Intro, Swift, Xcode

Welcome

Hello! Welcome to AppDev's iOS Training Course. Throughout this course, you will learn all the fundamentals of iOS programming and learn how to build your own full-fledged iOS applications! Each lecture will be accompanied by a handout (like this) summarizing the main concepts from that lecture. This is the first handout and we will be covering Swift and Xcode.

What is Xcode?

Xcode is the main integrated development environment (IDE) that iOS developers use to develop their applications. In simpler terms, Xcode is just the application that we use to write iOS code (similar to how you might use Atom, Sublime, vim, etc.). The reason all iOS developers use Xcode (instead of other IDE's) is that other than being a place for us to write iOS code, Xcode allows us to run our code and build our application on a simulator. This is important because otherwise we would have no way to testing if our code actually does what it wants!

Creating a new Xcode project

Lets now begin by creating our first, new Xcode project! Here are the steps:
STEP 1. Launch the Xcode application and press the **Create a new Xcode project** option

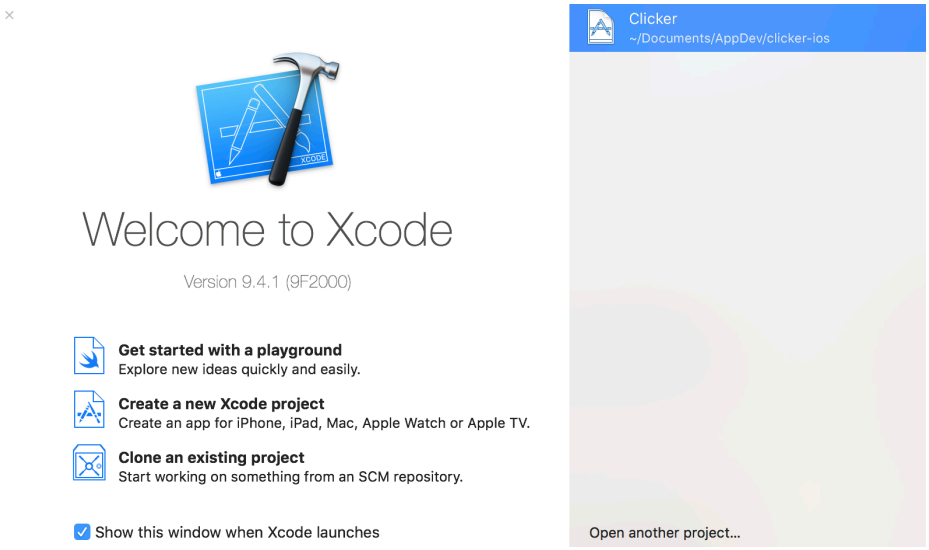


Figure: Xcode Launch Screen

STEP 2. You will now be prompted to choose the type of application you want to create. We will **always** be creating a **Single View App**

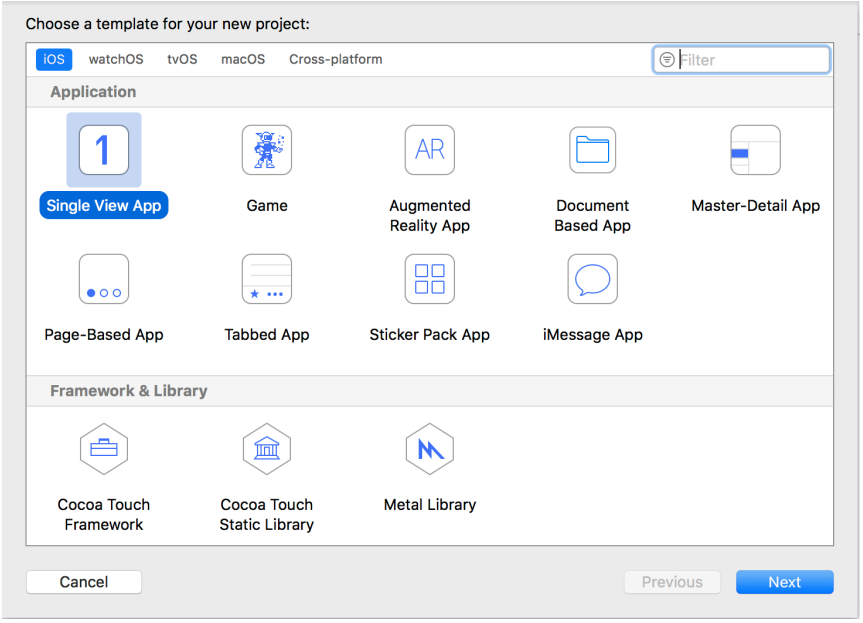


Figure: Create new application screen

STEP 3. Next, give a **name** for your new application, click **Next**, and then save your application in the directory that you want to save it in.

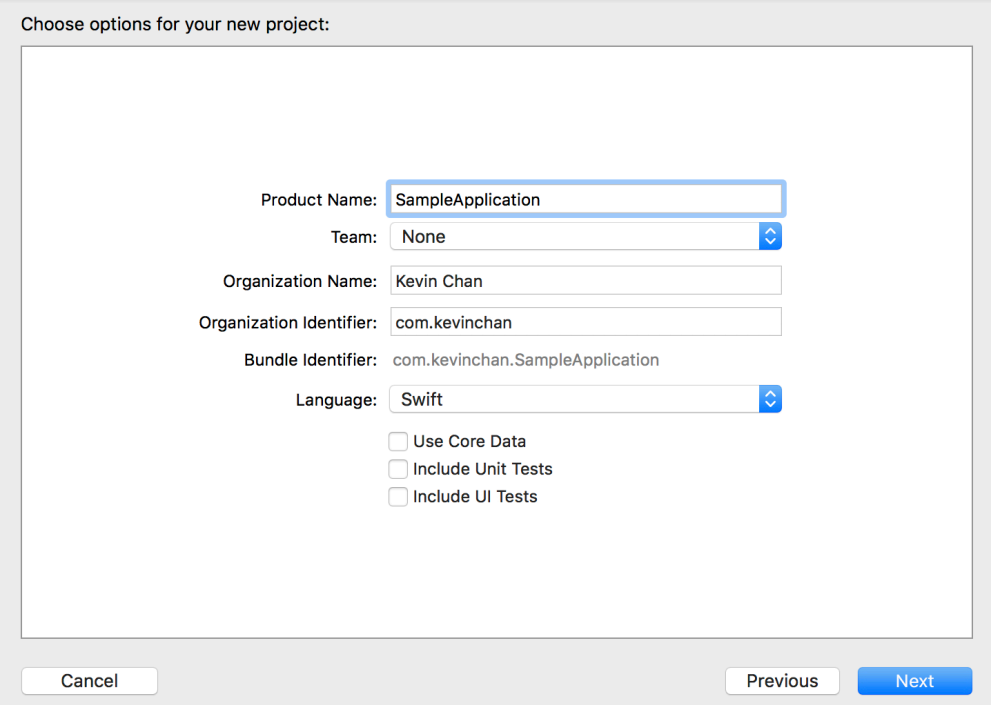


Figure: Naming new project screen

Navigating through Xcode

Now that you've created a new Xcode project, you can start working! Before you start, however, we are going to go over some small tips and tricks for using and navigating Xcode. Firstly, on the left side of the project, you should see the **file hierarchy** for the project. This allows you see how the files that your project contains as well as navigate or change any of them by simply clicking on the file that you want to change.

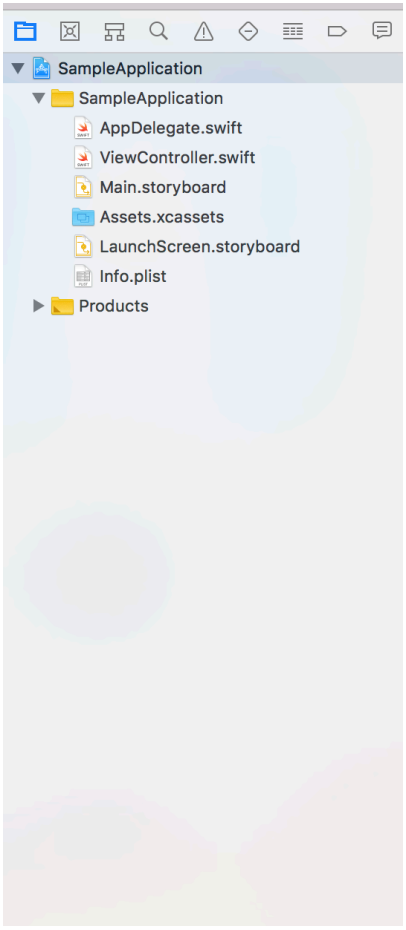


Figure: Project directory display, located on the left side of Xcode

Now, if you look to the top-left of the project, you should see a **play button** next to another button that should say something like **SampleApplication > iPhoneX**. The play button is the button that you will want to click when you have written some code and want to see how your application looks. Clicking on the play button (**or clicking CMD + R**) will run your application on a simulator device. To change the type of device to build you application on,

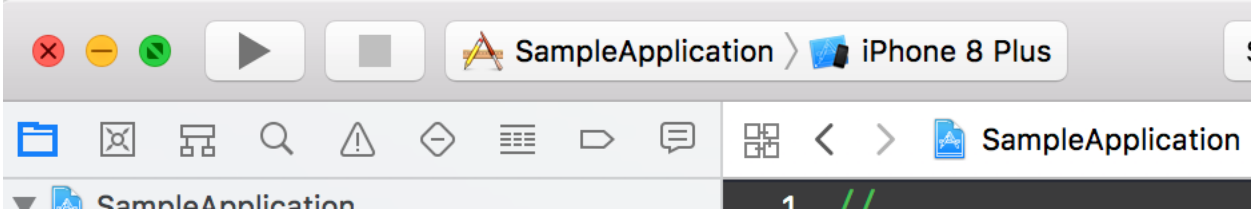


Figure: Run application button, located on top left of Xcode

click on the button next to the Play Button that says **iPhone 8 Plus** (it may something different on your application, i.e. iPhoneX). This should then bring down a dropdown menu of all the possible devices that you can run your application on. Simply click on the device that you want and then you are done!

Now, lets say that you want to **create a new Swift file**. One way to do is to do ***File -> New -> File..*** . The other way to do this is just to run **CMD + N**. Using either way, you should now see a window asking you what type of file you want to create. If you just want to create an empty Swift file, simply select the **Swift File** option. On the other hand, if you know you want to create a certain type of file (i.e. a file to hold a UIViewController, UITableView, UICollectionViewCell, etc) and want to have the boilerplate code already in the file when you create it, select the **Cocoa Touch Class** option.

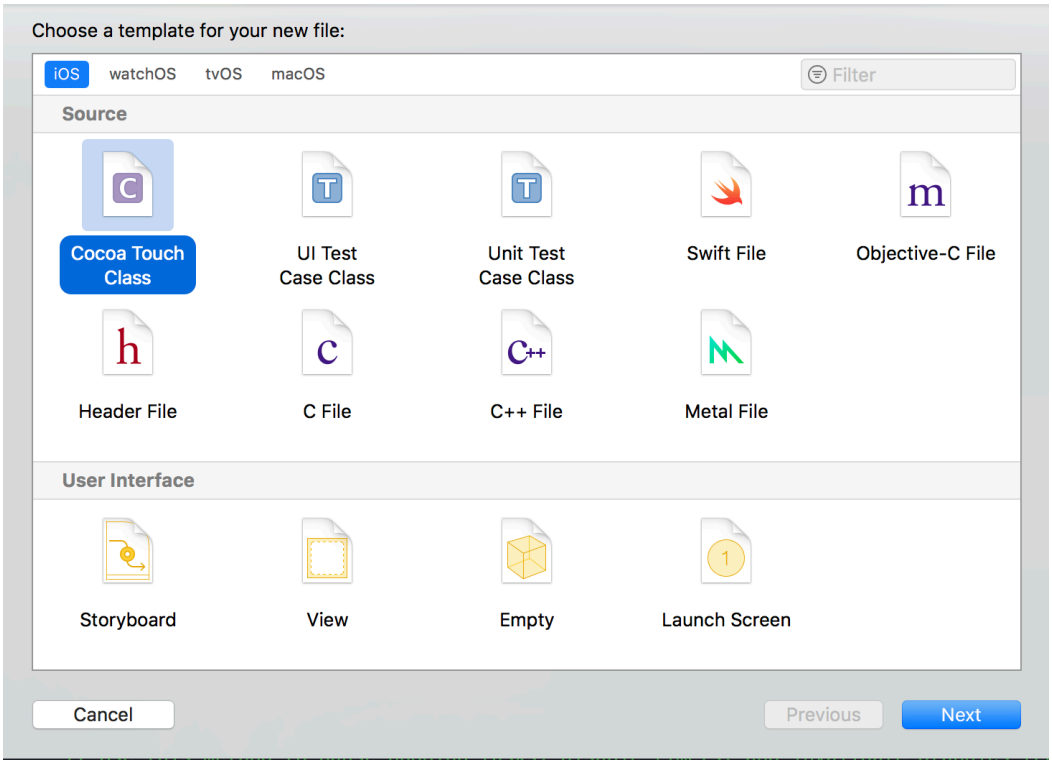


Figure: New file screen

After clicking on the **Cocoa Touch Class** option, you should see a window like the one below. Next to the **Class** textfield, put in the name of your new file. Then, in the **Subclass of:** dropdown, find and select the type of file that you want to create. In the screenshot below, I want to create a new **UIViewController** file. Lastly, click **Next** and then you are done!

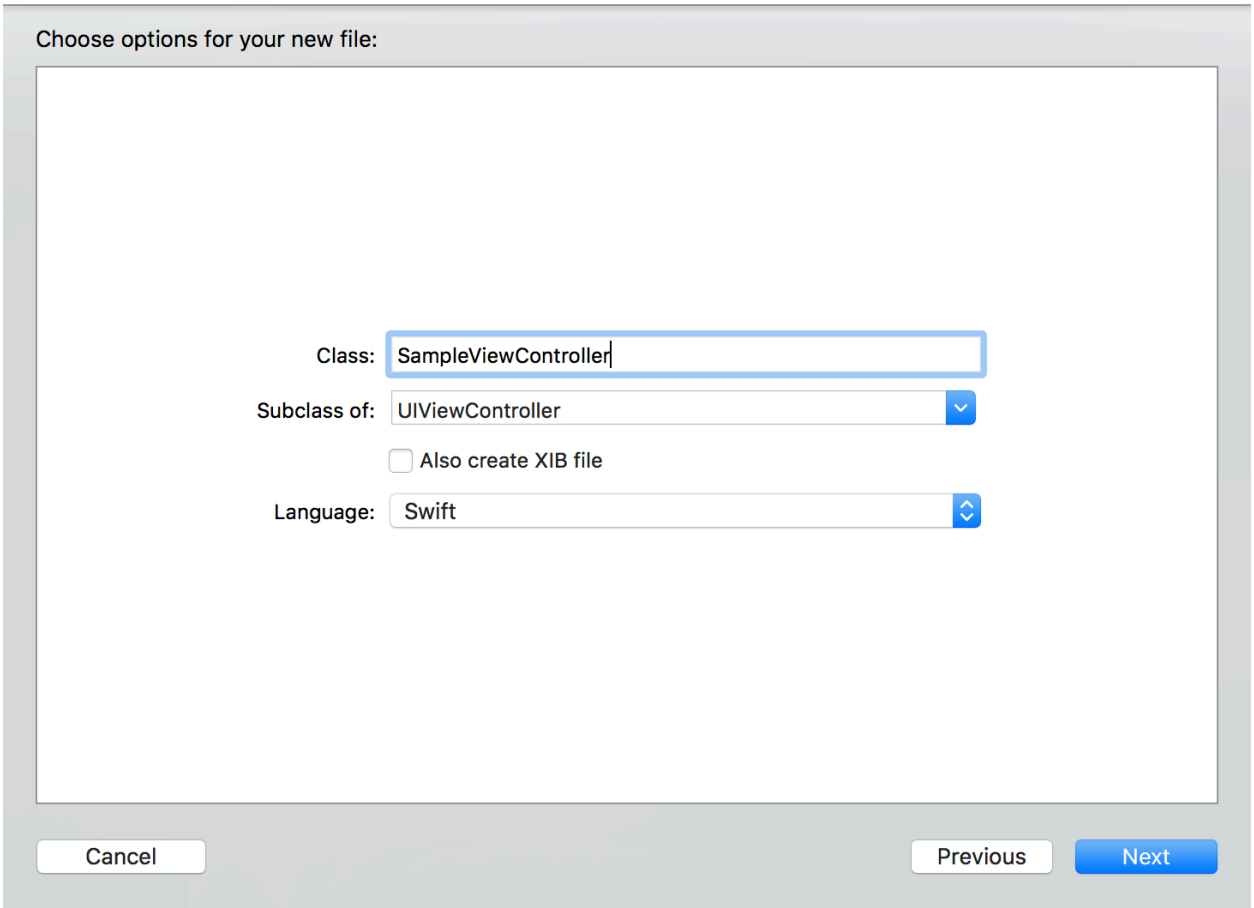


Figure: Naming new CocoaTouchClass screen

What is Swift?

Swift is Apple's newest programming language for developing iOS applications. One other popular programming language that you can use is Objective-C. However, in this course we will be using Swift. Thus, you will be writing Swift code inside your Xcode project and then

Basics of Swift

We will now cover some fundamentals of the Swift programming language. One thing to note is that while we will not be able to cover every single aspect of Swift in this handout, any aspect not covered can be found through a simple Google search.

I. Variables and Constants

```
var x = 5
var y: Int = 5
y = 6
let z = 7
```

Like other programming languages, we can declare a **variable** in Swift using the syntax **var [variable name] = [value]**. In the example above, one thing we see is that Swift has **type-inferencing**. What this means is that Swift is able to infer the type of your variable based on the value that you assign to it. For example, by assigning **5** to **x**, Xcode knows that **variable x** is of type **Int** because **5** is of type **Int**. This means that when declaring a new variable and assign a value to it, it is not required to also declare the type of the variable. If you wanted to, however, you could do so using the syntax **var [variable name]: [type] = [value]**.

In addition to variables, we can also declare a **constant**. One important distinction to make between variables and constants is that after you assign a value to a constant, you cannot change the constant's value again in the future (you can change a variable's value). The syntax to declare a constant is **let [variable name] = [value]**. Similar to variables, it is not required to declare the type of the constant.

II. Classes and Structs

```
// class name
class Song {

    // field declarations
    var name: String
    var author: String

    // methods
    init(name: String, author: String) {
        self.name = name
        self.author = author
    }
}
```

In Swift, we can also declare **classes** and **structs**. Every class contains several different parts: a **class name**, **fields**, and **methods**. Fields are just **variables/constants** that belong to that class. In our sample class above, for example, our **Song** class contains a **name** field of type String and an **author** field of type String. Similarly, **methods** are just functions that belong to that class. In our example above, we have an **init** method which is the method

that gets called when you create an instance of that class.

A **struct** has the same exact syntax as a class (except you replace **class** with **struct**). However, a core difference between structs and classes is that structs are **value types** whereas classes are **reference types**. What does this mean? A **value type** "is a type whose value is *copied* when it's assigned to a variable or constant, or when it's passed to a function." On the other hand, **reference types** "are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used." To see what this means, let us look at the following example.

```
var songA = Song(name: "Lucky You", author: "Eminem")
var songB = songA
songB.name = "Walk on Water"
print(songA.name) // prints "Walk on Water"
print(songB.name) // prints "Walk on Water"
```

Using the **Song** class that I declared before, I create an object of type **Song** and assign variable **songA** to that Song object. Next, I create another variable **songB** and assign its value to be **songA**. Then, I set the **name** field of **songB** to be "Walk on Water" (previously it was "Lucky You"). Lastly, I **print** the value of songA's name field and the value of songB's name field. In both cases, "Walk on Water" gets printed.

Why does this happen? Well, because Song is a **class** and we know that classes are **reference types**, when we created our new Song object and assigned that to songA, what really happened is that songA's value was a **pointer** to that Song object. Then, when we assigned songA to songB, what we were doing was assigning that exact pointer to songB. Thus, when I changed the value of songB's name field, the value of songA's name field changed as well because they were both pointing to the exact same Song object.

```
struct Song { ... }
var songA = Song(name: "Lucky You", author: "Eminem")
var songB = songA
songB.name = "Walk on Water"
print(songA.name) // prints "Lucky You"
print(songB.name) // prints "Walk on Water"
```

If instead Song was a **struct**, we would see something else happen. What exactly? Well, looking at the example above, we see that printing out songA's name prints out "Lucky You" and printing out songB's name prints out "Walk on Water" (unlike the previous case where they both printed out "Walk on Water"). Why did this happen? Well, remember that **structs** are **value types**. Thus, when songA was assigned to songB, a **copy** of the Song object that

songA was pointing to was passed to songB (unlike the previous case where the pointer to the exact same Song object was passed to songB). Thus, when we changed songB's name field, songA's name field was not changed because songA and songB are pointing to two different Song objects.

III. Optionals

In Swift, we also have a type that you may not often see in other programming languages, **optionals**. Optional types just mean that a variable may hold some value or it may not (i.e. it is **nil**). Let us take a look at an example

```
var a: Int? = 5
var b: Int = a // error: value of optional type 'Int?'
```

Here, we have declared a variable **a** whose type is **optional Int** and have assigned it a value of 5. We also could have set the value of **a** to be **nil** because **a** is an *optional* Int which means that it can be nil. However, if we then declare another variable **b** of type **Int** (not optional) and then we try to assign **a** to **b**, an error message saying "error: value of optional type 'Int?'". Why did this error show up? Well, this is because unlike **a,** variable **b** is an Int which means that it **must** hold some value. However, since **a** is of type optional Int, we are not guaranteed that **a** currently holds some value.

How do we solve this issue? In other words, how can we assign an optional variable to be the value of another variable? The answer is **unwrapping**. If we have an optional variable, we can **unwrap** it to see if the variable actually holds a value, and if it does to store that value in another variable. There are two main ways to unwrap a variable: using an **if-let** statement and using an **guard-let** statement.

```
var a: Int? = 5
var b: Int
if let unwrappedA = a {
    // this block gets run if variable a holds a value
    b = unwrappedA // b is now equal to 5
} else {
    // this block gets run if variable a was nil
    b = 0
}
```

First, we will take a look at unwrapping using an **if-let** statement. The syntax for an if-let statement is **if let [unwrapped constant name] = [optional variable] { … } else { … }**. In our example above, we take optional variable a and then we try to unwrap it. If **a** actually does hold some value, then that value gets assigned to the constant **unwrappedA** and then we enter the first block of code. If, however, **a** was **nil**, the **else** block would get run.

```
var a: Int? = 5
var b: Int
guard let unwrappedA = a else {
    // this block gets run if variable a was nil
    // in here, you should return or throw an error
    return
}
// this gets run if variable a holds some value
b = unwrappedA
```

Now, lets see how we could do the same thing using a **guard-let** statement. The syntax for a guard-let statement is **guard let [unwrapped constant name] = [optional variable] else {...}**. In our example above, we take optional variable a and then we try to unwrap it. If **a** actually does hold some value, then that value gets assigned to the constant **unwrappedA** and then we skip the else block and run the code that comes after. If, however, **a** was **nil**, the else block will get run. One thing to take note when using guard-let statements, however, is that in the else-block, you should either **return** or **throw** some error at the end of it.

AutoLayout and UIKit

What is UIKit?

UIKit is a framework which provides us as iOS developers with the basic components that we need to use in order to create an iOS application. To list just some of these components:

- UIButton, UILabel, UITextField, UIImageView, UITableView, UICollectionView, UIView
- UIColor, UIFont, UIGestureRecognizer
- UIViewController

The first row of items are just examples of subclasses of **UIView**. As you can probably guess from the name, these are all views that you can create and see when you run your application. The second row of items are classes that help you to work with these views. For example, a **UILabel** has a **textColor** property which is of type **UIColor** and a **font** property which is of type **UIFont**. The last row is a **UIViewController** (we will discuss MVC next lecture).

```
import UIKit

class ViewController: UIViewController { ... }
```

Now, how do we get access to these components and actually use them in our application? Well, you simply have to include the line **import UIKit** on the top of the file that you need to use it in. Later on in the course, we will be using other types of frameworks and to use them, we simply do the same thing and include the **import [Name of Framework]** line at the top of our file to use them.

How do you make views show up on screen?

There are multiple ways to layout views to a screen in iOS development – some examples include **frame based, storyboards,** and **programmatic AutoLayout**. Frame based layout involves specifying an **origin, width, and height** for each of your views. However, as you can probably imagine, using frame based layout can be troublesome. Why? Well, iPhone devices come in all different sizes (some may be wider, some may be taller) and placing views using

using coordinates when your x-y coordinate system changes depending on the device can lead you to do a lot of unnecessary math. Storyboard is a method of creating your views using an interface builder. The interface builder allows you create and layout your views simply by dragging the views onto it (and thus involves no code). While this may be useful for developing small, quick demo applications, developing a large scale application using Storyboards is extremely disastrous because there is no form of reusability and making changes to a complicated application can take a lot of time. Not to mention, Storyboards with version control systems are horrible. The last method, and the method you will be learning in this course is programmatic AutoLayout.

What is AutoLayout?

AutoLayout is a way to layout views in relation to one another. AutoLayout works with all device sizes and orientations without worrying about the coordinates of where everything is on screen. Thus, it is a more reliable way of laying out our views and also involves less intensive thinking, helping us to avoid errors more often.

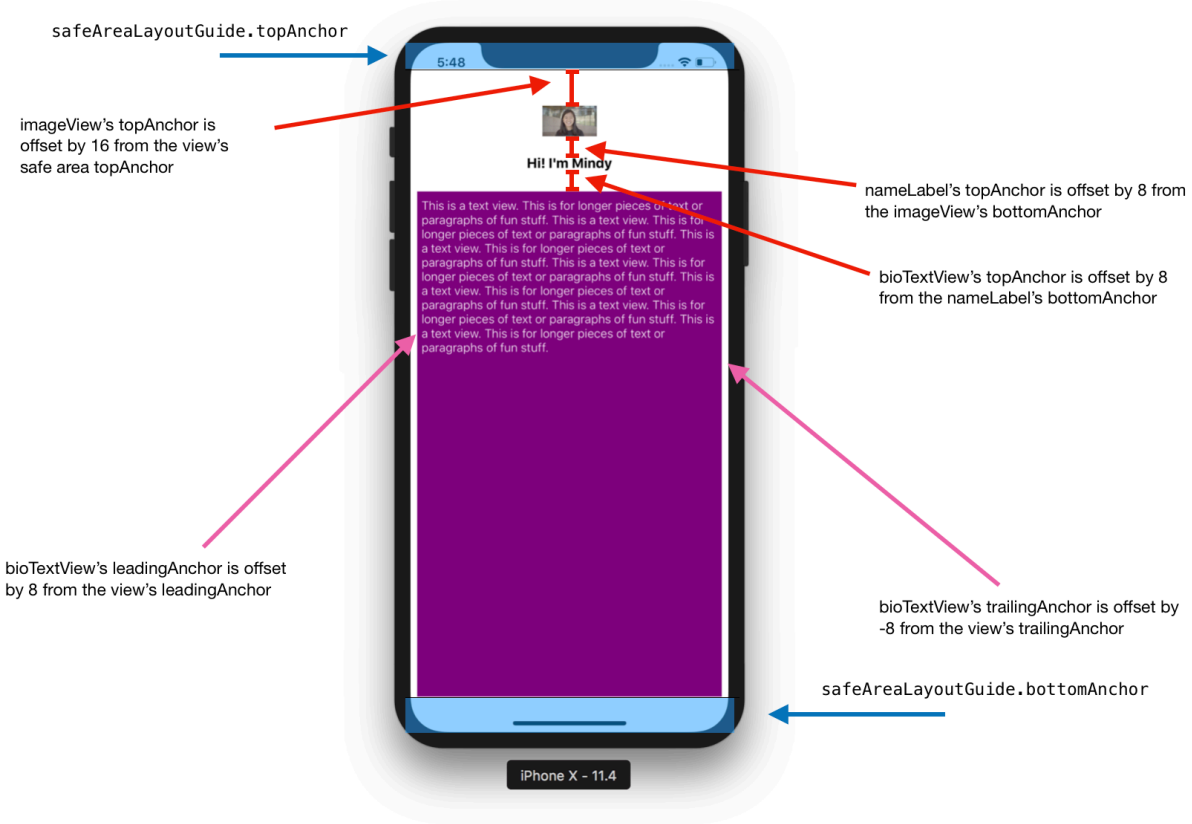
AutoLayout Terminology

Every subview of **UIView** has these four properties: **topAnchor, leadingAnchor, bottomAnchor,** and **trailingAnchor**. As the names imply, topAnchor refers to the view's top edge, leadingAnchor refers to the view's left edge, bottomAnchor refers to the view's bottom edge, and trailingAnchor refers to the view's right edge. An **IMPORTANT** note to always remember is that in order for us to use these anchors to create constraints (layout our views), we must remember to set the view's **translatesAutoresizingMaskIntoConstraints** property to be false before setting these anchors. How do we use these anchors to layout our views? Well, lets take an example. Assume we have two labels, labelA and labelB, and we want to make it so that labelB is 20 pixels to the right of labelA. This is how we would do that:

```
    var labelA: UILabel!
    var labelB: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        labelA = UILabel()
        labelA.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(labelA)

        labelB = UILabel()
        labelB.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(labelB)

        // We have to activate our constraints in order to
        NSLayoutConstraint.activate([
            // Here, we can insert all the constraints that we want
activated
            labelB.leadingAnchor.constraint(equalTo:
labelA.trailingAnchor, constant: 20)
            ])
    }
```

What exactly is happening here? Lets break it down. Inside our UIViewController class, we declare our labels as variables of type UILabel. Then, inside our **viewDidLoad** function (the function that gets called when our ViewController is displayed on the screen), we initialize our two labels (**remembering** to set translatesAutoresizingMaskIntoConstraints to false for both of our labels), and then add the labels as subviews to the ViewController's view. Next, we have to create our **constraints** and **activate** them (a constraint is just some rule that we apply to one of the four anchors of a view). An easy way for us to use this is to call the **NSLayoutConstraint.activate([NSLayoutConstraint])** method. This method takes in an array of constraints and then automatically activates them for us. In our example above, we are just going to activate one constraint. If we want labelB to be 20 pixels to the right of labelA, that just means we want labelB's left edge to be 20 pixels to the right of labelA's right edge. However, we know that this is equivalent to saying we want labelB's **leadingAnchor** to be 20 pixels to the right of labelA's **trailingAnchor**. The syntax for doing this is **labelB.leadingAnchor.constraint(equalTo: labelA.trailingAnchor, constant: 20)**. Breaking this down, we take labelB's leadingAnchor property and call the anchor's **constraint(equalTo: NSLayoutAnchor, constant: CGFloat)** method to constrain it to be equal to labelA's leadingAnchor, offset by 20 pixels.

As a final exercise to see if you fully understand anchors, try thinking about what constraints you would create this screen



Navigation, MVC, Delegation

How do we create a multi-view application?

When building an iOS application, we need someway to move between different views. Otherwise, we would just be staring at the same screen the whole time and that would not provide great user experience. So, how do we create a multi-view application? The answer is **navigation** (Note: When we talk about navigation, we are talking about transitioning from one *view controller* to another). Specifically, there are two forms of navigation, **pushing/popping** and **presenting/dismissing**.

Pushing/Popping ViewControllers

Before we can discuss pushing/popping view controllers, we need to make sure we understand another concept, **UINavigationController**. From Apple's documentation, "*A navigation controller is a container view controller that manages one or more child view controllers in a navigation interface. In this type of interface, only one child view controller is visible at a time*." In other words, a UINavigationController can be thought of as a stack that manages multiple view controllers. However, only one is ever present on the screen at a time.

Using navigation controllers, we can accomplish the first form of navigation, **pushing/popping**. What does this form of navigation look like? Lets take a look:

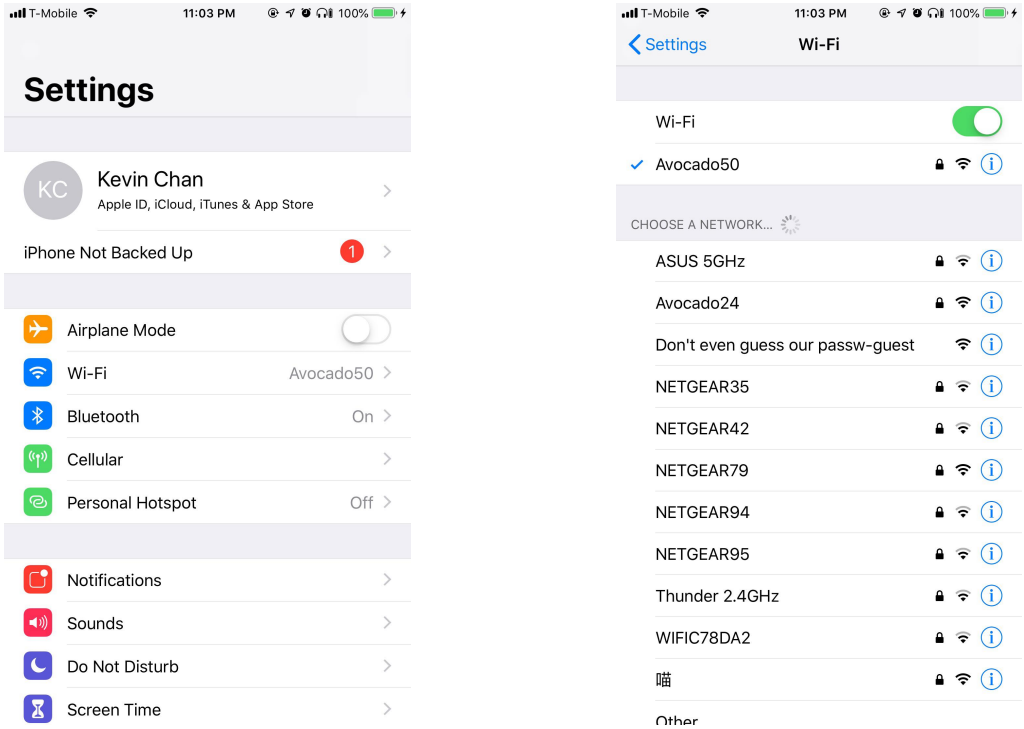


Figure: Display of push navigation (Settings app)

If you go into your **Settings** application and then click on **Wi-Fi**, the transition from the Settings screen to the Wi-Fi screen is the **pushing** navigation. Thinking about this in terms of the navigation controller's stack of view controllers, the act of pushing is equivalent to pushing that view controller onto the top of the stack so that the view controller is now the only visible view controller on your screen. Now that you are on the Wi-Fi screen, if you were to press the **< Settings** button on the top left, you will be brought back to the Settings screen. This transition is the **popping** navigation. Similarly, the act of popping is equivalent to popping the top view controller from the stack so that another view controller is visible.

```
self.navigationController?.pushViewController(viewController: UIViewController,
animated: Bool)
self.navigationController?.popViewController(animated: Bool)
```

Presenting/Dismissing ViewControllers

Unlike pushing/popping, in order to present/dismiss a view controller, we do not need to have a UINavigationController, and thus no stack. When we **present** a view controller, all we are doing is presenting a view controller directly on top of the current view controller. What does this look like? Lets take a look:

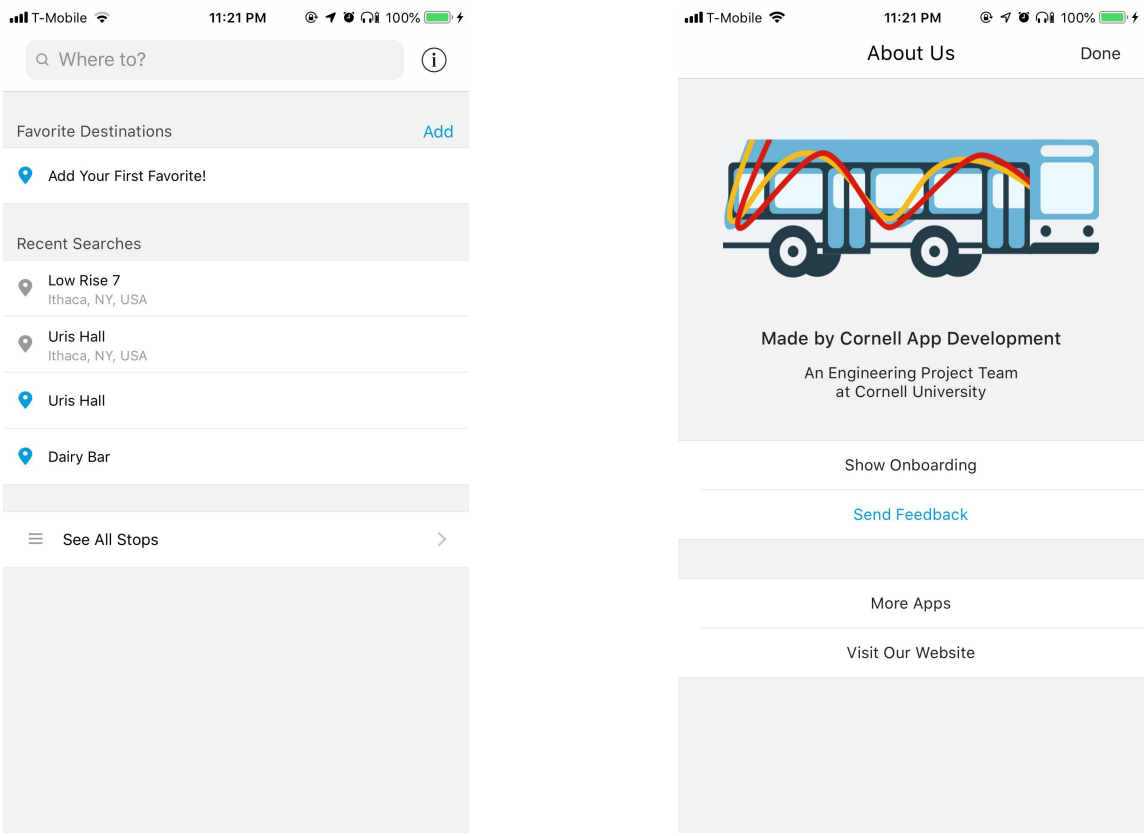


Figure: Display of presenting navigation (Ithaca Transit app)

If you open the Ithaca Transit application on your phone and then click the **(i)** icon on the top right, the **About Us** view controller gradually comes up from the bottom of the screen. This transition is the presenting navigation. Similarly, if you press the **Done** button on the top right, you will see that the **About Us** screen will gradually go towards the bottom of the screen until it disappears. This transition is the dismissing navigation.

```
self.present(viewControllerToPresent: UIViewController, animated: Bool,
completion: (() -> Void)?)
self.dismiss(animated: Bool, completion: (() -> Void)?)
```

MVC

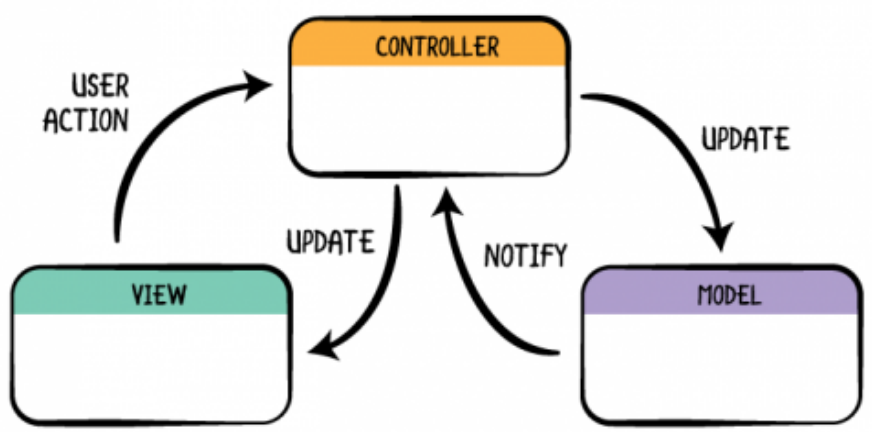


Figure: Model-View-Controller diagram, showing the relationship of each

What is **MVC** and why is it important? MVC is a type of software design pattern. A software design pattern is just set of rules regarding code architecture that we follow when writing code. MVC stands for **Model-View-Controller** and it is the software design pattern that we will be using when developing iOS applications. Before we explain the idea between MVC, let us break down the individual components.

Model
Models refer to objects that represent the data for our application. For example, if we were making a Music Player application, a model object that we would have is a **Song**. Or if we were making a Social Network application, a model object that we would have is a **User**. These model objects usually experience change as the user interacts with our application.

View
Views refer to the actual UIView components that we use to build our application. For example, this includes UIButton, UILabel, UITextField, etc. Essentially, views are just the UI part of our application

Controller
The controller's main goal is to act as an interface between the model and view. The controller's job is to process any server responses that the client receives and then update the model objects. This change in the model objects usually also necessitate a change in the views of our application which the controller also handles.

Putting this all together, MVC can be summarized into a couple of steps. Any user interaction with our view gets notified to the controller which in turn can lead to an update of our views or model objects. Similarly, any change in our model objects (perhaps through a server request/response) gets notified to the controller which can in turn lead to an update of our views or model objects. The most **important** feature to keep in mind about MVC is that the model and view components should be independent of each other. Both of these should communicate with each other only through the controller.

Delegation

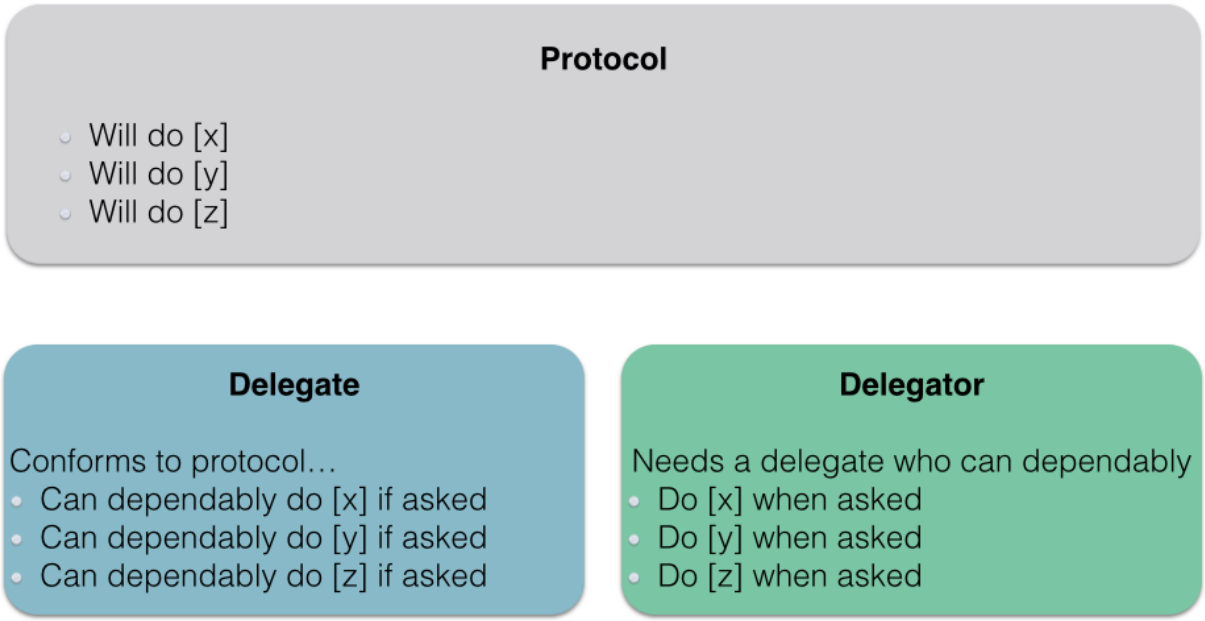


Figure: Roles of Protocol, Delegate, and Delegator

Delegation is used to allow child objects to communicate with its parent without having to know the exact type of its parent. In iOS programming, this is useful when we want our child view controller to communicate with its parent view controller or when we want our tableview cell to communicate with the parent tableview in response to some event. In other words, delegation is just a way for one class to give responsibility to another class.

How do we implement delegation in Swift? The answer are **protocols**. According to Apple, "*A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that protocol.*"

In other words, you can think of protocols as being similar to interfaces in Java. It is just a set of methods and/or properties that any class that conforms to it needs to implement. Lets take a look at an example:

```
protocol ViewControllerDelegate {

    var someVariable: Int { get }

    func viewControllerDidReceiveSomeEvent()

}
```

To declare a protocol, we simply have to preface the name of our protocol with the **protocol** keyword. Inside the block, we then define the set of methods and properties that we want any class that conforms to this protocol to implement. Thus, if **ViewControllerB** conforms to **ViewControllerDelegate**, ViewControllerB must define some value for a variable called **someVariable** and also have a method whose name is **viewControllerDidReceiveSomeEvent()**.

One good way to wrap your head around delegation and protocols is to take a real life example. Lets say that we were in a restaurant. This restaurant has a chef, a waiter, and a menu. In this case, the **menu** is our protocol. Why? The menu contains a list of dishes that some one must be able to make. Similarly, the **waiter** is the delegator. The waiter's job is just to receive the orders of customers but they cannot themselves make the dishes, they need someone else who can. Lastly, the **chef** is the delegate because they are the one who can make the dishes on the menu when asked by the waiter.

REVIEW

Whats the difference between pushing/popping and presenting/dismissing a view controller?
Well, pushing a view controller means that you're "pushing" a view controller onto the navigation controller's stack of view controllers (for those unfamiliar with stacks, when ever you insert an element onto a stack, you place it on the very top. similarly when you pop, you remove the element thats at the very top). For example, lets say that we had a UINavigationController whose rootViewController was ViewController.
The navigation stack would currently just be: [View Controller]

If we then pushed RedViewController, then the navigation stack would be:
[
RedViewController,
ViewController
]

Similarly, if we were to "pop" RedViewController, the stack would return to just be: [ViewController].
In other words, pushing/popping involves working with a UINavigationController.
On the other hand, presenting/dismissing a view controller simply means displaying a view controller on top of the current screen (does not affect the navigation stack). For the differences in animation between pushing/popping and presenting/dismissing, refer to the Lecture 3 code.

UITableView

What is a UITableView?

A table view (UITableView) is a subclass of UIScrollView, which means that it is a view that users can scroll through. Simply put, the main purpose of a table view is to display information in the form of a list. A bit more specifically, a tableview consists of sections where each section consists of 3 parts: a header, cells (in this case, you can just think of a cell as a row), and a footer.

Why are UITableViews important?

UITableViews are an extremely powerful type of view and are used in almost all of the successful apps today (i.e. Spotify playlist, Gmail inbox, Messenger messages). They provide an easy way to display a lot of information in a compact, organized manner.

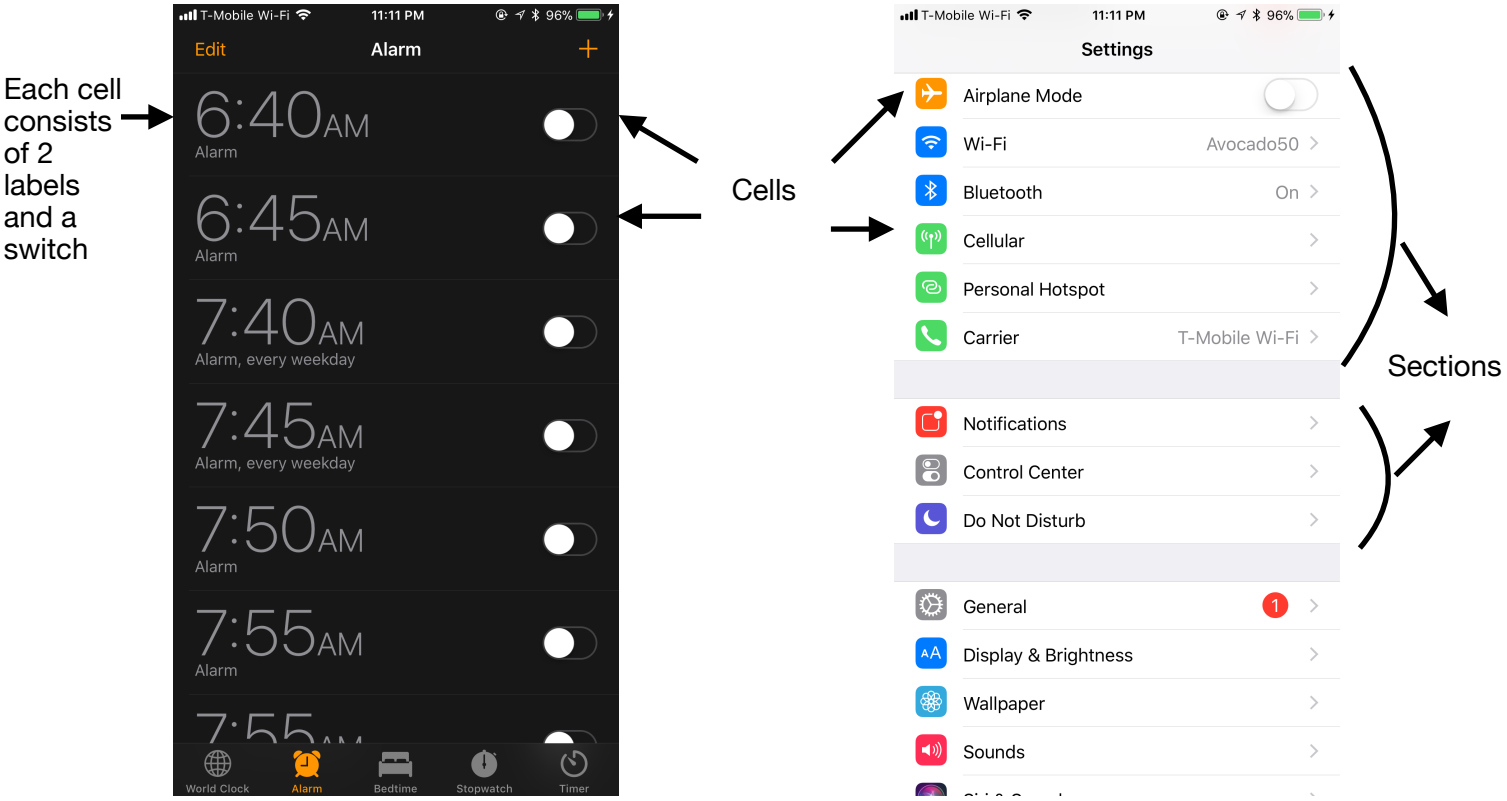


Figure: Display of table view inside the Clock and Settings app

How do we setup a UITableView?

Besides the usual initialization process for initializing any UIView (calling init function, set translatesAutoresizingMaskIntoConstraints to false, adding view as a subview to parent view, and setting up constraints), there are 3 additional things that you need to do.

1. Set the tableView's delegate
2. Set the tableView's dataSource
3. Register your cell class

We will discuss **1** and **2** in the next section. What does **3** mean? Basically, in order to use any custom cell class that we create inside the tableView and be able to **dequeue** it later on, we need to let the tableView know by *registering* the cell class and associating it with an identifier (this can be any string).

What does it mean to dequeue a cell and why do we want to do it? Well, imagine that we were using a tableview to display a list of all the students at Cornell. This means that we would have thousands of cells (**really expensive** in terms of performance). Instead of creating a new cell for each student, as the user scrolls down the list and the cells near the top disappear from the screen, we reuse those cells to display the next set of cells. This way, we will only be creating 5 or 6 cells instead of thousands.

```
// Initialize tableView
tableView = UITableView(frame: .zero)
tableView.translatesAutoresizingMaskIntoConstraints = false
tableView.delegate = self
tableView.dataSource = self
tableView.register(CafeteriaTableViewCell.self,
forCellReuseIdentifier: someReuseIdentifier)
// add tableView as a subview to parent view and setup constraints
```

UITableViewDelegate and UITableViewDataSource

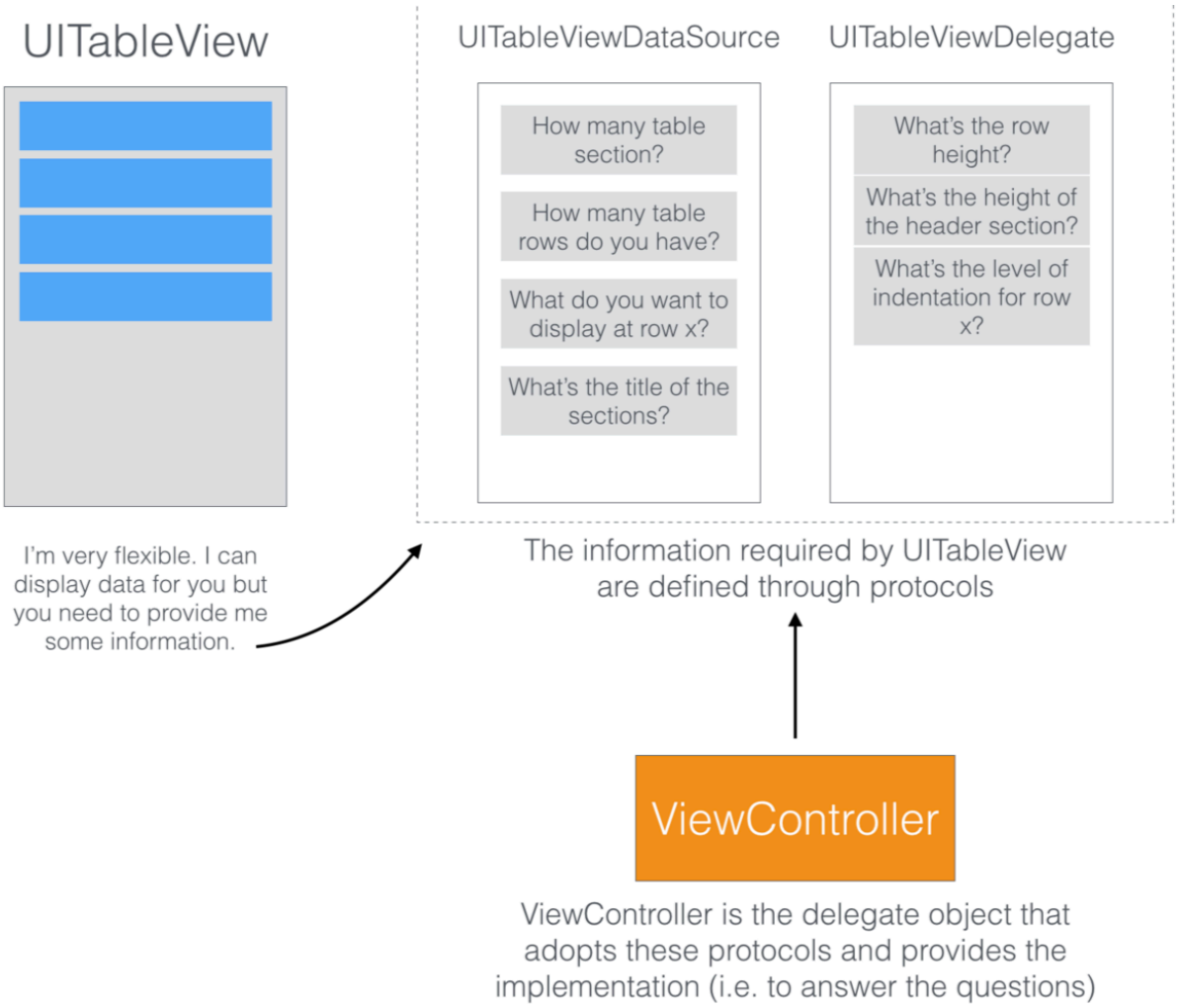


Figure: Diagram of role of UITableViewDataSource/Delegate

A **UITableViewDelegate** must conform to certain methods and can also provide some optional methods which allows the delegate to manage selections, configure section headings and footers, help to delete and reorder cells, and perform other actions. The one method that you should always implement is

```
func tableView(_ tableView: UITableView, heightForRowAt indexPath:
IndexPath) -> CGFloat {
        return 50
}
```

This function requires you to return a CGFloat representing the height that the cell at a certain row or section should have. Some other common methods that you will probably need is

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
        // Perform some action upon selecting a row
}
```

This function is what gets called whenever one of the cells in your table view is selected by the user. Thus, if you want to trigger some sort of action or animation upon selection, this is the place you would do it.

Similarly, a **UITableViewDataSource** must conform to certain methods. Unlike the UITableViewDelegate, however, the role of a UITableViewDataSource is to provide the table-view object with the information it needs to construct and modify a table view. The first method that you need to implement is

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
customReuseIdentifier, for: indexPath) as! CustomCell
        let dataModel = dataModelArray[indexPath.row]
        cell.configure(…) // call some custom cell configuration function
        cell.setNeedsUpdateConstraints() // tell cell to updateConstraints
        return cell
    }
```

This function requires you to return a **UITableViewCell** which will be inserted at a particular location in the tableView. One parameter in this function that is important to take note of is **indexPath** which is of type **IndexPath**. The indexPath has two important properties that you may sometimes need, **row** and **section**. Thus, if you wanted to see what row or section the cell you're returning is going to be used for, you can just call **indexPath.row** or **indexPath.section**.

Going over the contents of the code in this example, in the first line, you will first want to dequeue the appropriate cell using the associated reuse identifier that you created when you registered your new cell class. Next, you will usually have some **configure(…)** function as a part of your cell class that takes in some data or model object to configure the views inside your cell. If you are using some dataModelArray to create your table view, you can access the dataModel object that you need for this row using **dataModelArray[indexPath.row]**. Then, you should call **setNeedsUpdateConstraints()** on your cell. This will call the cell's **updateConstraints()** function which is where you should be placing your constraints for views in your cell. Lastly, you return the cell object itself.

The other method you need to implement is

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
        // Each model object in the array is used to configure one cell
        return dataModelArray.count
    }
```

This function requires you to return an integer representing the number of rows that *section* should have. Usually, the number of cells that you want to display in your table view is dependent on the number of model objects that you have. For example, if you wanted to use a table view to display a list of restaurants, you would probably have an array of **Restaurant** objects. Thus, the number of cells you need in your table view is just the number of restaurants which is equivalent to **restaurantsArray.count**. However, it is also possible that you just want to create a static number of cells, in which case you can just return that constant.

Some other common methods that you may sometimes want to implement is

```
func numberOfSections(in tableView: UITableView) -> Int {
        // Return number of sections that you want
        return 2
}
```

As the name implies, this allows you to control the number of sections that your table view should have. One thing to note is that if you do no implement this function, your table view will by default have 1 section.

Note: You can find a list of all the **UITableViewDelegate & UITableViewDataSource**

Lets Build Messenger!

Now that we've covered the basics of setting up and creating an actual table view, lets take a look at real-life example of table views in action and see how we would build it if we had the actual data just to make sure we fully understand the concepts. The example application we're going to be looking at is Messenger, used by over **1.3 billion** people around the world!

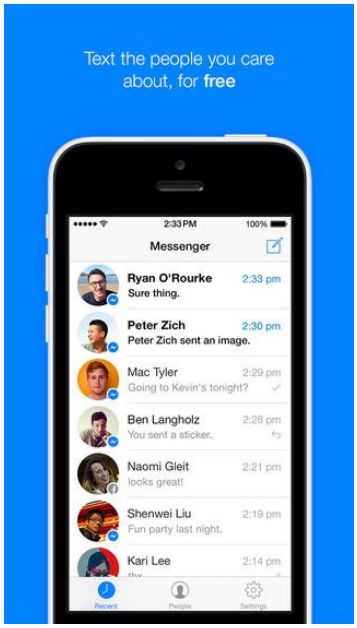


Figure: Display of table view in Messenger App.

Breaking down this table view, we see that each cell has a couple pieces of information: a profile **image**, a user's **name**, the latest **message** sent/received, and the **timestamp** of that latest message. Lets assume that all of this information is embodied in some **User** object that we have. Thus, we would probably have an array of Users in some **users** array. It also seems that this table view is just comprised of one section (since there are no major breaks between groups of cells). Let us also assume that the height of each cell is 75 pixels and that upon selecting a cell, we want to push another view controller which displays all the conversation history that the logged in user has had with the user of the cell that was clicked on. Putting this all together, we have:

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
        return users.count
    }
```

Because we want the number of cells to just be the number of users that we have messaged, in **numberOfRowsInSection**, we can just return **users.count**.

```
    func tableView(_ tableView: UITableView, heightForRowAt indexPath:
IndexPath) -> CGFloat {
        return 50
```

Here, we arbitrarily want the height of each cell to be 50 pixels but you can choose this to be some other value if you would like.

```
    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
MessengerCell.self, for: indexPath) as! MessengerCell
        let user = users[indexPath.row]
        cell.configure(user: user)
        cell.setNeedsUpdateConstraints()
        return cell
    }
```

Here, we dequeue a **MessengerCell**, get the appropriate **User** model for this cell, use it to configure the cell, tell the cell to set constraints, and then finally return the cell.

```
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
         let user = users[indexPath.row]
        let conversationVC = ConversationViewController(user: user)
        self.navigationController?.pushViewController(conversationVC,
animated: true)
```

Here, we have decided to make it so that when a **MessengerCell** is tapped, we push a **ConversationViewController** after grabbing the **User** model associated with this cell.

UICollectionView

What is a UICollectionView?

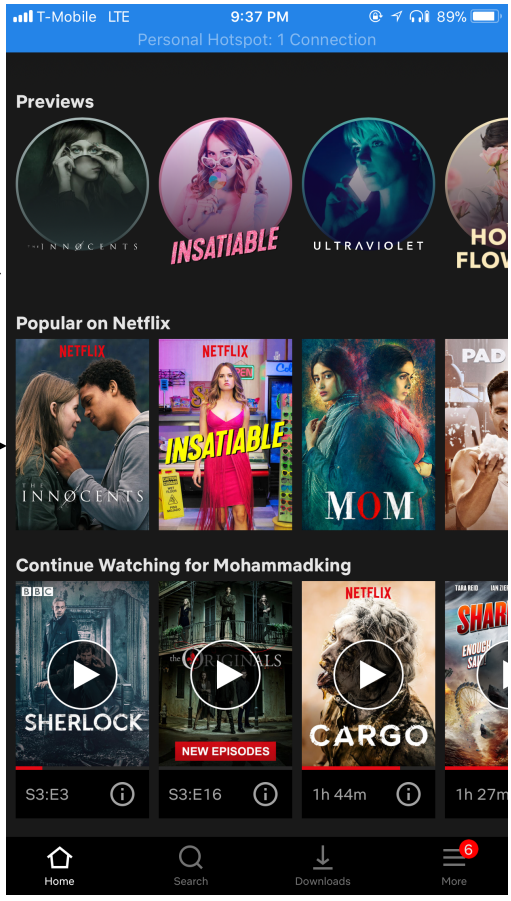
A collection view (UICollectionView) is a subclass of UIScrollView, which means that the user can scroll through it, either horizontally or vertically. Just like a table view, a collection view consists of sections where each section consists of 3 parts: a header, cells, and a footer. However, unlike the table view (in which each cell is constrained to be the full width of the table view), the size of the cells in a collection view can be set. In other words, a collection view is a more dynamic, customizable table view.

Why are UICollectionViews important?

Like table views, UICollectionViews are an extremely powerful type of view and are used in almost all of the successful apps today (i.e. Instagram, Photos, Stories, etc.). They provide an easy way to display a lot of information in a compact, **grid-like** manner.

Each category is a cell. Each cell contains a collection view

Each show in this row is a collection view cell



This **collection** of images make up one collection view. Each image here is a cell.

Figure: Display of collection view inside Netflix and Photos app

How do we setup a UICollectionView?

Similar to a table view, besides the usual initialization process for initializing any UIView (calling init function, set translatesAutoresizingMaskIntoConstraints to false, adding view as a subview to parent view, and setting up constraints), there are **4** additional things that you should do.

1. Set the collectionView's delegate
2. Set the collectionView's dataSource
3. Register your cell class
4. (**NEW**: the first 3 steps are similar to table view) Initialize the collectionView with a **UICollectionViewLayout** (we use *UICollectionViewFlowLayout)*. Also, your collectionView's delegate should conform to **UICollectionViewDelegateFlowLayout**.

For explanations on what the first 3 mean, refer to the **L4-UITableView** handout. In terms of **4**, the initializer for a UICollectionView takes in a *frame: CGRect* and a *layout:UICollectionViewLayout*. In most cases, we just use an instance of **UICollectionViewFlowLayout** for this layout parameter. Why? Well, a UICollectionViewFlowLayout "*layout object that organizes items into a grid with optional header and footer views for each section.*"

Some nice properties of **UICollectionViewFlowLayout** that we can set to customize our collectionView are *scrollDirection, minimumLineSpacing (minimum spacing to use between lines of items in the grid), minimumInteritemSpacing (*minimum spacing to use between items in the same row), and a lot more.

```
let layout = UICollectionViewFlowLayout()
layout.scrollDirection = .vertical
layout.minimumLineSpacing = padding // optional
layout.minimumInteritemSpacing = padding // optional
collectionView = UICollectionView(frame: .zero, collectionViewLayout:
layout)
collectionView.translatesAutoresizingMaskIntoConstraints = false
collectionView.delegate = self
collectionView.dataSource = self
collectionView.register(PhotoCollectionViewCell.self,
forCellWithReuseIdentifier: photoCellReuseIdentifier)
```

Lets quickly go over what this code does. We initialize a *UICollectionViewFlowLayout* and then set its **scrollDirection** to be vertical. Then we set the **minimumLineSpacing** and **minimumInteritemSpacing** to be some constant we defined as padding. This means that items in the same row will have spacing between them equal to padding and items on different rows will also spacing between them equal to padding. Next, we initialize our collectionView using our layout and set the **delegate** and **dataSource** to be self (the view controller that holds this collectionView). Lastly, we register our custom **PhotoCollectionViewCell** which means tells the collectionView that we will be dequeueing PhotoCollectionViewCells.

UICollectionViewDataSource, UICollectionViewDelegate, and UICollectionViewDelegateFlowLayout

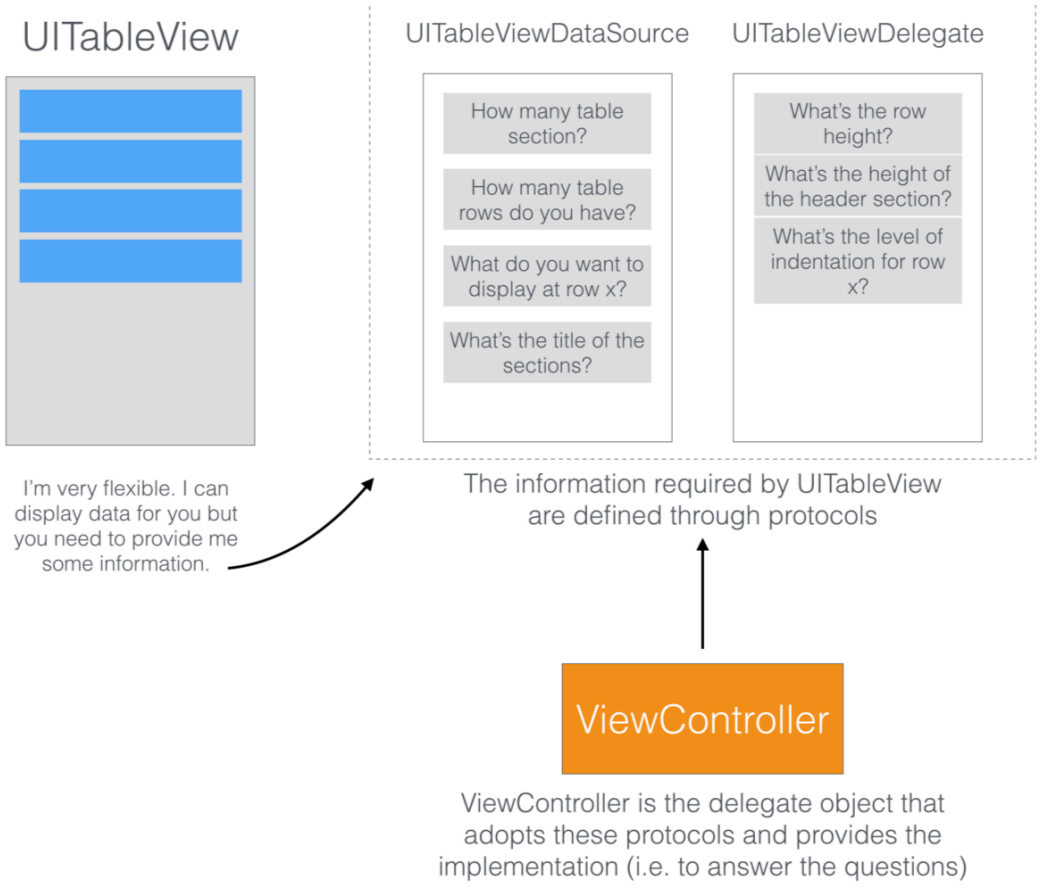


Figure: Display of UITableViewDataSource/Delegate.
NOTE: This is basically the same as UICollectionViewDataSource/Delegate except replace "rows" with "items"

A **UICollectionViewDataSource** must conform to certain methods and is "*responsible for providing the data and views required by a collection view*." The one method that you should always implement is

```
func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
    let cell =
collectionView.dequeueReusableCell(withReuseIdentifier:
photoCellReuseIdentifier, for: indexPath) as! PhotoCollectionViewCell
    let dataModel = dataModelArray[indexPath.item]
    cell.configure(for: dataModel)
    cell.setNeedsUpdateConstraints()
    return cell
}
```

This function requires you to return a **UICollectionViewCell** which will be inserted at a particular location in the collectionView. One parameter in this function that is important to take note of is **indexPath** which is of type **IndexPath**. The indexPath has two important properties that you may sometimes need, **item** (**note**: for tableViews, we have *rows* instead of *items*) and **section**. Thus, if you wanted to see what item or section the cell you're returning is going to be used for, you can just call **indexPath.item** or **indexPath.section**.

Going over the contents of the code in this example, in the first line, you will first want to dequeue the appropriate cell using the associated reuse identifier that you created when you registered your new cell class. Next, you will usually have some **configure(...)** function as a part of your cell class that takes in some data or model object to configure the views inside your cell. If you are using some dataModelArray to create your collection view, you can access the dataModel object that you need for this row using **dataModelArray[indexPath.item]**. Then, you should call **setNeedsUpdateConstraints()** on your cell. This will call the cell's **updateConstraints()** function which is where you should be placing your constraints for views in your cell. Lastly, you return the cell object itself.

```
func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
    return dataModelArray.count
}
```

This function requires you to return an integer representing the number of rows that *section* should have. Usually, the number of cells that you want to display in your collection view is dependent on the number of model objects that you have. For example, if you wanted to use a collection view to display a collection of photos, you would probably have an array of **Photo** objects. Thus, the number of cells you need in your table view is just the number of photos which is equivalent to **photosArray.count**. However, it is also possible that you just want to create a static number of cells, in which case you can just return that constant.

Another method that you may sometimes want to implement (but is not required):

```
func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 2
}
```

As the name implies, this allows you to control the number of sections that your collection view should have. One thing to note is that if you do no implement this function, your collection view will by default have 1 section. (similar to table views)

A **UICollectionViewDelegate** does not actually *have* to conform to any specific methods since all the methods are optional. However, one useful one that you may find yourself using a lot is:

```
func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
    // Perform some action upon selecting an item
}
```

This function is what gets called whenever one of the cells in your collection view is selected by the user. Thus, if you want to trigger some sort of action or animation upon selection, this is the place you would do it. For example, if you wanted to change some data model property upon selecting a cell, you could get that specific data model using **indexPath.item**.

Lastly, a **UICollectionViewDelegateFlowLayout** also does not actually *have* to conform to any specific methods since all the methods are optional. However, some methods that you may want to implement is:

```
func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt
indexPath: IndexPath) -> CGSize {
    return CGSize(width: 50, height: 50)
}
```

This method expects a CGSize which represents the size of the cell at *indexPath*. Thus, if we wanted each of our cells to be 50 pixels by 50 pixels, we would return **CGSize(width: 50, height: 50)**.

Note: You can find a list of all the **UICollectionViewDelegate, UICollectionViewDataSource, & UICollectionViewDelegateFlowLayout** methods by simply performing a google search for either one and going to the Apple documentation.

Cocoapods, HTTP, Networking, and JSON

When building out our applications, there will come times when we want to build a very complicated UI or perform some sort of special task. In these cases, we have two options. We can try to build this new feature from scratch all by ourselves **OR** we could use a library (code) which other people have already written and made public because they figured that other people would want to have this feature in their application.

What is Cocoapods?

From the cocoa pods website: "*CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects. It has over 53 thousand libraries and is used in over 3 million apps.*" Let us break down what this means. A **dependency** just refers to a library, or code, that someone else has written and made public for others to use. A **dependency manager** is a tool that manages all the libraries that are used in your application. Just to give some examples for Python application's dependency manager is pip and node.js application's dependency manager is npm.

For Swift application's this is Cocoapods. Some extremely popular and useful Cocoapods that we definitely recommend you guys check out is **Alamofire** (networking library) and **SnapKit** (library for easy AutoLayout). Essentially, we will be using Cocoapods to install and be able to use any external libraries that we want to use in our application.

Setting up Cocoapods

Before we can use Cocoapods, we must install it on our computers. To do so, go to your terminal and run the following command (NOTE: you should have Xcode installed already beforehand):

```
$ sudo gem install cocoapods
```

Once this command finishes running, you should have cocoapods installed and should not have to reinstall it ever again. Now, in order to setup cocoapods in our Xcode project, use your terminal and navigate to your project's root directory. Once inside run the following command:

```
$ pod init
```

What this command does is create a boilerplate **Podfile** inside the current directory. Upon opening the Podfile, you should see something like this:

```
platform :ios, '9.0'

# ignore all warnings from all pods
inhibit_all_warnings!

target 'SampleProject' do
  use_frameworks!

  # Pods for SampleProject
  pod 'Alamofire'
  pod 'SnapKit'
end
```

In the snippet above I inserted two cocoapods already as an example but yours shouldn't contain any in the beginning. Essentially, anytime you want to add a new cocoapod to your project, all you have to do is come into this Podfile and insert it as **pod '[name]'**. In this snippet, I am declaring that I want to use the **Alamofire** and **SnapKit** cocoapods. Once you have written all the cocoa pods that you need, save the file and return to the terminal. The next command that we want to run is:

```
$ pod install
```

What this command does is it looks at your **Podfile** (needs to exist in the directory or else this command will not work) to see all the cocoa pods that you want to use, installs all of them, and integrates them with your .xcodeproj into a **.xcworkspace.** For example, if we our project was called SampleProject and we had a **SampleProject.xcodeproj**, after running **pod install**, we would have a **SampleProject.xcworkspace** in the same directory. After running pod install, you should always be developing in **.xcworkspace** instead of **.xcodeproj**. This is because **.xcworkspace** is where you will be able to import and use all the cocoa pods that you installed. If at any point later on you want to install more cocoapods, simply go back to the Podifle, add your cocoa pod, and then run pod install again.

How do you make an app communicate with the internet?

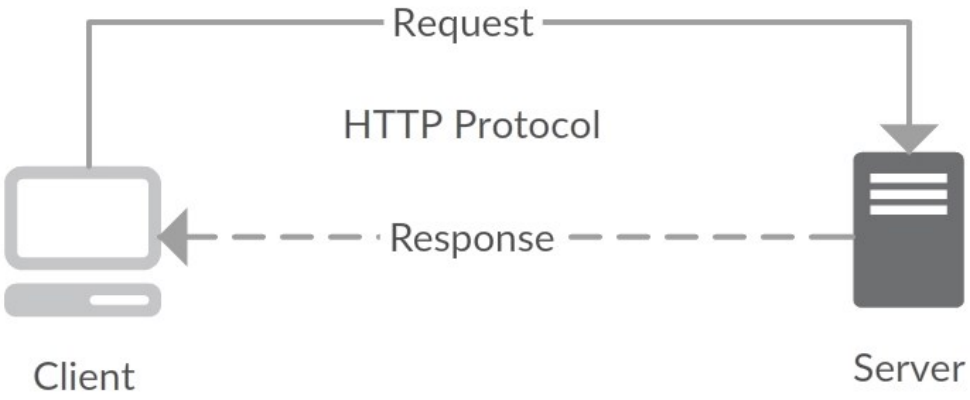


Figure: Display of networking flow.
iOS app makes a request to server, then server sends back a response

So far in this course, we've only dealt with hard-coded data that we've made ourselves to show on screen, but most apps don't have static information. **HTTP requests** are a method to communicate between a client (your iOS app) and a server (the internet).

HTTP requests are used all over development, and there are many different types. The most common types of requests are **GET** and **POST** requests. GET requests are used to get information from the internet (ex. getting information from google.com), and POST requests are used to post information onto the internet (ex. submitting a form). Other methods include PUT, DELETE, and more.

In this class, we'll be using **Alamofire**, a Cocoapod, to make HTTP requests. Here's an example of making a GET request to fetch recent posts from The Cornell Daily Sun's website. We'll just print the data we get back.

Notice that here the method we're using is **responseJSON**. This method takes the response from the server and converts it into a **JSON** object, which we can then print out and see the response in a friendly format. Later on, when we decode the responses, we will switch to using the **responseData** method instead (this converts the response into a **Data** object instead).

```
let endpoint = "http://cornellsun.com/wp-json/wp/v2/posts"

Alamofire.request(endpoint, method: .get)
    .validate().responseJSON { response in
        // Depending on what response JSON we get here,
        // we can appropriately handle it.
        switch response.result {
        // If the response is a success, print the data
        case let .success(data):
            print(data)
        // If the response is a failure, print the error
        case .failure(let error):
            print(error.localizedDescription)
        }
}
```

The printed response will look something like this:

```
[{"id":
3597603,"date":"2018-07-17T01:45:30","date_gmt":"2018-07-17T05:45:30","guid"
:{"rendered":"http:\/\/cornellsun.com\/?
p=3597603"},"modified":"2018-07-17T01:45:30","modified_gmt":"2018-07-17T05:4
5:30","slug":"jurassic-world-fallen-kingdom-bites-off-more-than-it-can-
chew","type":"post","link":"http:\/\/cornellsun.com\/2018\/07\/17\/jurassic-
world-fallen-kingdom-bites-off-more-than-it-can-chew\/","title":
{"rendered":"Jurassic World: Fallen Kingdom Bites Off More Than It Can
Chew"},"content":{ ... etc
```

Interpreting responses from the internet

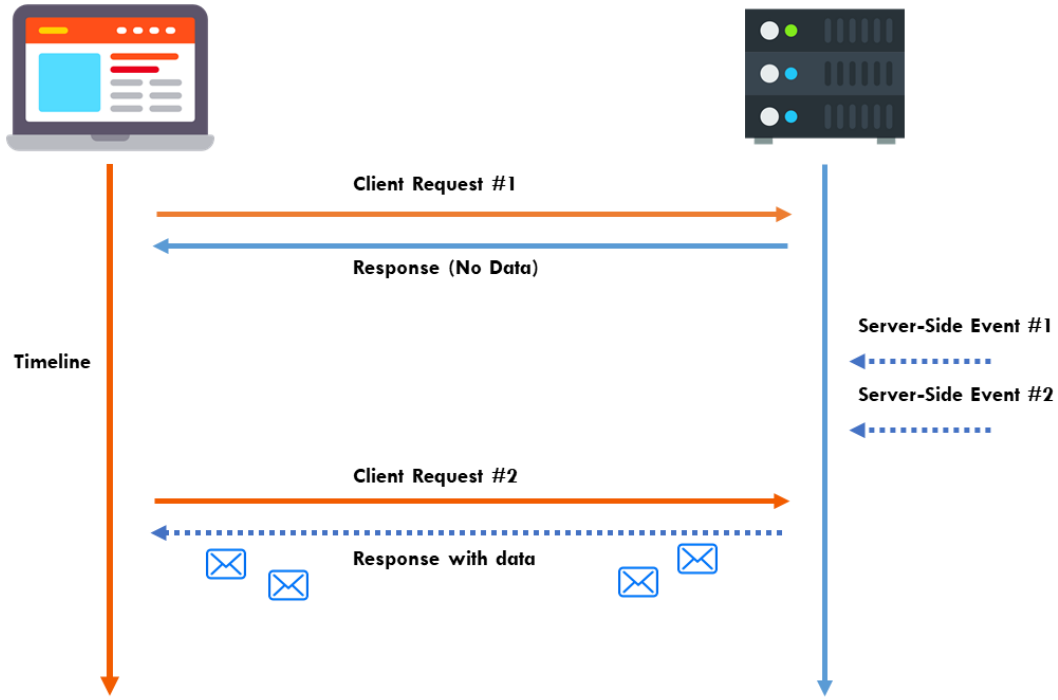


Figure: Timeline of request/response flow

The response below is formatted in **JSON (JavaScript Object Notation)**, which is a key-value coding format commonly used in server responses. Each JSON is wrapped in two curly braces { }, and data is separated by commas. Values in JSON can be strings, numbers, objects (items wrapped in curly braces), null, booleans, or arrays.

Here's an example of a JSON response you might get:

This entire thing is an object because it's wrapped between { }.

```
{
      "title":         "HTTP, Networking, and JSON",
      "year_created": 2018,
      "success":       true,
      "topics":        [ "HTTP", "Networking", "JSON" ],
}
```

Keys Values

Usually, the responses will be condensed into one blob of text (as seen in the previous example), so you can use a JSON pretty-printer to format it to make it easier to read. Here's an example of a good JSON printer that might be useful: https://jsonformatter.org/json-pretty-print

So, how do you decode these responses and put them on screen?

Decoding responses from the internet

Starting in Swift 4, there's a protocol called **Codable** that objects, structs, and enums can conform to to make them encodable and decodable from JSON. Codable items can contain an enum for the coding keys of type String and CodingKey in order to decode the values from JSON.

For example, if we wanted to decode the example JSON response into a model of type lecture, we could create a Lecture struct that conforms to Codable as follows:

```
struct Lecture: Codable {

    var title: String
    var yearCreated: Int
    var success: Bool
    var topics: [String]
}
```

Notice how the names of our variables match the fields in our JSON. This is done on purpose. With Codable, the names of the variables are expected to be the keys in the JSON response.

JSON decoding

In order to decode the JSON response into the Lecture class, we could then create a JSONDecoder, and then call the decode method to convert it from a JSON to a Lecture struct. How cool is that?

```
// Assume we already received the data from an Alamofire call

let jsonDecoder = JSONDecoder()

// Decode the data as a Lecture (optional type)

let lecture = try? jsonDecoder.decode(Lecture.self, from: data)
```

Note: Since the decoding could throw errors, it's preceded by a try?, which will make the lecture constant an optional. This means that lecture will either contain a Lecture item or be nil.

Using your decoded data: `@escaping` completions

Once you've decoded the lecture JSON as a Lecture class, how do you use it? Since you've downloaded this JSON from the internet, it might take forever to load, and you only want to display this information on the screen when it's ready (and possibly use a loading indicator in the meantime as the information loads). We also don't want our application to just wait until we get a response to do something else.

This is where a **completion handler** comes in handy. A completion handler is an argument passed into whatever networking function you make that will take in your decoded object(s) and pass them to the ViewController (or wherever you're calling this function) and then run some block of code that you write (i.e. perhaps reload the view with the data on screen).

This completion handler is marked as @escaping because it can escape from the networking function call itself and hand it off to whoever called the function - it doesn't need to go through every line inside the function. Back to the Lecture class example from before: suppose we want to make a function that makes the network call and returns to us the Lecture JSON decoded as a Lecture class. Here's an example of what that would look like:

```
static func getLecture(completion: @escaping (Lecture) -> Void) {

    Alamofire.request(endpoint, method: .get)
        .validate().responseData { response in
            switch response.result {
            case let .success(data):
                let jsonDecoder = JSONDecoder()
                if let lecture = try? jsonDecoder.decode(Lecture.self, from:
data) {
                    completion(lecture)
                }
            case .failure(let error):
                print(error.localizedDescription)
            }
    }

}
```

you can call the completion function on the lecture object you received

Here, you can see that **completion** is a function that takes in one argument which is of type **Lecture** and then has a return type of **Void** (i.e. does not return anything). We only call completion when we know that the response succeeded and we actually have an actual Lecture object.

Also note that here, we are using the **responseData** method instead of the **responseJSON** method. This is important because responseData converts the response into a **Data** object which is what **JSONDecoder** needs to decode it.

Making network calls

It's good practice to create a NetworkManager class with static methods for every network call, with the endpoint urls hidden to the public (in case it's private information).

Then, in your ViewController, you can make a network request and handle it appropriately. For example, if you have a variable for lecture and need to reload the data of a table view after you get your lecture, you would do it like so:

```
func getClasses() {
   NetworkManager.getLecture { lecture in
        self.lecture = lecture
        self.tableView.reloadData()
   }
}
```

The `{ lecture in … }` is the escaping completion we defined in the NetworkManager earlier. In this case, we get the lecture variable only if we made the network call to the endpoint, received a success, received the JSON, and then decoded it. If succeeded, we run this block of code. This ensures that we receive the Lecture object only when it's ready, so the screen isn't frozen while the app is trying to execute all of these network request steps.

POST requests and query parameters

Sometimes, you want to send information to the network instead of just receiving it. For example, when you search for something (for example, "Taylor Swift") in iTunes, you make a network request by sending the string "Taylor Swift" to the backend in a POST request.

In Alamofire, you can pass in a Parameter object in a network request to the endpoint you choose. The Parameter object is a dictionary mapping strings to strings. For example, in iTunes, the search endpoint requires four parameters: term, country, media, and entity. The search term is the keywords joined by the + sign (so "Taylor Swift" would become "Taylor+Swift".

For example, let's say we're searching songs by Taylor Swift in the US so the media type is music and the entity type is songs. Here's what this would look like as an Alamofire network request (on the next page).

Note on Alamofire query parameters: Depending on how your endpoint receives parameters on the backend, you might need to add in another argument to your Alamofire request. For example, if your search endpoint takes in query strings, you'll need to add the extra argument for query string parameter encoding, which is `URLEncoding(destination: .queryString)`. If you're having trouble with your Alamofire POST requests this most likely will fix it! For more information, see this StackOverflow post here.

```
private static let endpoint = "https://itunes.apple.com/search"

static func getTracks(withQuery query: String, completion: @escaping
([Song]) -> Void) {

    let parameters: Parameters = [
        "term" : query.replacingOccurrences(of: " ", with: "+"),
        "country" : "US",
        "media" : "music",
        "entity" : "song"
     ]

    Alamofire.request(endpoint, parameters: parameters)
                        .validate().responseJSON { response in

            // handle response here
            switch response.result {
              …
            }
    }
}
```