Project 1: Storm Viewer: an introduction to Swift

Overview

Brief: Get started coding in Swift by making an image viewer app and learning key concepts.

Learn: Constants and variables, method overrides, table views and image views, app bundles, NSFileManager, typecasting, arrays, loops, optionals, view controllers, storyboards, outlets, UIImage.

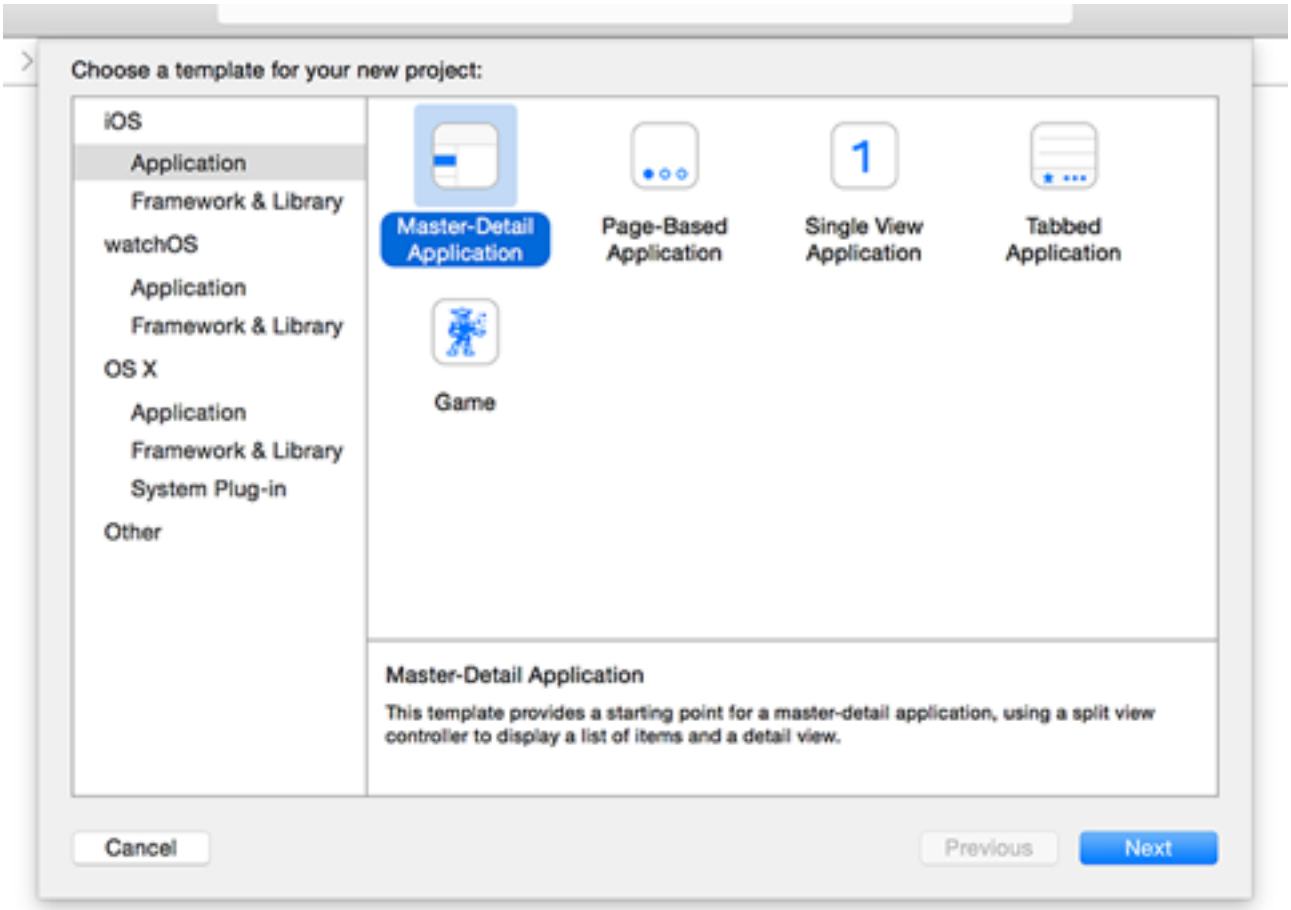
- Setting up
- Deleting skeleton code
- Listing images with NSFileManager
- Introducing Interface Builder
- Loading images with UIImage
- Final tweaks: hidesBarsOnTap
- Wrap up

Setting up

In this project you'll produce an application that lets users scroll through a list of images, then select one to view. It's deliberately simple, because there are many other things you'll need to learn along the way, so strap yourself in – this is going to be long!

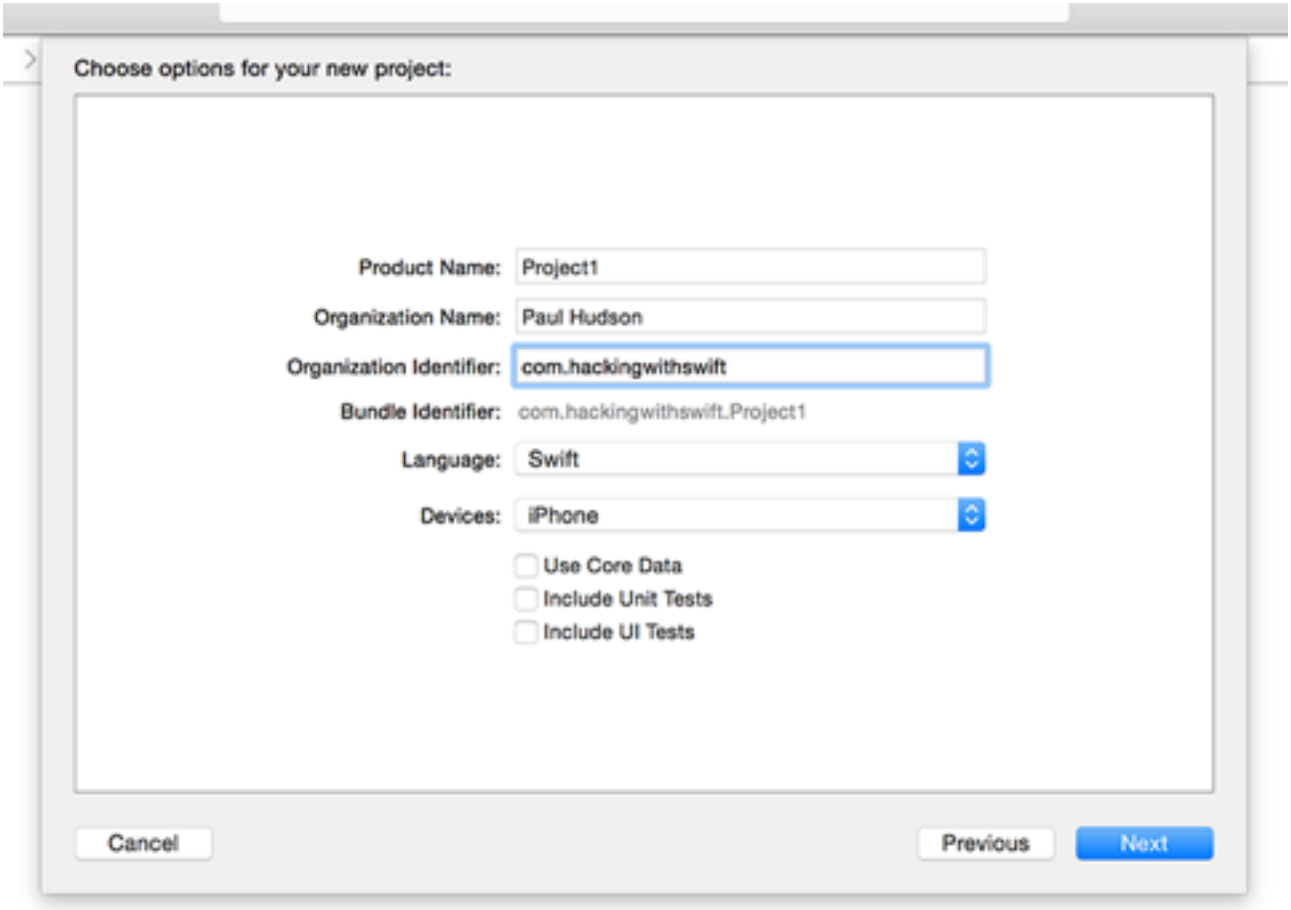
Launch Xcode, and choose "Create a new project" from the welcome screen. Choose Master-Detail Application from the list and click Next. For

`Product Name` enter `Project1`, then make sure you have `Swift` selected for language and `Universal` for devices.



One of the fields you'll be asked for is "`Organisation Identifier`", which is a unique identifier usually made up of your personal web site domain name in reverse. You'll need to put something valid in there if you're deploying to devices, but otherwise you can just use **`com.example`**.

Important note: some of `Xcode's` project templates have checkboxes saying "`Use Core Data`", "`Include Unit Tests`" and "`Include UI Tests`". Please ensure these boxes are unchecked for this project and indeed all projects in this series.



Now click `Next` again and you'll be asked where you want to save the project – your desktop is fine. Once that's done, you'll be presented with the example project that `Xcode` made for you. The first thing we need to do is make sure you have everything set up correctly, and that means running the project as-is.

When you run a project, you get to choose what kind of device the `iOS Simulator` should pretend to be, or you can also select a physical device if you have one plugged in. These options are listed under the `Product > Destination` menu, and you should see `iPad 2`, `iPad Air`, `iPhone 6`, and so on.

There's also a shortcut for this menu: at the top-left of Xcode's window is the `Play` and `Stop` button, but to the right of that it should say `Project1` then a device name. You can click on that device name to select a different device.

For now, please choose `iPhone 5s`, and click the `Play` triangle button in the top-left corner. This will compile your code, which is the process of converting it to instructions that `iPhones` can understand, then launch the simulator and run the app. As you'll see when you interact with the app, this basic project adds dates to the table when you click the + button, you can delete them either by swiping or using the `Edit` button, and tapping on a date brings in a new screen showing the date in its center.

You'll be starting and stopping projects a lot as you learn, so there are three basic tips you need to know:

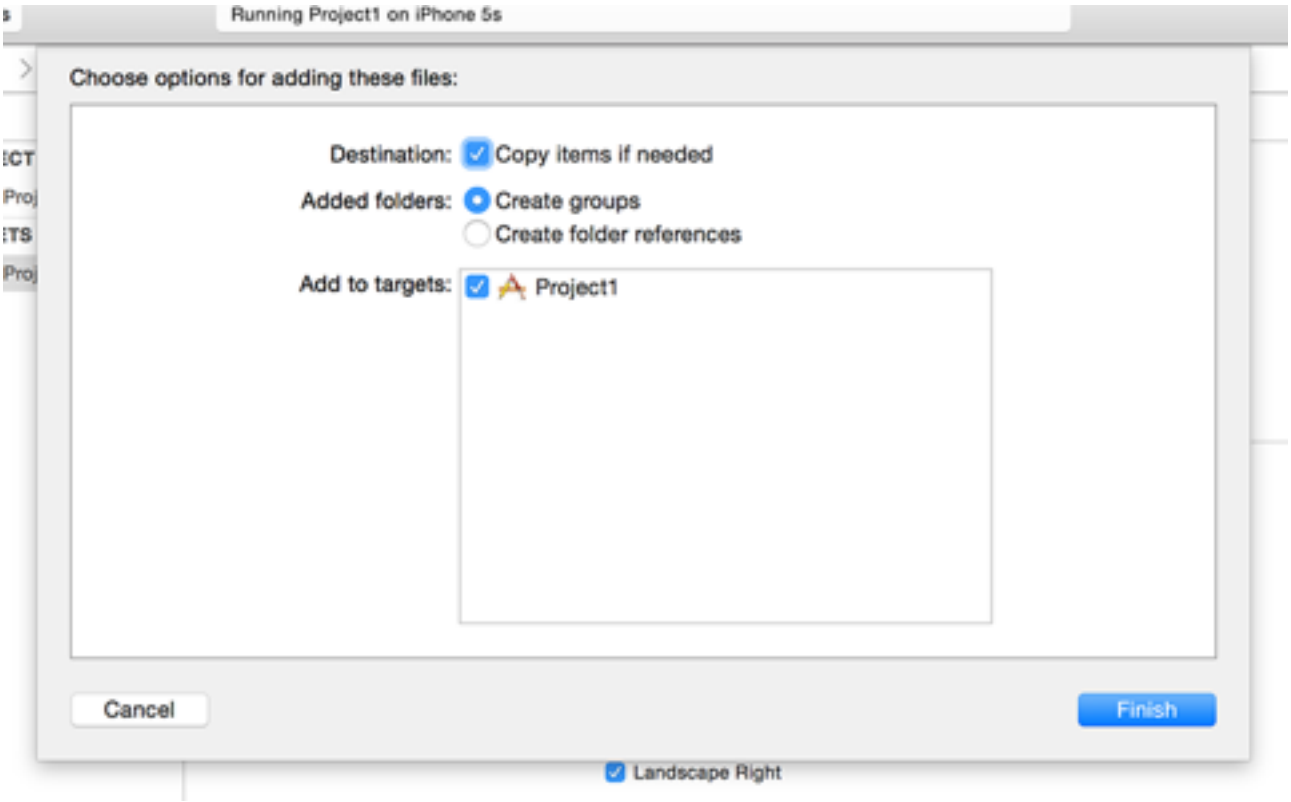
- You can run your project by pressing "`Cmd + R`". This is equivalent to clicking the `Play` button.

- You can stop a running project by pressing "`Cmd +`".

- If you have made changes to a running project, just press "`Cmd + R`" again. `Xcode` will prompt you to stop the current run before starting

another. Make sure you check the “Do not show this message again” box to avoid being bothered in the future.

This project is all about letting users select images to view, so you're going to need to import some pictures. Open the “Tutorial 1. Storm Viewer” folder. You'll see another folder in there called Project1, and inside that a folder called Content.



I want you to drag that Content folder straight into your Xcode project, just under where it says “Info.plist”. A window will appear asking how you want to add the files: make sure “Copy items if needed” is checked, and “Create groups” is selected. Important: do not choose “Create folder references” otherwise your project will not work.

Click Finish and you'll see a yellow Content folder appear in Xcode. If you see a blue one, you didn't select “Create groups”, and you'll have problems following this tutorial!

Deleting skeleton code

Apple's example contains lots of code we don't need, so let's zap it: select the file MasterViewController.swift to open it for editing. Around line 17 you'll see the code starting with override func viewDidLoad() {, and there'll be some more lines of code until it reaches a } on a line all by itself on line 28. If you're not sure which } I mean, it's the one that is aligned directly beneath the first letter in override.

No line numbers? If your Xcode isn't showing line numbers by default, I suggest you turn them on. Go to the Xcode menu and choose Preferences, then choose the Text Editing tab and make sure "Line numbers" is checked.

This block of code is the viewDidLoad() method, which is code that gets called when the system has finished creating the screen and is giving you the chance to configure it.

The method starts at the func viewDidLoad() { line and ends on the } not far below. These symbols, known as braces (or sometimes curly brackets) are used to mark chunks of code, and it's convention to indent lines inside braces so that it's easy to identify where code blocks start and end. But enough of the theory: almost everything inside this method is not needed, so delete its contents except for the line super.viewDidLoad().

Note: when I say "delete its contents" I mean leave the func viewDidLoad() { and } intact, but remove everything in between except for that one line. So, it should look like this:

```
override func viewDidLoad() {
```

```
    super.viewDidLoad()
}
```

This method does nothing now, but that's OK: we'll be filling it in later.

Next up, look for the `insertNewObject()` method, which starts with `func insertNewObject() {` and ends with a closing brace a few lines later. Delete the entire method – yes, even the `func insertNewObject(sender: AnyObject) {` and `}` parts.

These methods start with func, which is short for "`function`", which for nearly all intents and purposes is identical to a method in `Swift`. There is one small exception, but you won't come to it until project 24 so for now please just consider functions and methods identical.

Finally, delete the very last method in the file. This one has a very peculiar name, and it's a bit of a hangover from `Apple's` previous language, `Objective-C`. The method is quite long, and although we don't need it here, you do need to understand what it means. Here's the method definition:

```
override func tableView(tableView: UITableView,
commitEditingStyle editingStyle: UITableViewCellEditingStyle,
forRowAtIndexPath indexPath: NSIndexPath)
```

Let's break that down…

- `override` – This means the method has been defined already, and we want to override the existing definition with this new definition. If you didn't override it, then the previously defined method would execute, and in this instance it would do nothing.

- `func tableView` – The method's name is `tableView`, which doesn't sound very useful. But the way `Apple` defines methods is to ensure that the information that gets passed into them – the parameters – are named usefully, and in this case the very first thing that gets passed in is the table view that triggered the code. A table view, as you might have gathered, is the scrolling thing that's containing all the dates in the example project, and is a core component in `iOS`.

- `tableView: UITableView` – As promised, the first thing that gets passed into to the method is the table view that triggered the code, and here it is. But this contains two pieces of information: tableView is the name that we can use to reference the table view inside the method, and `UITableView` is the data type – the bit that describes what it is. Everything that begins with "`UI`" is something `Apple` provides as part of its `iOS` development kit, so `UITableView` is the default `Apple` table view.

- `commitEditingStyle editingStyle: UITableViewCellEditingStyle` – This provides the really important part of the method: what it's trying to do. We know it involves a table view because that's the name of the method, but the `commitEditingStyle` part is the actual action: this code will be triggered when the user tries to commit the editing style of a table view. Translated, that means this code will be called when the user tries to add or delete data from the table.

  But wait: there's more! When we had tableView: `UITableView`, it meant we could reference the table view by using the "`tableView`" parameter, but here the parameter is `commitEditingStyle`, which would be quite a clumsy way to reference the parameter. So `Swift` lets you give parameters external names: one used when passing data in

(commitEditingStyle, in this case) and one to be used inside the method (editingStyle). Finally, there's the colon and UITableViewCellEditingStyle, which means that the editingStyle parameter will be of data type UITableViewCellEditingStyle.

- forRowAtIndexPath indexPath: NSIndexPath – Here's that sneaky parameter naming again, but again hopefully you can see how it's useful: people calling the method would write forRowAtIndexPath, and inside the method you would use just indexPath. It's of data type NSIndexPath, which holds two pieces of information: a table section and a table row. We aren't using sections here, but we are using rows – in the example project, every date that is inserted is one row.

- I'm not going to pretend it's easy to understand how Swift methods look and work, but the best thing to do is not worry too much if you don't understand right now because after a few hours of coding they will be second nature. At the very least you do need to know that these methods are referred to using their name (tableView) and any named parameters. Parameters without names are just referenced as underscores: _.

So, the method you just deleted is referred to as tableView(_:commitEditingStyle:forRowAtIndexPath:) – clumsy, I know, which is why most people usually just talk about the important bit, for example, "in the commitEditingStyle method".

The last things to delete are a little more subtle: look for the tableView(_:cellForRowAtIndexPath:) method (I'll just call it cellForRowAtIndexPath from now on) and you'll see these two lines of code:

```
let object = objects[indexPath.row] as! NSDate
cell.textLabel!.text = object.description
```

I want you to delete the part that says `as! NSDate` and the part that says `.description`. Your final code should look like this:

```
let object = objects[indexPath.row]
cell.textLabel!.text = object
```

When you make this change, Xcode will start telling you there are problems. This is perfectly normal – there are more changes to come.

Now find the `prepareForSegue()` method and you'll see another as `NSDate` in there that you should delete, so the line should read this:

```
let object = objects[indexPath.row]
```

If you were able to run the project now, you would see it's basically useless because the add button has been removed. But you can't run the program just yet, because it has problems that need to be fixed. That's OK, though, because we're about to bring it to life again…

Listing images with NSFileManager

The images I've provided you with come from the `National Oceanic and Atmospheric Administration (NOAA)`, which is a US government agency and thus produces public domain content that we can freely reuse. Once they are copied into your project, Xcode will automatically build them into your finished app so that you can access them.

Behind the scenes, an iOS (and OS X) app is actually a directory containing lots of files: the binary itself (that's the compiled version of your code, ready to run), all the media assets your app uses, any visual layout files you have, plus a variety of other things such as metadata and security entitlements.

These app directories are called bundles, and they have the file extension .app. Because our media files are loose inside the folder, we can ask the system to tell us all the files that are in there then pull out the ones we want. You may have noticed that all the images start with the name "nssl" (short for National Severe Storms Laboratory), so our task is simple: list all the files in our app's directory, and pull out the ones that start with "nssl".

As I said before, the viewDidLoad() method starts at func viewDidLoad() { and ends at the } a few lines later. We're going to put some more code into that method, just beneath the line that says super.viewDidLoad():

```
let fm = NSFileManager.defaultManager()
let path = NSBundle.mainBundle().resourcePath!
let items = try! fm.contentsOfDirectoryAtPath(path)

for item in items {
    if item.hasPrefix("nssl") {
        objects.append(item)
    }
}
```

I already told you that any data types that start with UI belong to Apple's iOS development kit, but that's only partially true. UI stands for User Interface, and so these types primarily relate to things the user can interact with – UITableView

is a table, UIButton is a button, UITextField is a text entry field, and so on. But there are lots of other data types that Apple provides you with, and here we can see two: NSFileManager and NSBundle.

To cut a long story short, NS is short for NeXTSTEP, software that Apple bought in 1997. NeXSTEP developed technology that still today lies at the heart of iOS. NSBundle and NSFileManager are data types that can do some great work for you, but they don't have a visual component. Incidentally, these NS data types nearly always existing on OS X identically to iOS, whereas all those UI things are available only on iOS.

Enough history, let's look at what this code does:

- `let fm = NSFileManager.defaultManager()` – This declares a constant called fm and assigns it the value returned by `NSFileManager.defaultManager()`. This is a data type that lets us work with the filesystem, and in our case we'll be using it to look for files.

- `let path = NSBundle.mainBundle().resourcePath!` – This declares a constant called path that is set to the resource path of our app's bundle. Remember, a bundle is a directory containing our compiled program and all our assets. So, this line says, "tell me where I can find all those images I added to my app".

- `let items = try! fm.contentsOfDirectoryAtPath(path)` – This declares a third constant called items that is set to the contents of the directory at a path. Which path? Well, the one that was returned by the line before. As you can see, Apple's long method names really does make their code quite self-descriptive!

- `for item in items {` – This starts a loop. Loops are a block of code that execute multiple times. In this case, the loop executes once for every item we found in the app bundle. Note that the line has an opening brace at the end: that signals the start of a new block of code, and there's a matching closing brace four lines beneath.

  Everything inside those braces will be executed each time the loop goes around. We could translate this line as “treat items as a series of text strings, then pull out each one of those text strings, give it the name item, then run the following code block…” We use text strings because `contentsOfDirectoryAtPath()` returns a list of filenames.

- `if item.hasPrefix("nssl") {` – This is the first line inside our loop. By this point, we'll have the first filename ready to work with, and it'll be called `item`. To decide whether it's one we care about or not, we use the `hasPrefix()` method: it takes one parameter (the prefix to search for) and returns either `true` or `false`.

  That “`if`” at the start means this line is a conditional statement: if the item has the prefix “`nssl`”, then… that's right, another opening brace to mark another new code block. This time, the code will be executed only if `hasPrefix()` returned `true`.

- `objects.append(item)` – This code will be executed only if `hasPrefix()` returned `true`, and it adds the matching filename to the end of a list called objects. No, we didn't create that – that was in the Xcode sample project.

In just those few lines of code, there's quite a lot to take in, so let's recap:

- We use `let` to declare constants. Constants are pieces of data that we want to reference, but that we know won't have a changing value. For example, your birthday is a constant, but your age is not – your age is a variable, because it varies.

- `Swift` coders really like to use constants in places most other developers use variables. This is because when you're actually coding you start to realise that most of the data you store doesn't actually change very much, so you might as well make it constant. Doing so allows the system to do some optimisation, and also adds some extra safety because if you try to change a constant the compiler will issue errors.

- Text in `Swift` is represented using the `String` data type. `Swift` strings are extremely powerful and guaranteed to work with any language you can think of – `English`, `Chinese`, `Klingon` and more.

- Collections of values are called arrays, and are usually restricted to holding one data type at a time. An array of strings is written as `[String]` and can hold only strings. If you try to put numbers in there, the compiler will emit errors. There is a special data type called `AnyObject` that means, as you might imagine, any data type can be placed inside

- The `try!` keyword is not used frequently, but we're using it here to mean "`I realise calling this code might fail, but I'm certain it won't`". If the code does fail, our app will crash. At the same time, if the code fails it means our app can't read its own data, so something must be seriously wrong, which is why `try!` is OK here.

- You can use for `someVar` in `someArray` to loop through every item in an array. `Swift` pulls out each item and runs the code inside your loop once for each item.

If you're extremely observant you might have noticed one tiny, tiny little thing that also one of the most complicated parts of `Swift`, so I'm going to keep it as simple as possible for now, then expand more over time: it's the exclamation mark at the end of `NSBundle.mainBundle().resourcePath!` No, that wasn't a typo from me. If you take away the exclamation mark the code will no longer work, so clearly `Xcode` thinks it's important – and indeed it is. `Swift` has three ways of working with data:

1. A variable or constant that holds the data. For example, `foo: String` is a string called `foo`.

2. A variable or constant that might hold the data, but we're not sure. This is called an optional type, and looks like this: `foo: String?` You can't use these directly, instead you need to "`unwrap`" them first.

3. A variable or constant that might hold the data, and in fact we're sure it does – at least once it has first been set. This is called an implicitly unwrapped optional, and looks like this: `foo: String!` You can use these directly.

When I explain this to people, they nearly always get optional and implicitly unwrapped optional confused, largely because they aren't all that different. Implicitly unwrapped optionals – the `!`s – serve two purposes: they make code easier to work with, and they make for easier compatibility with Apple's vast collection of existing APIs.

We'll look at optionals in more depth later, but for now what matters is that `NSBundle.mainBundle().resourcePath` may or may not return a string, so what it returns is a `String?` – that is, an optional string. By adding the exclamation mark to the end we are force unwrapping the optional string, which means we're saying, "`I'm sure this will return a real string, it will never be nil, so please just give it to me as a regular string`".

Important warning: if you ever try to use a constant or variable that has a `nil` value, your app will crash. As a result, some people have named `!` the "crash" operator because it's easy to get wrong. The same is true of `try!`, which is also easy to get wrong. Don't worry if this all sounds hard for now – you'll be using it more later, and it make more sense over time.

Before we move on to the next piece of work, there's one small change to make: now you know what `AnyObject`, `String` and arrays are, you should be able to see that at the top of `MasterViewController` there is a line that says `var objects = [AnyObject]()`, which defines our array as containing `AnyObject`. With all the changes we've made, we now know for sure that the objects we're adding will always be strings, so we can change that declaration like so:

```
var objects = [String]()
```

With that change, `Swift` will now make sure we're always putting in and taking out strings from the objects array, which means it's helping keep our code safe. Your project should now build correctly, and if you run it you should see your image names inside the table. Success! Let's make it more interesting…

Introducing Interface Builder

As you saw when you first ran the template app, selecting a date brought in a new screen showing the date all by itself. This was all done with a smooth animation from left to right, and included adding a `Back` button so you can get back to the previous screen. You might also have noticed that you can swipe from the left edge to the right to go back to the table view.

All this behavior was provided for us by two important `iOS` data types: `UISplitViewController` and `UINavigationController`. From those names you can divine two things:

1. The “`UI`” means it's a user interface component designed for `iOS`.
2. The “`Controller`" part means it provides functionality.

Controllers are part of the holy trinity of software development: `Model`, `View`, `Controller`. In an ideal world, every part of your app can be split into one of these three types: it's either a model (something that describes the data you are

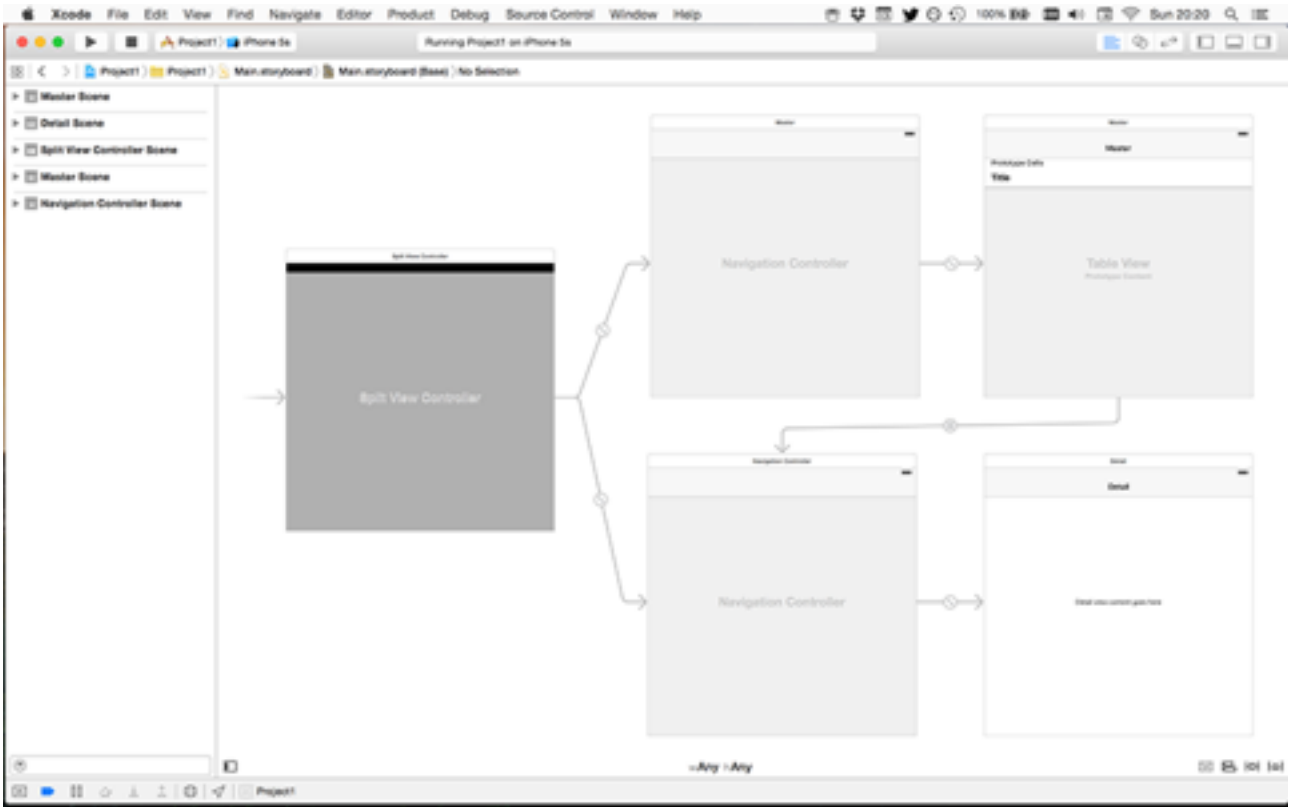
working with), a view (the user interface of your app), or a controller (the code that sends model data to and from the view).

In reality, things are rarely this clean, and it's extremely common to have a `Fat Controller problem`: lots of code that ought to be in models and views ends up in your controller. It's not ideal, but you can always go back later and rewrite the code to be neater. (Spoiler: you will never do this.)

Now, you might be thinking that you haven't written any code using split view controllers or navigation controllers, and you'd be right. This is because `Apple` has a dedicated tool for editing the visual layouts of your apps, and it's called `Interface Builder`. You'll see the file `Main.storyboard` in your project, so select that now to show `Interface Builder`.

`Interface Builder` (also known as the storyboard editor) is designed to show a visual flow of your program. You will need to zoom out, though: your user interface is surprisingly complicated for such a simple app! Hold down `Cmd` and tap the **"–"** key once to zoom out one level. You should now be able to see that there are five squares arranged on the screen: one saying `Split View Controller`, two saying `Navigation Controller`, one saying `Table View` and one saying **"**`Detail view content goes here`**"**.

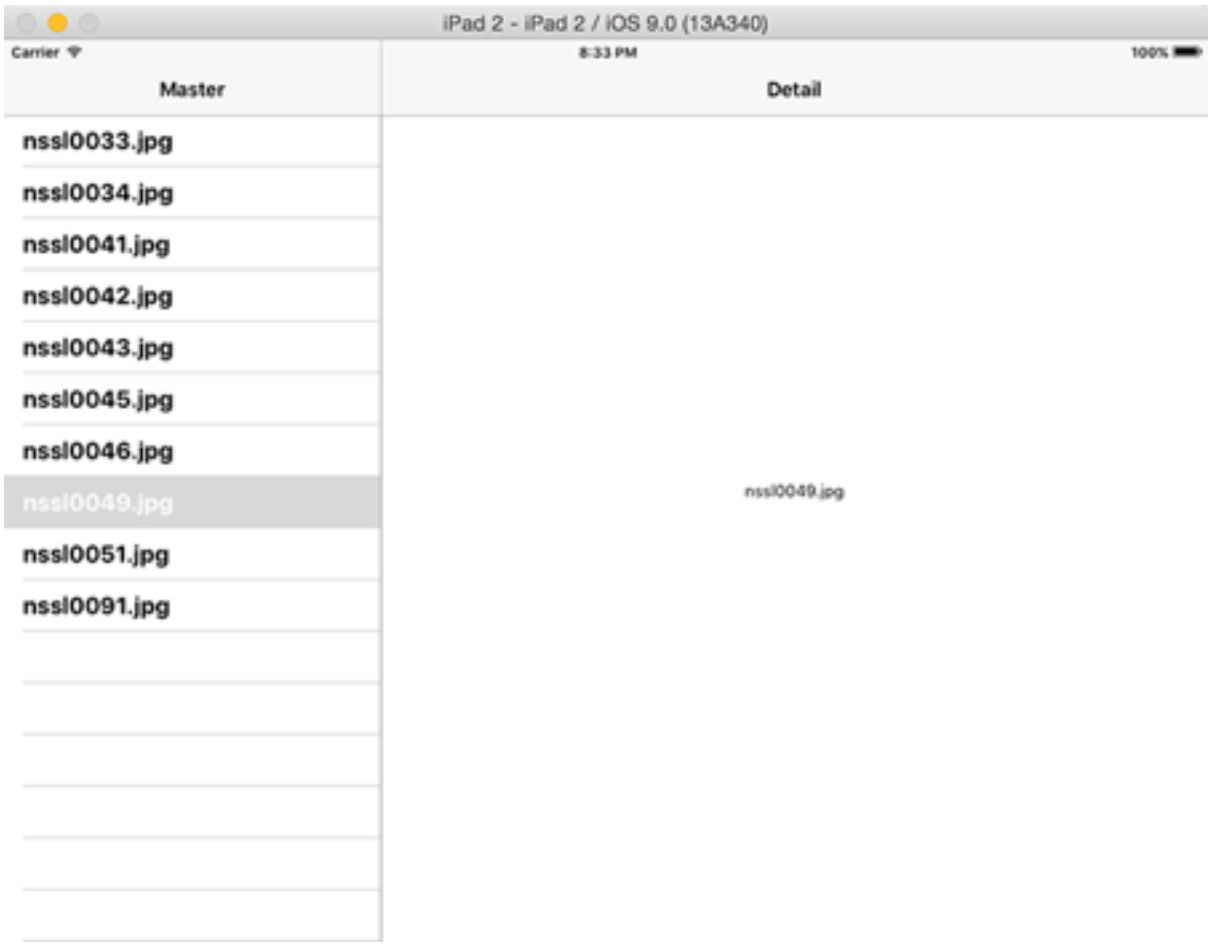
Between each of the squares are arrows moving from left to right, showing the program flow. These are called `segues` (pronounced segways, like the motorised scooter). The segue with two circles connected by a line is called a relationship segue, and it's used to connect the split view controller to both navigation controllers, and also to connect both navigation controllers to the squares immediately to their right.



We'll talk about view controllers in a moment, but for now all you need to know is that these relationship segue describe one screen of content being embedded inside another. So, the relationship segue between the top navigation controller and the table view to its right means that the table view is inside the navigation controller.

Where this gets complicated – and extraordinarily clever – is when you look at the relationship segues between the split view controller and its navigation controllers. It has two navigation controllers as its relationship, which means both are embedded inside it. Can you see them both? Nope. But that's because we asked to use the "`iPhone 5s`" device simulator earlier on.

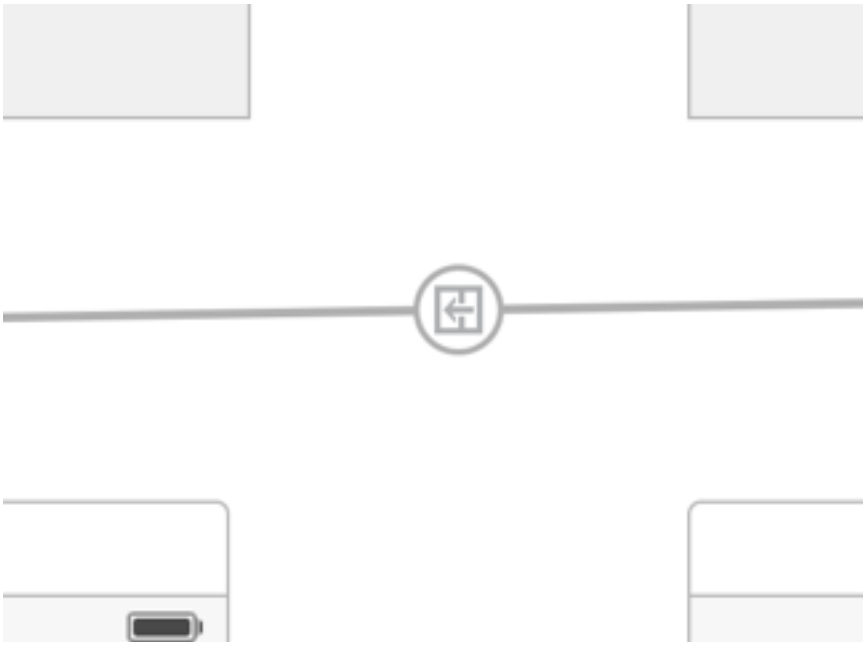
If you change that to be "`iPad 2`" then run it again, yuo'll see something different: in portrait, you'll see a button at the top saying "`Master`" that, when tapped, makes the table of picture names slide in from the left side. If you press `Cmd + Right Arrow` on your keyboard, you'll see both the table view and the detail view at the same time, as the screen automatically splits in two.



This is what Apple calls "adaptive user interfaces", which means you design your app once and iOS automatically determines the best way of showing it on various devices. We've now seen three different ways of seeing the same information: iPhone, iPad portrait and iPad landscape, all from the same code. So, even though it means having a split view controller and two navigation controllers, the end result means that our app looks great on all devices.

Back inside Interface Builder again, you'll notice there's a second kind of segue going from the table view at the top to the navigation controller on the bottom. This segue looks like two rectangles with a left arrow over them, and it's called a "Show Detail" segue, and it creates the behavior you already saw: the detail view controller slides in from the right. It's an adaptive segue (part of the "adaptive user interfaces" way of working) and it means "on iPhone,

animate in from the right, but on iPad just change the detail view controller".



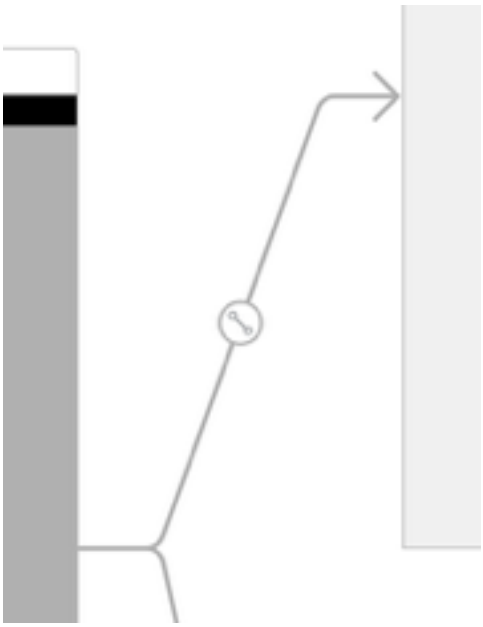
This is a good time to touch on view controllers just briefly, because they are one of the most important components in iOS. The data type UIViewController is used to represent all screens in your app, and comes with a huge array of functionality built in. For example, it can rotate with the device, it can respond if the device is running out of memory, it can show and hide other view controllers, and so on.

View controllers can also be used to represent parts of screens in your app, and this is done in a clever way. As an example, the mail app on iPhone has a view controller to show the messages in your inbox, and when you tap on one it brings in a new view controller to show message detail.

This set up is perfect for the iPhone, where screen space is limited and you can really only do one thing at a time. But on iPad, the same mail app shows the messages on the left and the detail on the right, which is two view controllers at the same time. Behind the scenes, this uses the exact same split view controller you're using here, and iOS just adapts to make sure the right layout is automatically used. Hurray for code reuse!

In our current app, we have five view controllers: the split view controller, two navigation controllers, our master view controller (the table view) and the detail view controller. Each of these is based on the `UIViewController` data type, but add their own functionality on top using a technique known as inheritance – each custom view controller literally inherits the functionality of `UIViewController`, before adding its own.

So, the split view controller is responsible for choosing the right layout depending on the device, the navigation controller adds the title bar at the top, the master controller adds the table view and the code we wrote with `NSFileManager`, and the detail controller has the text showing the date that was selected. You can have multiple levels of inheritance, which is where data type `D` inherits from `C`, which itself inherits from `B`, and `B` in turn inherits from `A`. It might sound a little confusing, but it does mean you get to share as much code as possible.

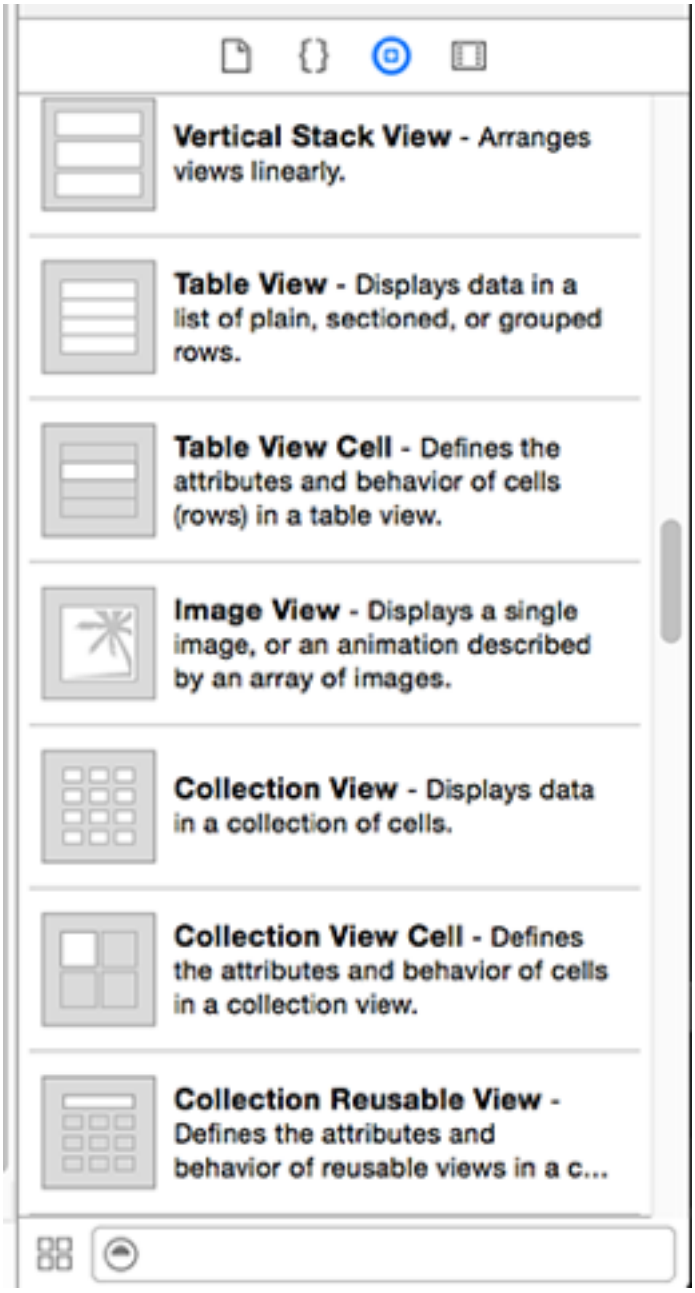


Now, enough thinking – time for some action. We need to change the detail view controller because right now it has some text in the middle for showing dates. We scrapped all that date code and instead want to show images, so we need to redo its user interface. Double-click on the detail view controller to zoom in and select it. In the middle you'll see the text "`Detail view content goes here`" inside a `UILabel`, which is a simple view type that shows text that can't be edited. Select it and press delete.

We're going to replace that label with a large `UIImageView`, which, as you might guess, is a `UI` component that shows images. `iOS` comes with an awesome collection of view types that you can drag and drop into place using storyboards,

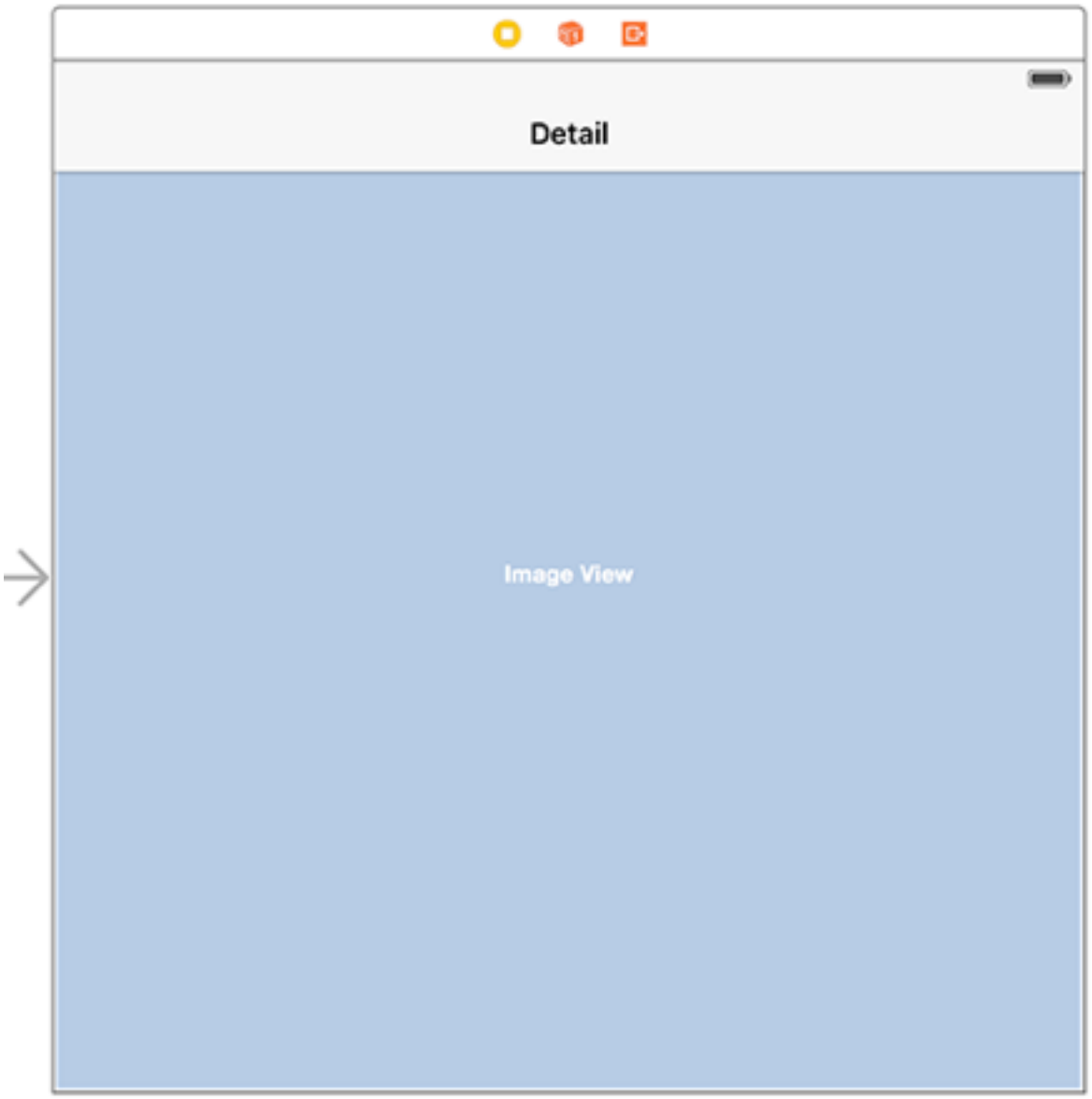
and these are all stored inside the object library. This is usually visible in the bottom-right corner, and starts with “`View Controller`”, “`Storyboard Reference`” and so on. If you can't see it, press `Ctrl + Alt + Cmd + 3` and it should appear.

Some users have the object library set to icon view, which means you'll see just an arrow followed by a series of yellow circles. This isn't helpful for a beginner, so look to the bottom of the object library and you'll see a search box, and directly to the left of that you'll see a button that uses list view instead of icon view.



In the picture below you can see my object library, with the image view selected. At the bottom is the search area where you can filter the list of items, and just to the left of the search field is the icon/list toggle button.

This object library contains all the built-in view types that you can use, and you'll see there a quite a few. You can look at them all you want later on, but for now use the search box: type "`image`" to bring up the `Image View component`. Click and drag the image view from the object library onto the detail view controller, then let go. Now drag its edges so that it fills the entire view controller – yes, even under the gray navigation bar that says "`Detail`".



This image view has no content right now, so it's filled with a pale blue background and the word Image View. We won't be assigning any content to it right now, though – that's something we'll do when the program runs. Instead, we need to tell the image view how to size itself for our screen, whether that's iPhone or iPad.

This might seem strange at first, after all you just placed it to fill the view controller, and it has the same size as the view controller, so that should be it, right? Well, not quite. For a start, the detail view controller you can see on the storyboard is square, and have you ever seen a square iPhone? And what about when you have multiple devices to support: iPhone 4s, iPhone 5s and iPhone 6 all have different sizes, never mind iPad and iPad Pro, so how should the image view respond?

iOS has an answer for this. And it's a brilliant answer that in many ways works like magic to do what you want. It's called Auto Layout: it lets you define rules for how your views should be laid out, and it automatically makes sure those rules are followed. But it has two rules of its own, both of which must be followed by you:

- Your layout rules must be complete. That is, you can't specify only an X position for something, you must also specify a Y position. If it's been a while since you were at school, "X" is position from the left of the screen, and "Y" is position from the top of the screen.

- Your layout rules must not conflict. That is, you can't specify that a view must be 10 points away from the left edge, 10 points away from the right edge, and 1000 points wide. An iPhone 5 screen is only 320 points wide, so your layout is impossible. Auto Layout will try to recover from these problems by breaking rules until it finds a solution, but the end result is never what you want.

You can create Auto Layout rules – known as constraints – entirely inside Interface Builder, and it will warn you if you aren't following the two rules. It will even help you correct any mistakes you make by suggesting fixes!

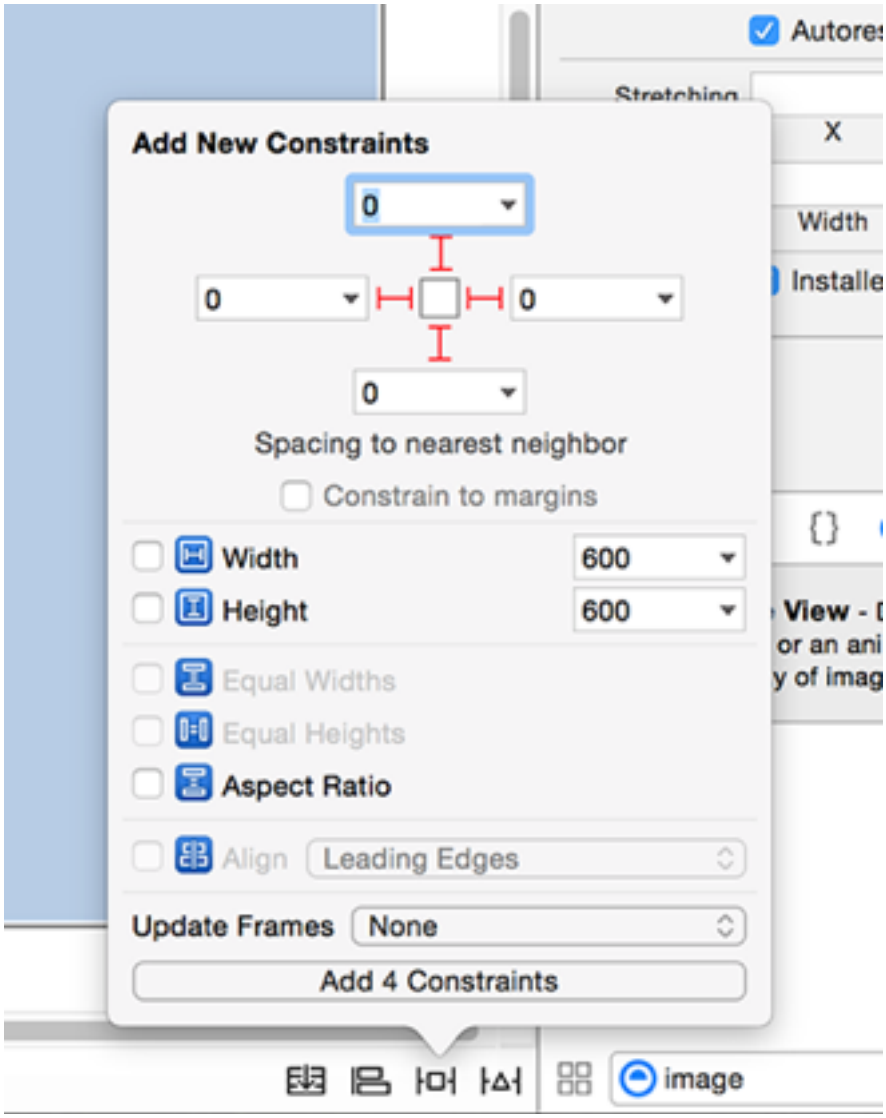
We're going to create four constraints now: one each for the top, bottom, left and right of the image view so that it expands to fill the detail view controller regardless of its size. There are lots of ways of adding Auto Layout constraints, but the easiest way right now is to use the Pin button: a tiny, almost anonymous button in the bottom-right of Interface Builder.

There are four buttons down there: the first has three rectangles with a line pointing down, the second has two rectangles one above the other, the third has a square with lines either side, and the fourth has a triangle with lines either side. All without titles – who said Apple were geniuses at user interface design? The one we want is the Pin button, which is the third one: a square with lines either side. You can hover over it to see the tooltip, which should say Pin.

In the picture below you can see the strip of buttons at the bottom of my Interface Builder view. The button on the left toggles the document outline, and the group of four buttons towards the right control Auto Layout. You want to click the third one of the four on the right.

This Pin menu is what we're going to use now, so make sure your image view is selected then click the Pin button. A popup will appear, with the title Add New Constraints. Deselect the "Constrain to margins" checkbox, then click on the four red dotted lines near the top to turn them into solid red lines.

This will create constraints that set the distance between the image view and its superview – the view that it's sitting inside, in this case our detail view. As you're



clicking these red lines, the button at the bottom of the popup will change to "Add 4 Constraints", so click that now to dismiss the popup and add the constraints.

Visually, your layout will look pretty much identical once you've added the constraints, but there are two subtle differences. First, there's a thin blue line surrounding the UIImageView on the detail view controller, which is Interface Builder's way of showing you that the image view has a correct Auto Layout definition.

Then in the Document Outline pane you'll see a new entry for "Constraints" beneath the image view. If you're not sure where to look, or if your document outline pane is hidden, select the image view then go to the Editor menu and choose

Reveal in Document Outline to show the document outline and highlight the image view. All four constraints that were added are hidden under that Constraints item, and you can expand it to view them individually.

With the constraints added, there's one more thing to do here before we're finished with Interface Builder, and that's to connect our new image view to some code. You see, having the image view inside the layout isn't enough – if we actually want to reference the image view inside code, we need to create a property for it that's attached to the layout.

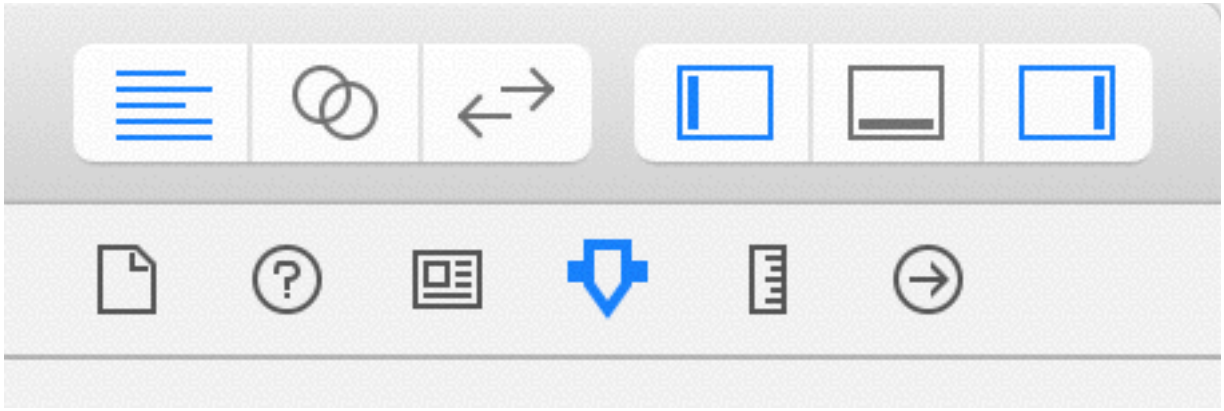
What's a property? Well, you've already met constants (using let) and variables (using var), but these are just temporary so they last only until the method ends. A property is the name given to a constant or variable when it's attached to a data type such as our detail view controller. The image view property we're going to create exists for as long as the detail view controller does, because it belongs to it.

The Xcode template project had a UILabel in the detail view controller, and it had a corresponding property attached. We're going to delete that property, then create a new one.

To make this process a little bit easier, Xcode has a special display layout called the Assistant Editor. This splits your Xcode editor into two: the view you had before on top, and a related view at the bottom. In this case, it's going to show us Interface Builder on top, and the code for the detail view controller below.

Xcode decides what code to show based on what item is selected in Interface Builder, so make sure the image view is selected now and choose View > Assistant Editor > Show Assistant Editor from the menu. You can also use the keyboard shortcut Alt + Cmd + Return to do the same, or click the

assistant editor button. This is the second of six button at the top-right of your



Xcode window, and looks like two overlapping circles.

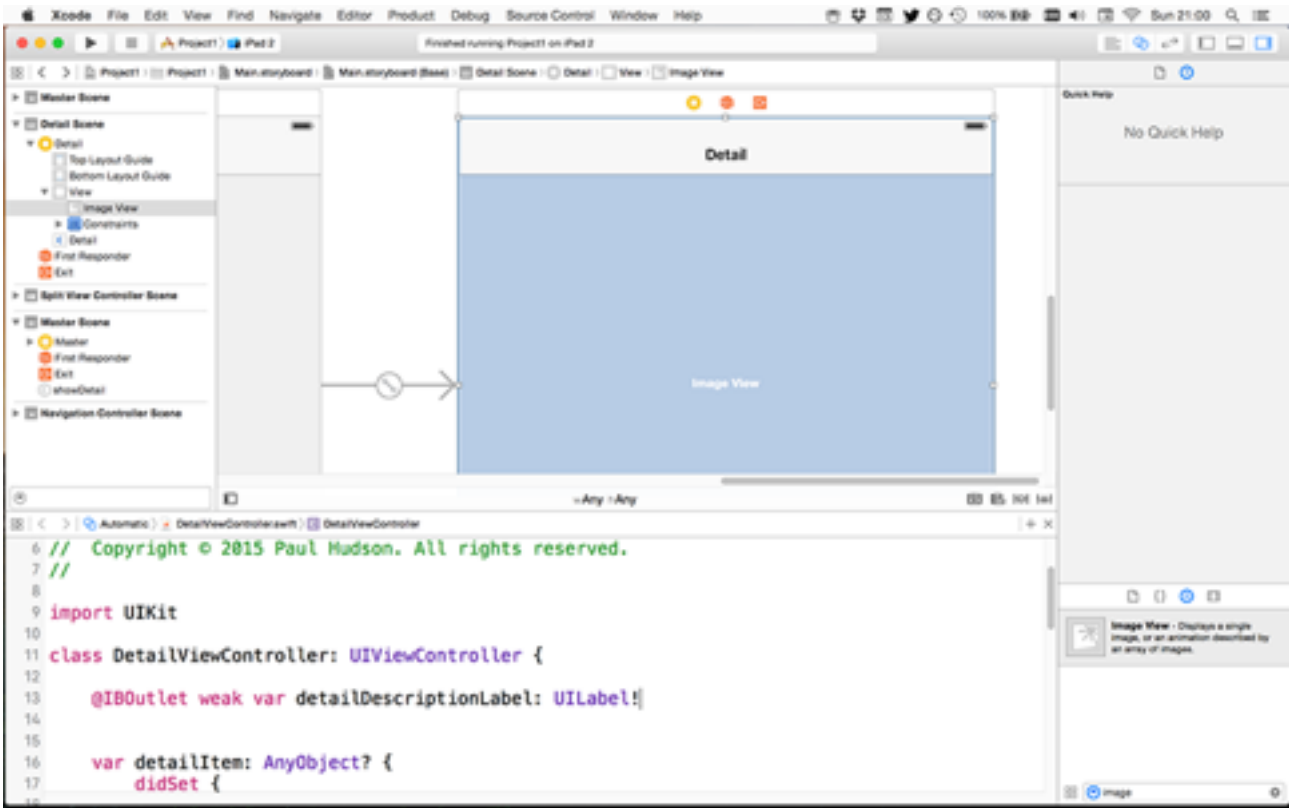
In the picture below, you can see the buttons in the top-right of my Xcode window when I'm in Interface Builder. The first row are the standard editor, the assistant editor, the version editor, then three buttons to control showing and hiding panes on the bottom, left and right of Xcode. The second row shows various Interface Builder inspectors, with the third, fourth and fifth inspectors being the most commonly used.

You should now see the detail view controller in Interface Builder in the top pane, and in the bottom pane the source code for a file we haven't looked at yet: DetailViewController.swift. This is (predictably) the code that manages the detail view controller, and you'll see in there near the top this following line:

```
@IBOutlet weak var detailDescriptionLabel: UILabel!
```

Some bits of that are new, so let's break down the whole line:

- @IBOutlet: This attribute is used to tell Xcode that there's a connection between this line of code and Interface Builder.



- `weak:` This tells `iOS` that we don't want to own the object in memory. This is because the object has been placed inside a view, so the view owns it.
- `var:` This declares a new variable or variable property. We already used let to declare constants, and this is how you declare variables. Remember, a constant's value can be set only once, whereas a variable's value can change.
- `detailDescriptionLabel:` This was the property name assigned to the `UILabel`. Note the way capital letters are used: variables and constants should start with a lowercase letter, then use a capital letter at the start of any subsequent words. For example, `myAwesomeVariable`. This is sometimes called camel case.
- `UILabel!:` This declares the property to be of type `UILabel`, and again we see the implicitly unwrapped optional symbol: `!`. This means that `UILabel` may be there or it may not be there, but we're certain it definitely will be there by the time we want to use it.

If you were struggling to understand implicitly unwrapped optionals (don't worry; they are complicated!), this code might make it a bit clearer. You see, when the detail view controller is created, its view hasn't been loaded yet – it's just some code running on the CPU.

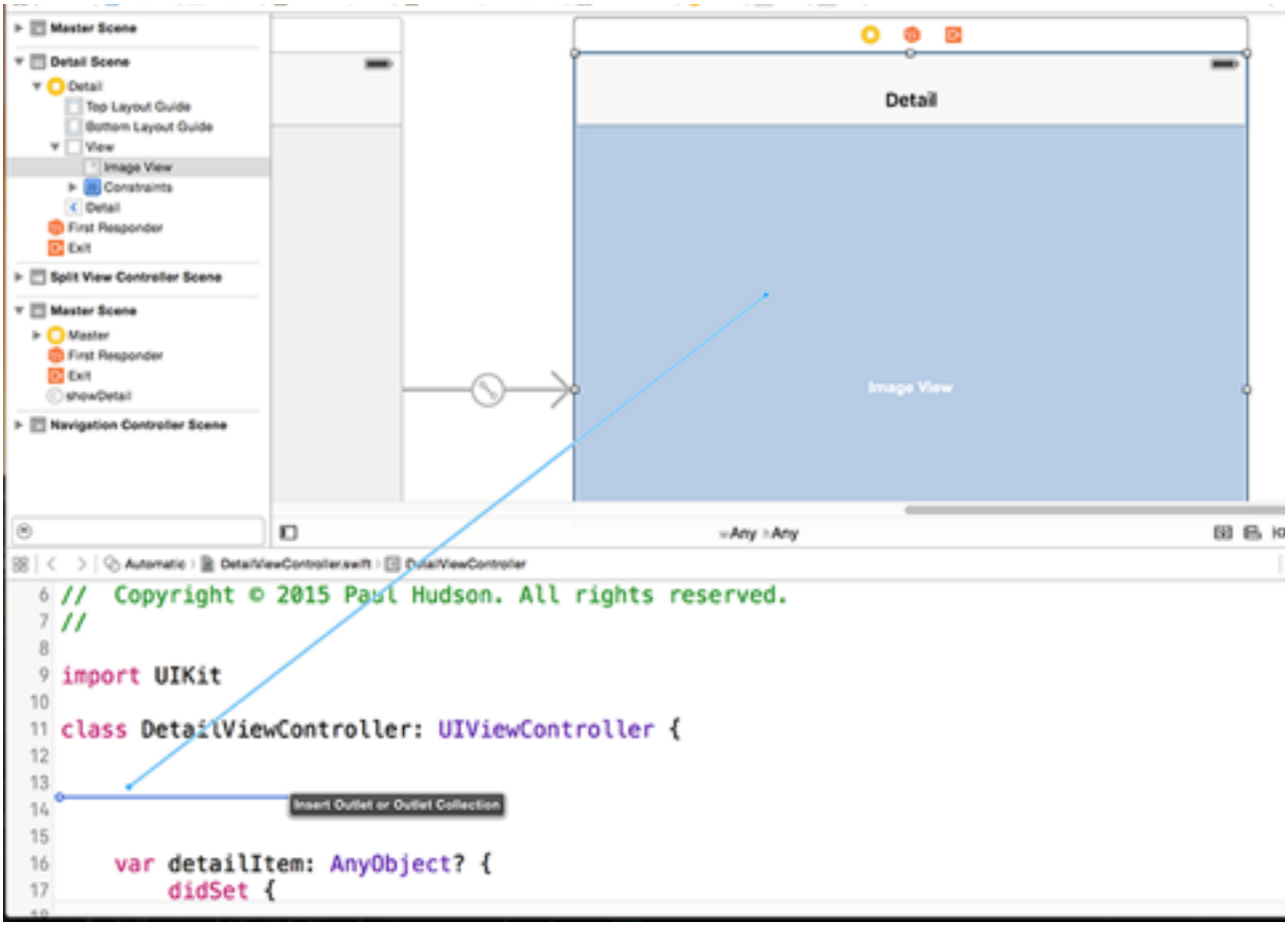
When the basic stuff has been done (allocating enough memory to hold it all, for example), iOS goes ahead and loads the layout from the storyboard, then connects all the IBOutlets from the storyboard to the code.

So, when the detail controller is first made, the UILabel doesn't exist because it hasn't been created yet – but we still need to have some space for it in memory. At this point, the property is nil, or just some empty memory. But when the view gets loaded and the outlet gets connected, the UILabel will point to a real UILabel, not to nil, so we can start using it.

In short: it starts life as nil, then gets set to a value before we can use it, so we're certain it won't ever be nil by the time we want to use it – a textbook case of implicitly unwrapped optionals.

(PS: If you still don't understand implicitly unwrapped optionals, that's perfectly fine – keep on going and they'll become clear over time.)

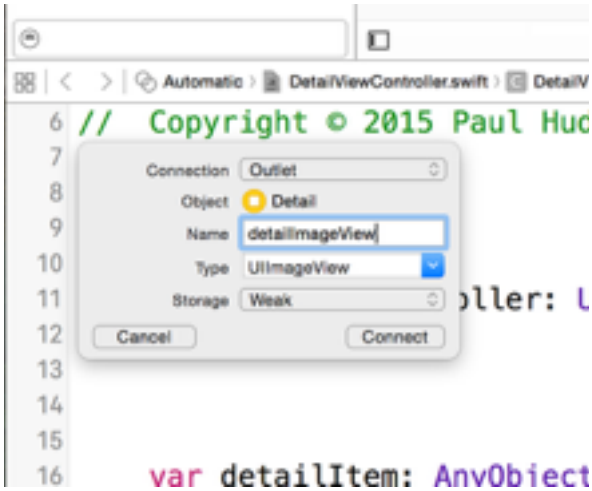
Back to the project: we don't have a UILabel any more, so you can delete that whole line of code. Don't worry: learning all that @IBOutlet stuff wasn't in vain, because we're about to create a new outlet for the image view. This is done, again, visually: hold down Ctrl then click and drag from the UIImageView in the top pane to the where the @IBOutlet line was in the bottom pane. If you've forgotten, it was just below the line that says, class DetailViewController: UIViewController {.



As you're `Ctrl-dragging` from the image view, a blue line will appear. When you move over the code, a tooltip will appear next to your pointer saying "`Insert Outlet or Outlet Collection`", and a second blue line will appear horizontally to show exactly where the outlet will be inserted.

Let go of your mouse button and a popup will appear asking you to fill in some information. Right now, all you need to do is give your outlet a name, so call it `detailImageView` and click `Connect`. You'll see this line of code gets written for you:

```
@IBOutlet weak var detailImageView: UIImageView!
```



Yes, that's mostly the same as the previous code, but now it's an image view rather than a label. Crucially, if you look to the left of this line of code, you'll see a gray circle with a ring around it. This is Xcode's way of telling you that this outlet has a connection in place in `Interface Builder`. If it didn't have a connection, you'd just see an empty ring.

Loading images with UIImage

In making this layout change, we've broken our code – you'll notice there's a red line in the scroll bar of our code editor, and also a red warning symbol at the top of the Xcode window. This is because other parts of `DetailViewController.swift` reference the `UILabel`, and we've just deleted that property. So we need to make a few changes to this file so that it knows how to handle our new data, then make some final changes to `MasterViewController.swift` in order to pass the new data correctly.

First, look for this code in `DetailViewController.swift`:

```
var detailItem: AnyObject? {
    didSet {
        // Update the view.
        self.configureView()
    }
}
```

This declares a property called detailItem, giving it the type `AnyObject?` – that's `Swift's` way of saying it might be any kind of object, or it might nothing at all. But this property has a twist, because it has a property observer attached, in the form of `didSet`.

The `didSet` property observer is a block of code that will be executed any time this property's value has been changed. There's also a corresponding `willSet` property observer that can execute code just before a property is changed, but it's not used as often as `didSet`.

In this code, `didSet` is being used to call `self.configureView()`, which means `“call the configureView() method on myself”`. The `“self.”` isn't actually needed, so you can delete it if you want. You might be interested to know that there are two trains of thought with regards to using `“self.”` when referring to variables and methods.

The first group of people never like to use self. unless it's required, because when it's required it's actually important and meaningful, so using it in places where it isn't required can confuse matters. The other group of people always like to use self. whenever it's possible, even when it's not required. To be fair to this group, using self. everywhere was good practice in `Objective-C`, and it can be a hard habit to break.

Before we fix the errors in our code, we're going to introduce some more. You see, detailItem is defined as `AnyObject?`, which is silly. We know exactly what kind of data is going to be sent across from the master view controller, because we're working with an array of strings. The only thing detailItem can be is a `String`, albeit an optional one.

So, change AnyObject? to be String?, then press Cmd + B to compile your project. This creates another error, but that's OK because we're about to fix it!

The error will be in the configureView() method of DetailViewController.swift. Here's its current code:

```
if let detail: AnyObject = self.detailItem {
    if let label = self.detailDescriptionLabel {
        label.text = detail.description
    }
}
```

The detailItem property is an optional data type: it was AnyObject? and now it's String?. This means we need to unwrap the value before we can use it – we need to check whether there is a value there, and, if so, extract it. The most common way to extract values from optionals is to use an "if let" condition, which is what this current code is doing. For example:

```
if let foo = bar {
    doStuff(foo)
}
```

That code means "see if the bar variable contains a value, and if it does unwrap that value and place it inside another variable called foo, then call doStuff() and pass in foo as a parameter". So, if bar was of type String?, foo will be of type String because it won't be optional any more. If bar didn't have a value, then doStuff() would never be called. The advantage of this "if let" syntax is that it both checks for the existence of a value and unwraps it all it once.

With all that in mind, let's take a look at the code again:

```
if let detail: AnyObject = self.detailItem {
    if let label = self.detailDescriptionLabel {
        label.text = detail.description
    }
}
```

You'll see that it first checks whether `detailItem` has a value, and if so it unwraps it into a new constant called detail. It then checks whether `detailDescriptionLabel` has a value, and if so it unwraps it into another new constant called label. If both of these had values and were unwrapped, then the label's text property is set to be detail.description. In the case of the dates that were originally in there, this would print the date as text. These lines are erroring because we don't have a label any more, so please delete them and replace them with this:

```
if let detail = self.detailItem {
    if let imageView = self.detailImageView {
        imageView.image = UIImage(named: detail)
    }
}
```

This changes a few things:

- We no longer need to declare the detail constant as `AnyObject`, because `self.detailItem` is of type `String?` not `AnyObject?`.

- We unwrap the property `detailImageView` into `imageView`, because we have an image view rather than a label now.

- Rather than setting the text of the label, we're setting the image of the image view.

This new code also introduces a new data type, called `UIImage`. This doesn't have "`View`" in its name like `UIImageView` does, so it's not a view – it's not something that's actually visible to users. Instead, `UIImage` is the data type you'll use to load image data, such as `PNG` or `JPEGs`.

When you create a `UIImage`, it takes a parameter called `named` that lets you specify the name of the image to load. `UIImage` then looks for this filename in your app's bundle, and loads it. By passing in the detail constant here, which was created by unwrapping `detailItem`, this will load the image that was selected by the user.

That's the only error inside `DetailViewController.swift`, but you're probably curious how the selected image name gets passed from the master view controller to the detail. To find out, go to `MasterViewController.swift`. If you're still using the assistant editor view, switch back to the standard editor by pressing `Cmd + Return`, by clicking the first of the six buttons in the top-right of the `Xcode` window, or by choosing the `View > Standard Editor > Show Standard Editor` menu item.

The image name is passed from `MasterViewController.swift` inside the `prepareForSegue()` method. This gets called when a segue is about to execute, and gives you a chance to configure the new view controller with information for it to use. In our project, this is where we tell the detail view controller what item was selected, and it's done using this existing code:

```
override func prepareForSegue(segue: UIStoryboardSegue,
sender: AnyObject?) {
```

```
    if segue.identifier == "showDetail" {
        if let indexPath =
self.tableView.indexPathForSelectedRow {
            let object = objects[indexPath.row]
            let controller = (segue.destinationViewController
as! UINavigationController).topViewController as!
DetailViewController
            controller.detailItem = object
            controller.navigationItem.leftBarButtonItem =
self.splitViewController?.displayModeButtonItem()

controller.navigationItem.leftItemsSupplementBackButton =
true
        }
    }
}
```

This is pretty complicated code, but only these three lines are important:

```
let object = objects[indexPath.row]
let controller = (segue.destinationViewController as!
UINavigationController).topViewController as!
DetailViewController
controller.detailItem = object
```

To make it easier to understand, we're going to rewrite it. Replace those three with these:

```
let navigationController = segue.destinationViewController
as! UINavigationController
```

```
let controller = navigationController.topViewController as!
DetailViewController
controller.detailItem = objects[indexPath.row]
```

The first line looks for the destination view controller of the segue, which is where the "`show detail`" segue pointed to. If you remember, that was the navigation controller, so this line of code forces that to be treated as a navigation controller using `as! UINavigationController`. This means "`I know you think that the destination view controller is a regular UIViewController, but trust me: it's actually a UINavigationController`".

This is required because when `prepareForSegue()` is called the new view controller that is about to be shown has already been created, and it's our job to customise it with any data it needs. We can access the new view controller by reading the `segue.destinationViewController` property, but the problem is that it could be any kind of view controller – `Swift` knows that it's definitely some sort of `UIViewController`, but has no idea that behind the scenes it's actually an instance of `UIViewController`.

Sometimes this doesn't matter. If you just want to show something and don't care to modify its values, you don't need to do any typecasting or indeed do any of this code – the segue will just happen. But `prepareForSegue()` exists specifically so that we can do any customization before the new view controller is shown, and in our case that means setting a property on the detail view controller.

`We need to typecast destinationViewController to be UINavigationController` because of what happens in line two: a navigation controller has a property called `topViewController` that points to whichever view controller it is showing right now. For us that's our detail view controller, but

iOS doesn't know that – it thinks `topViewController` is just a regular `UIViewController`. So we need to use `as!` again to override this: we're telling Swift that this object is definitely our `DetailViewController` data type.

Finally, once we've found the navigation controller (line 1), found the detail view controller inside it (line 2), we can update the `detailItem` property of that detail view controller to be the image that was selected. This is done by setting its `detailItem` property to be `objects[indexPath.row]`.

The objects property, you might recall, is an array of strings that we loaded from the app bundle. Arrays are an ordered collection of values, one after the other, and you can read them out by specifying their position in the array counting from `0`. So, the first item in the objects array is `objects[0]`, the second is `objects[1]`, the tenth is `objects[9]` and so on.

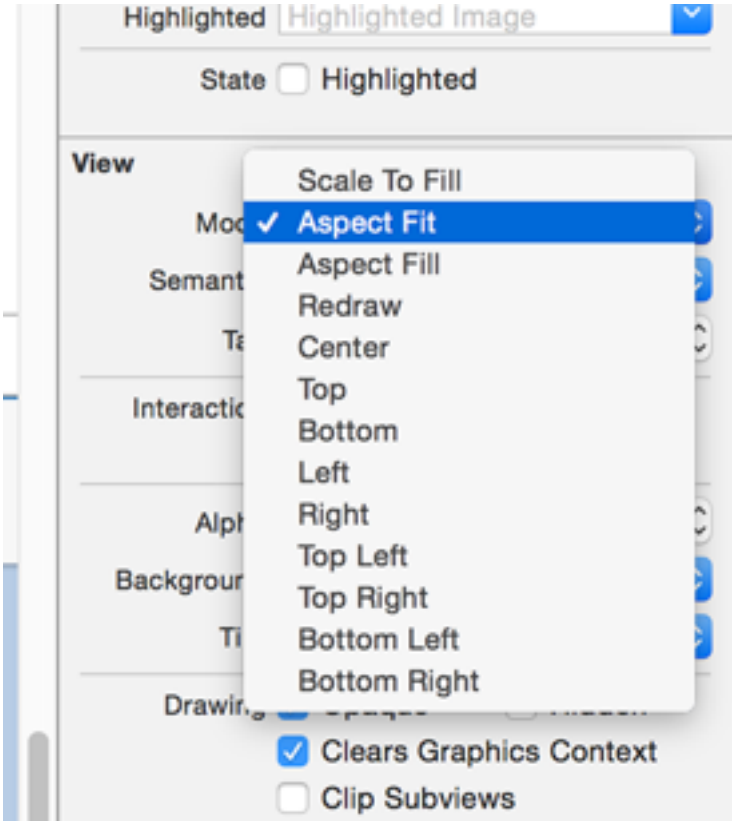
So what's `indexPath.row`? Well, `indexPath` is set a couple of lines above to be whatever row is selected in the table – i.e., the row that the user just tapped to trigger the segue. The row property of that `indexPath` is the position inside the table, so by using `objects[indexPath.row]` we're saying "`get the item in the objects array at the position where the user just tapped`".

Final tweaks: hidesBarsOnTap

At this point you have a working project: you can press `Cmd + R` to run it, flick through the images in the table, then tap one to view it. But before this project is complete, there are two other small changes we're going to make that makes the end result a little more polished.

First, you might have noticed that all the images are being stretched to fill the screen. This isn't an accident – it's the default setting of `UIImageView`. This takes just a few clicks to fix: choose `Main.storyboard`, select the image view in the detail view controller, then choose the Attributes Inspector. This is in the right-hand pane, near the top, and is the fourth of six inspectors, just to the left of the ruler icon.

If you don't fancy hunting around for it, just press `Cmd + Alt + 4` to bring it up. The stretching is caused by the view mode, which is a dropdown button that defaults to "`Scale to Fit`". Change that to be "`Aspect Fit`", and this first problem is solved.



If you were wondering, `Aspect Fit` sizes the image so that it's all visible. There's also `Aspect Fill`, which sizes the image so that there's no space left blank – this usually means cropping either the width or the height. If you use `Aspect Fill`,

the image effectively hangs outside its view area, so you should make sure you enable `Clip Subviews` to avoid the image overspilling.

The second change we're going to make is to allow users to view the images fullscreen, with no navigation bar getting in their way. There's a really easy way to make this happen, and it's a property on `UINavigationController` called `hidesBarsOnTap`. When this is set to true, the user can tap anywhere on the current view controller to hide the navigation bar, then tap again to show it.

Be warned: you need to set it carefully when working with iPhones. If we had it set on all the time then it would affect taps in the table view, which would cause havoc when the user tried to select things. So, we need to enable it when showing the detail view controller, then disable it when hiding.

You already met the method `viewDidLoad()`, which is called when the view controller's layout has been loaded. There are several others that get called when the view is about to be shown, when it has been shown, when it's about to go away, and when it has gone away. These are called, respectively, `viewWillAppear()`, `viewDidAppear()`, `viewWillDisappear()` and `viewDidDisappear()`. We're going to use `viewWillAppear()` and `viewWillDisappear()` to modify the `hidesBarsOnTap` property so that it's set to true only when the detail view controller is showing.

Open `DetailViewController.swift`, then add these two new methods directly below the end of the `viewDidLoad()` method:

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    navigationController?.hidesBarsOnTap = true
}
```

```
override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated)
    navigationController?.hidesBarsOnTap = false
}
```

There are some important things to note in there:

- We're using override for each of these methods, because they already have defaults defined in `UIViewController` and we're asking it to use ours instead. Don't worry if you aren't sure when to use override and when not, because if you don't use it and it's required `Xcode` will tell you.

- Both methods have a single parameter: whether the action is animated or not. We don't really care in this instance, so we ignore it.

- Both methods use the super prefix: `super.viewWillAppear()` and `super.viewWillDisappear()`. This means "`tell my parent data type that these methods were called`". In this instance, it means that it passes the method on to `UIViewController`, which may do its own processing.

- `All view controllers have an optional property called navigationController`, which, if set, lets us reference the navigation controller we are inside. It's optional because not all view controllers are inside a navigation controller. In `Swift` you can use a question mark inside a statement to evaluate an optional, and execution of the line will continue only if the optional could be unwrapped. So, if we're not inside a navigation controller, the `hidesBarsOnTap` lines will do nothing.

If you run the app now, you'll see that you can tap to see a picture full size, and it will no longer be stretched. While you're viewing a picture you can tap to hide the navigation bar at the top, then tap to show it again. We're done!

Wrap up

This has been a very simple project in terms of what it can do, but you've also learned a huge amount about `Swift`, `Xcode` and `storyboards`. I know it's not easy, but trust me: you've made it this far, so you're through the hardest part.

To give you an idea of how far you've come, here are just some of the things we've covered: constants and variables, method overrides, table views and image views, app bundles, NSFileManager, typecasting, arrays, loops, optionals, view controllers, storyboards, outlets, Auto Layout, UIImage and more.

Yes, that's a huge amount, and to be brutally honest chances are you'll forget half of it. But that's 0K, because we all learn through repetition, and if you continue to follow the rest of this series you'll be using all these and more again and again until you know them like the back of your hand.

If you want to spend a little more time on this app, try investigating the title property of your two view controllers. This lets you customise the text that appears in the navigation bar at the top when the view controller is being shown – it's easy to make this show the name of the image that was selected.