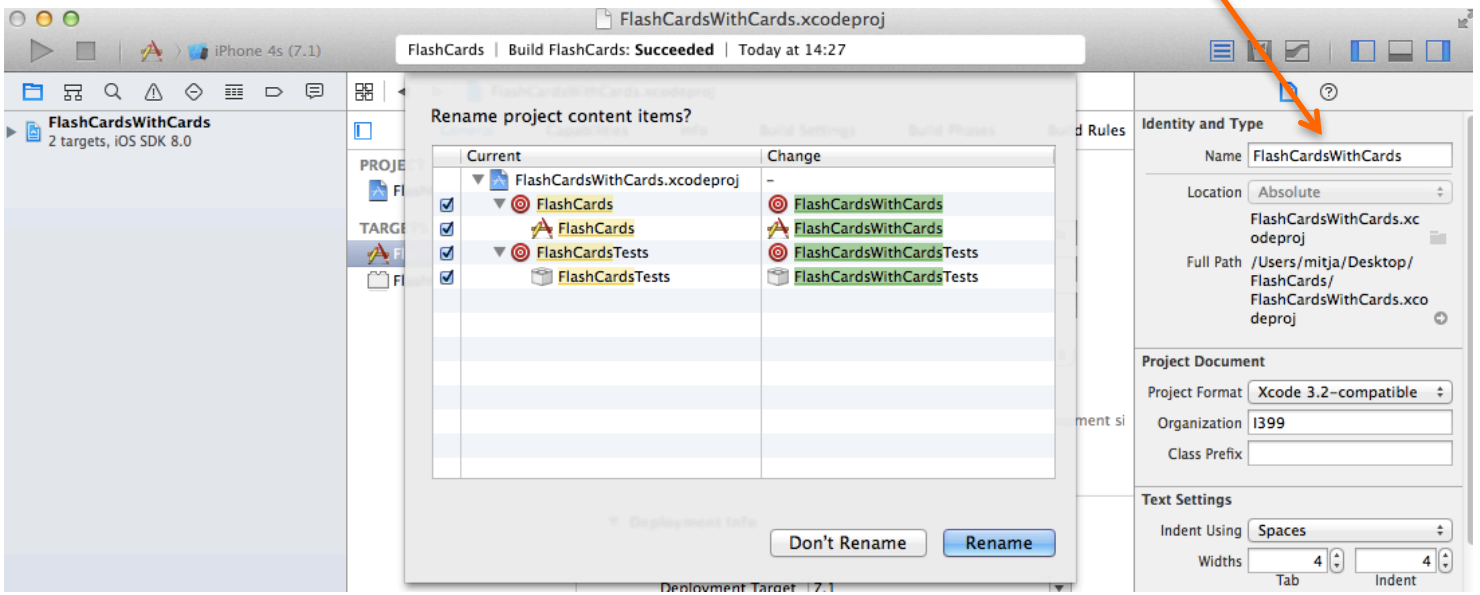
What we need to write now is:

4. the ***model*** code to generate content for the labels

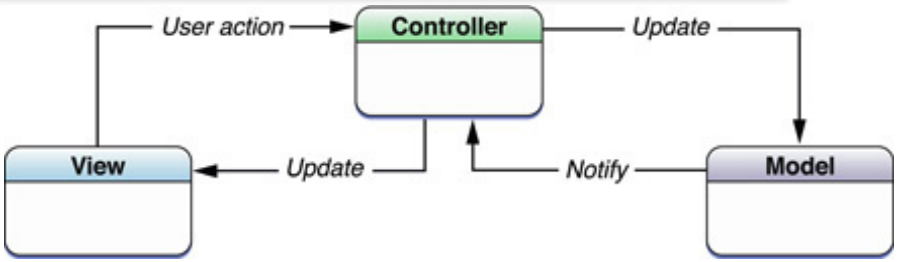
by the way… renaming the "*FlashCards*" project to "*FlashCardsWithModel*"? ***no thanks***.

(you ***could*** *rename* the App name in the Xcode Inspector:

- …select the "FlashCards" project (that's the blue "document" icon in the Xcode *project navigator*),
- …rename it using the *Identity and Type* tab in the Xcode inspector: you'll see this prompt:
- …note: doing this will rename the *project*, but it will *not* rename subdirectories, some filenames, etc. In general, ***this*** kind of renaming in Xcode ***is not recommended*** unless you know what's going on behind the scenes in the project directory)



the M-V-C design pattern



what we need to write now are the *Model Objects*:

- data and algorithms,
  i.e. data and methods/functions that work on that data.
- If we are modeling something that should behave like a real-life "*flash card*", we may name our class *FlashCardModel*.

(in our *FlashCard* app we already have:

Controller Objects:

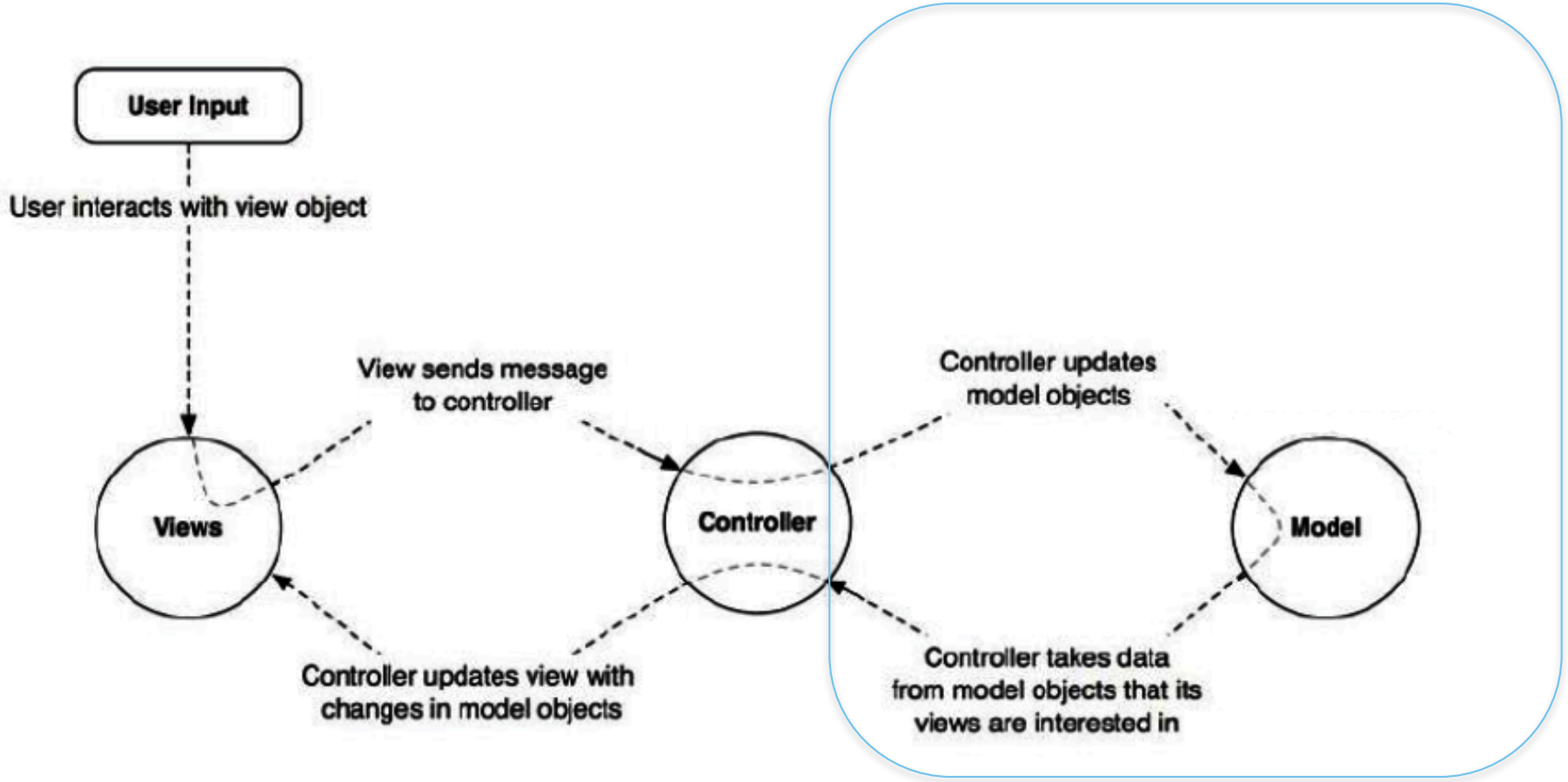
- are the intermediaries/controllers/managers
- they do the *configuring* of *view* objects, and the *synchronization* of *view* and *model* objects

View Objects:

- *directly visible* to the user
- in our case: text labels and buttons)

the M-V-C design for the *FlashCards* app

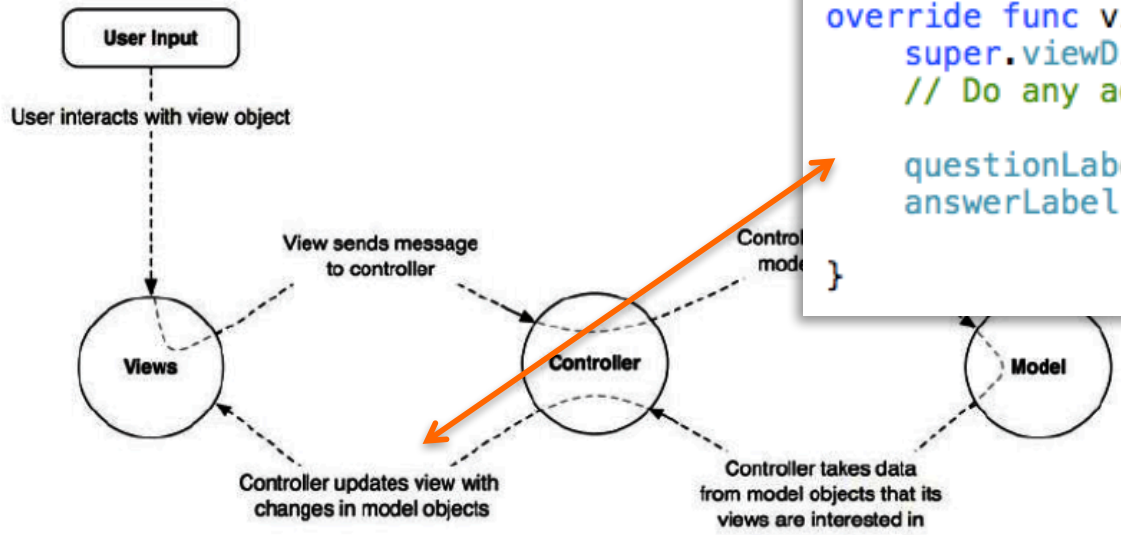
we now need to take care of the following parts:



the M-V-C design for the *FlashCards* app

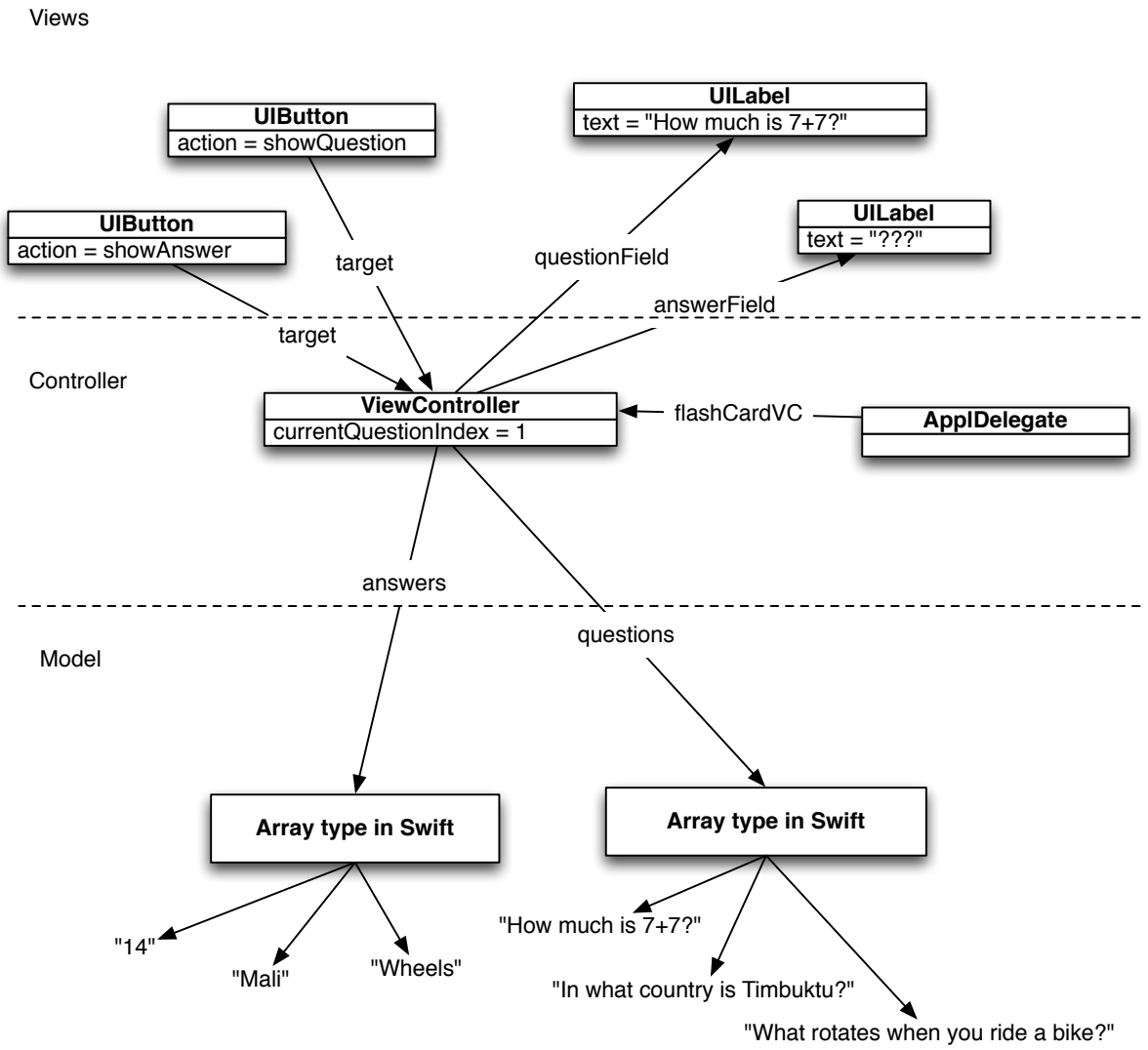
note about our *Controller Object:*

- in our ViewController, we haven't really implemented much functionality for the labels yet.
- we have tried something very simple, to make sure that ***Controller←→View*** connections are working.




```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view

    questionLabel.text = "what?"
    answerLabel.text = "yes."

}
```

the M-V-C design for the *FlashCards* app



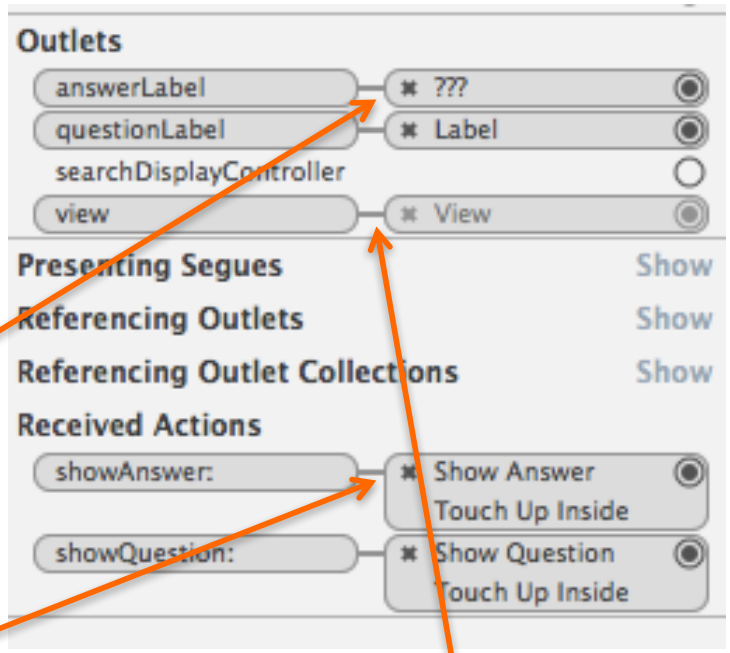
connections between View and Controller

there are now 5 connections

between the *ViewController*

and the *view objects* in Interface Builder:

a. the *answerLabel* and *questionLabel* properties in *ViewController.swift* are now connected to *label objects*.



b. the *ViewController class* contains two methods that are the targets for the two buttons: *showQuestion()* and *showAnswer()*.

c. (the project's template code and .storyboard already contained one connection: the background view of the application is already connected to the *view* property that's defined in the *UIViewController* class definition code that's part of the iOS SDK)

writing the Model Objects

let's create a new Swift file named FlashCardModel.swift:

press *[command ⌘]-N* on the keyboard

(or select the *Product→New→File...* menu)

then ***select***

iOS →

Source →

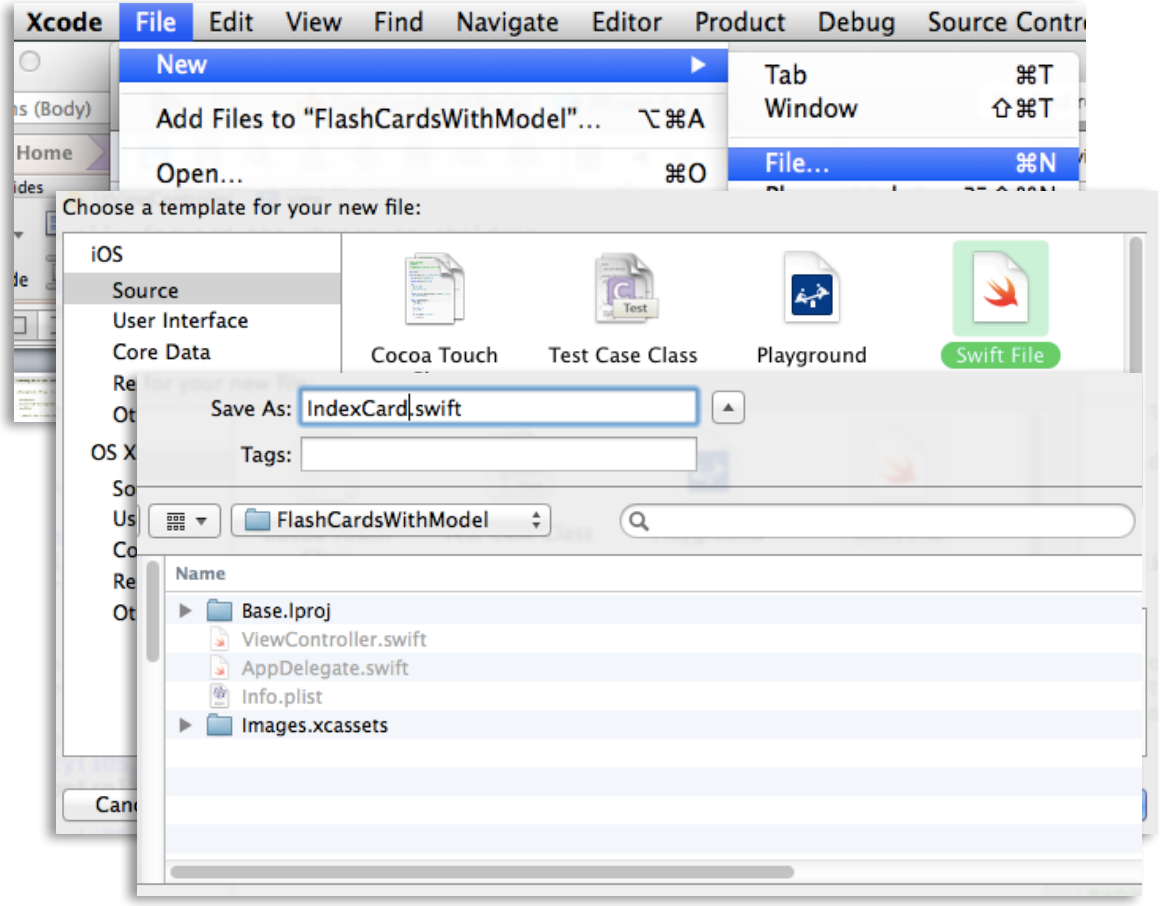
Swift File

...then ***click*** *Next...*

and ***save*** the file as

FlashCardModel.swift

in the same directory that contains your other *.swift* files in this project.



writing the Model Objects

now ***write***
some code
in the
FlashCardModel.swift
file:

```
class FlashCardModel {

    var questionsArray =
        [0: "What is your name?",
         1: "What is 42?",
         2: "What is the color of the sky?"]

    var answersArray =
        [0: "My name is not important.",
         1: "It's 6 times 7",
         2: "Kinda gray today."]

    var currentQuestionIndex = 0

    init () {
    }

    func getNextQuestion() -> String {
        currentQuestionIndex += 1
        if (currentQuestionIndex>=questionsArray.count) {
            currentQuestionIndex = 0
        }
        return questionsArray[currentQuestionIndex]!
    }

    func getAnswer() -> String {
        return answersArray[currentQuestionIndex]!
    }
}
```

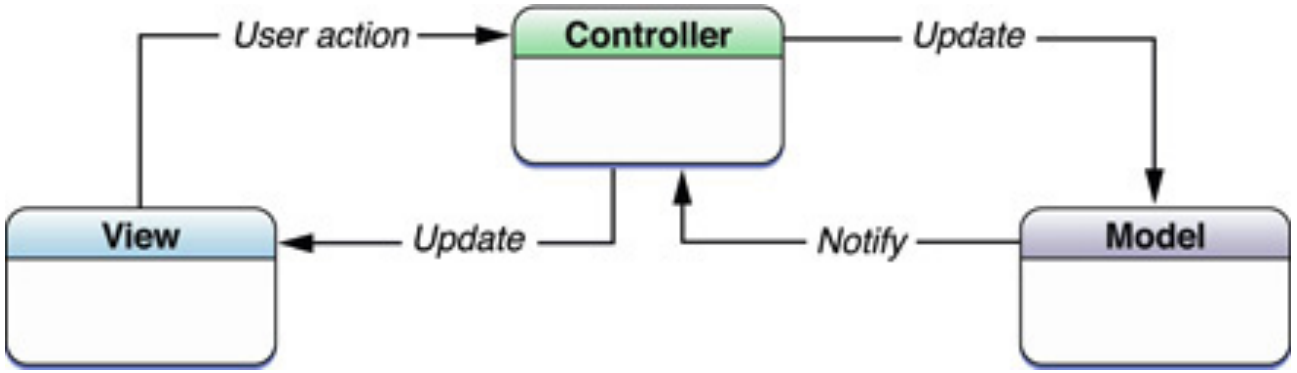
connecting *Model* and *Controller*

and ***add*** some code to the *ViewController.swift* file, i.e. ***create*** an instance of the new model we just defined in FlashCardModel.swift, then and ***write*** some code to implement the two *@IBAction* functions that we started writing earlier

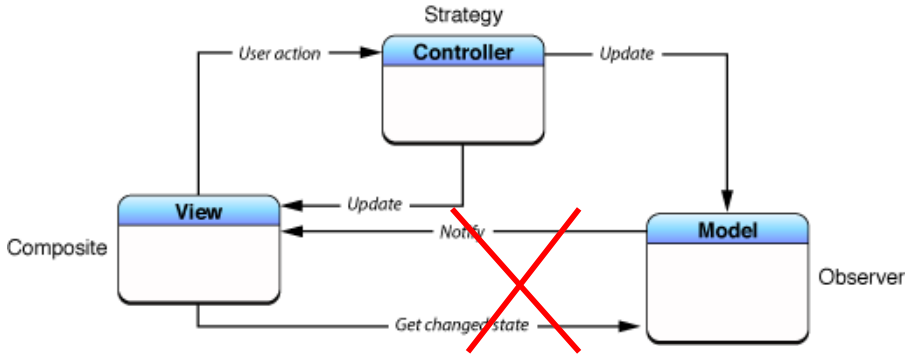
```
// add the model:
let myFlashCardModel = FlashCardModel()

// initially, no question has been asked:
var aQuestionIsAsked = false


// two IBAction methods, handling to "Button" events:

@IBAction func showQuestion(sender: AnyObject) {

    var lQuestion : String = myFlashCardModel.getNextQuestion()

    self.questionLabel.text = lQuestion
    self.answerLabel.text = "guess..."
}

@IBAction func showAnswer(sender: AnyObject) {
    var lAnswer : String = myFlashCardModel.getAnswer()

    self.answerLabel.text = lAnswer

}
```

MVC (Model-View-Controller) in iOS

- when designing an iOS app, we use the MVC design pattern thus:

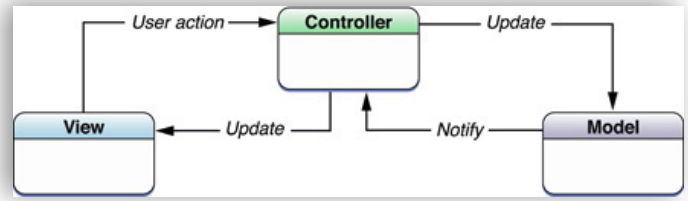


(note: there are other ways to define and implement the *traditional* MVC design pattern. However, the View-Model connections are *not* part of how MVC is used in designing iOS apps)

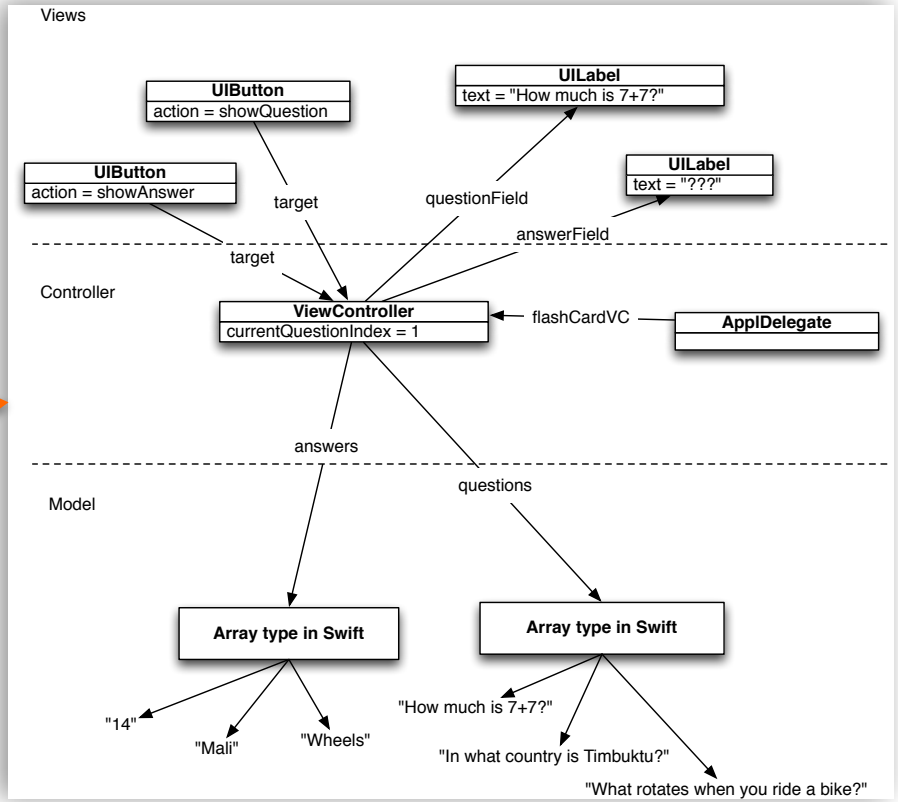


MVC in iOS

when designing an iOS app, we use the MVC design pattern thus:



for example, in the *FlashCards* app seen in Lectures 3 and 4, the MVC pattern is used to describe the Model (questions and answers) the View (buttons and labels) and the Controller (the app's behavior)



Turn in B481 Lab 06

1. When you're done experimenting with today's OpenGL Swift app project:
    1. select the Xcode menu *Product* → *Clean*
    2. quit Xcode, then rename your project folder to "foldername*yourusername*"
    3. place your "foldername*yourusername*" folder in your lab06 local github repository copy, then *add, commit & push*

(where *foldername* is the name of your Xcode project, not the word "foldername", and *yourusername* is your IU username, not the word "yourusername")