Project 10: Names to Faces with UICollectionView

Overview

`Brief`: Get started with `UICollectionView` and the photo library.
`Learn`: UICollectionView, UIImagePickerController, NSUUID, custom classes.

- Setting up
- Designing UICollectionView cells
- Data sources and delegates: UICollectionViewDataSource
- Importing photos with UIImagePickerController
- Custom subclasses of NSObject
- Connecting up the people
- Wrap up

Setting up

This is a fun, simple and useful project that will let you create an app to help store names of people you've met. If you're a frequent traveller, or perhaps just bad at putting names to faces, this project will be perfect for you.

And yes, you'll be learning lots along the way: this time you'll meet `UICollectionViewController`, `UIImagePickerController` and `NSUUID`. Plus you'll get to do more with your old pals `CALayer`, `UIAlertController`, `NSData` and closures. But above all, you're going to learn how to make a new data type from scratch for the first time.

Create a new `Single View Application` project in Xcode, call it `Project10`, set its target to any device you want, then save it somewhere. This should be second nature to you by now – you're becoming a veteran!

Designing UICollectionView cells

Open `Interface Builder` with `Main.storyboard`, then embed the initial view controller inside a navigation controller. Now, using the object library (`Ctrl + Alt + Cmd + 3`) search for a `Collection View` and drag it onto the view controller so that it takes up the full space. **`Important`**: don't choose the one with a yellow icon, because that's a `Collection View Controller` rather than a `UICollectionView`. They are different!

With the collection view selected, go to the size inspector and set `Cell Size` to have the width `140` and height `180`. Now set the section insets for top, bottom, left and right to all be `10`.

Collection views are extremely similar to table views, with the exception that they display as grids rather than as simple rows. But while the display is different, the underlying method calls are so similar that you would be able to dive right in if it weren't for the fact that we need to write all the code ourselves this time – there's no collection view template code we can modify!

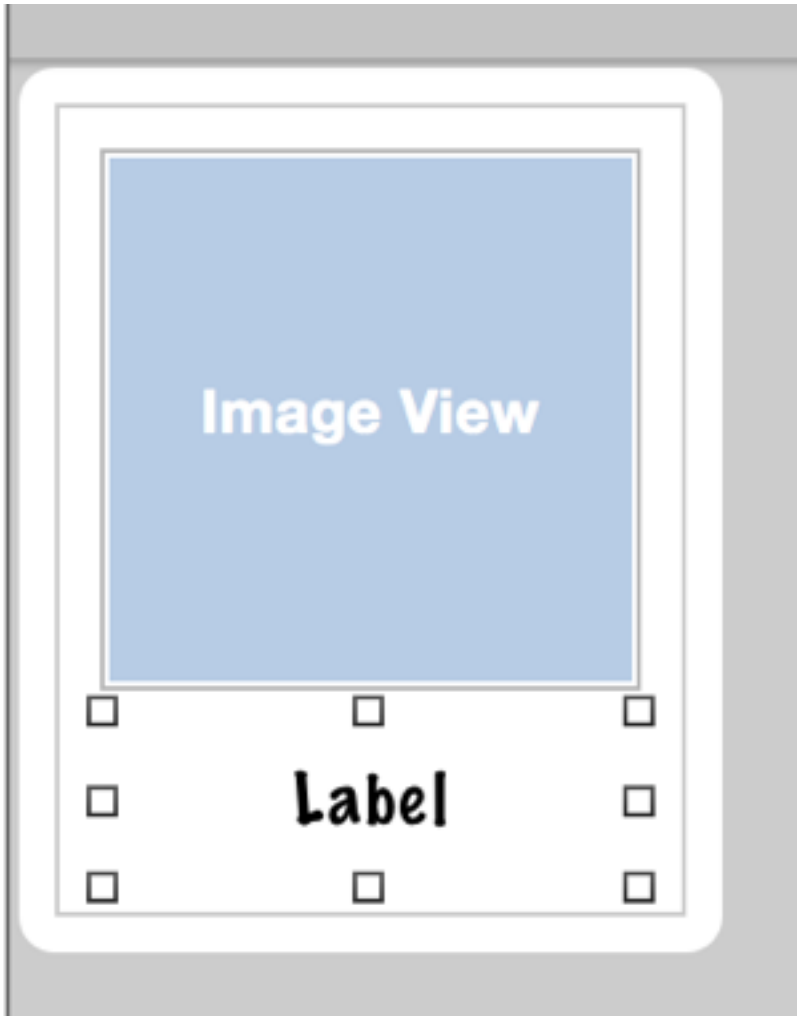
Using the assistant editor, create an outlet for the collection view called `collectionView`, then return back to the standard editor. Trying to edit a storyboard while you can only see half the screen isn't easy!

When you create a collection view, you get one initial collection view cell defined for you, called a *`prototype`*. This is the empty square you'll see in the top-left corner.

This works the same as with table views; you'll remember we changed the initial cell in `Project 7` so that we could add subtitles.

Select that collection view cell now, then go to the attributes inspector and change its `Background` from `"Default"` (`transparent`) to `white`. Now place a `UIImageView` in there, with `X:10`, `Y:10`, width `120` and height `120`. We'll be using this to show pictures of people's faces.

Place a `UILabel` in there too, with `X:10`, `Y:134`, width `120` and height `40`. In the attributes inspector, change the label's font by clicking the `T` button and choosing `"Custom"` for font, `"Marker Felt"` for family, and `"Thin"` for style. Give it the font size `16`, which is `1` smaller than the default, then set its alignment to be centered and its number of lines property to be `2`.



So far this has been fairly usual storyboard work, but now we're going to do two things we've never done before: assign a delegate from a storyboard, then create a custom class for our collection view cell. The first is simple: select the collection view (not the cell!) by clicking the large empty area, then Ctrl–drag from there up to where it says "View Controller" in the document outline on the left.

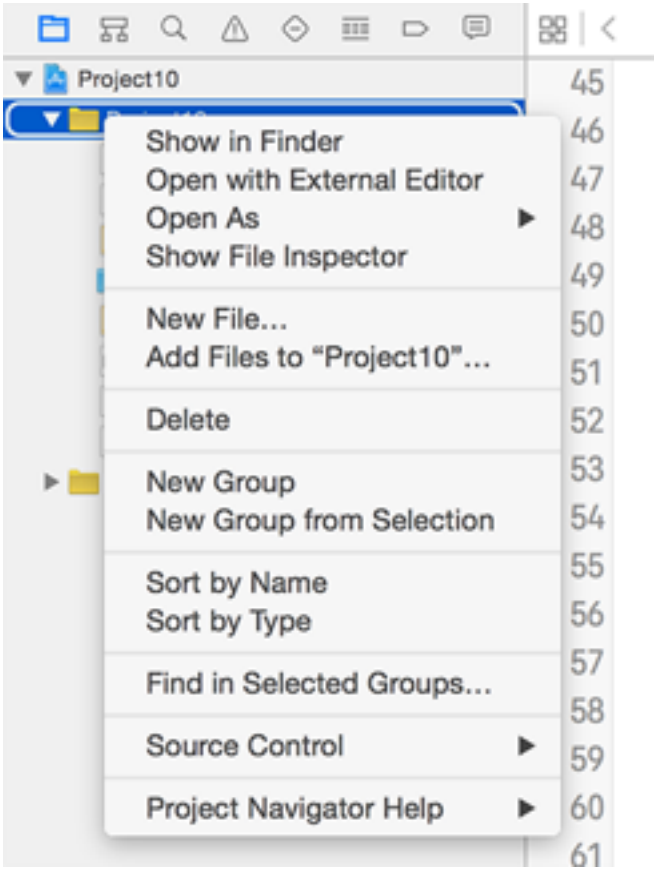
When you let go, you'll be asked what kind of connection you want to make, and you'll see two options you haven't seen before: dataSource and delegate. I want you to select both, which means you'll need to Ctrl–drag twice.

What you've just done is tell the collection view that your ViewController class (the one Xcode made for you as part of its template) will provide all the data for the collection view (it's data source) and will also respond to any interesting events from the collection view (its delegate). We'll need to return to this some more soon.

Now for something harder: we need to create a custom class for our collection view cell. This is because our collection view cell has two views that we created – the image view and the label – and we need a way to manipulate this in code. The shortcut way would be to give them unique tags and give them variables when the app runs, but we're going to do it The Proper Way this time so you can learn.

Read this carefully, because you'll need to do it a lot from here on. In the project navigator, the top thing is a blue project icon that says "Project10 / 2 targets". Beneath that is a yellow folder that also says Project10. Right-click on that and choose "New File…". Whenever I say to create a new file in later projects, please follow this exact procedure.

In the picture below you can see how adding a new file should look: right-click on the yellow folder that says Project10 (or whatever you named your project) then look for New File in the popup menu.



You'll be asked to choose a template for your new file. From the left, make sure "Source" is selected under the "iOS" heading, then choose Cocoa Touch Class and click Next. You'll be asked to fill in two text fields: where it says "Subclass of" you should enter "UICollectionViewCell", and where it says "Class" enter "PersonCell". Click Next then Create, and Xcode will create a new class called PersonCell that inherits from UICollectionViewCell.

This new class needs to be able to represent the collection view layout we just defined in Interface Builder, so it just needs two outlets. Modify the class to this:

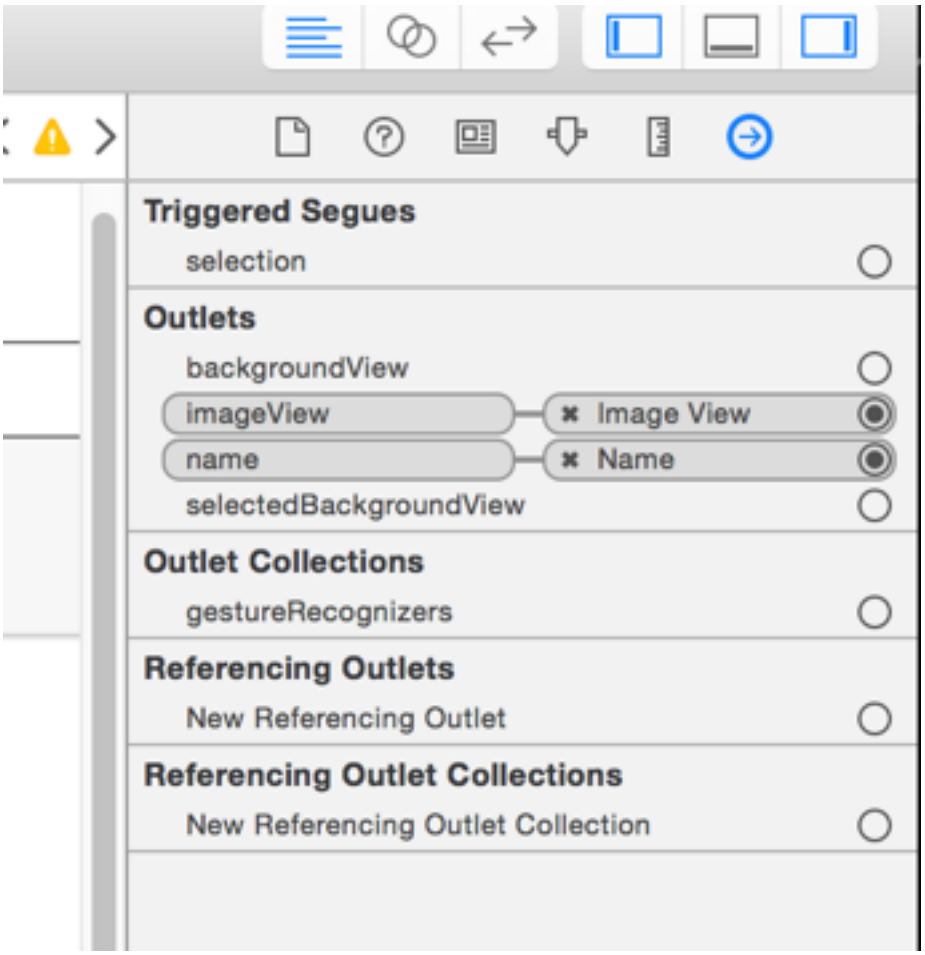
```
@IBOutlet weak var imageView: UIImageView!
@IBOutlet weak var name: UILabel!
```

Now go back to Interface Builder and select the collection view cell in the document outline. Select the identity inspector (Cmd + Alt + 3) and you'll see next to Class the word "UICollectionViewCell" in gray text. That's telling us that the cell is its default class type.

We want to use our custom class here, so enter `"PersonCell"` and hit return. You'll see that `"PersonCell"` now appears in the document outline. While you're there, go to the attributes inspector and give the cell the identifier `"Person"`.

Now that `Interface Builder` knows that the cell is actually a `PersonCell`, we can connect its outlets. Go to the connections inspector (it's the last one, so `Alt + Cmd + 6`) with the cell selected and you'll see `imageView` and name in there, both with empty circles to their right. That empty circle has exactly the same meaning as when you saw it with outlets in code: there is no connection between the storyboard and code for this outlet.

To make a connection from the connections inspector, just click on the empty circle next to `imageView` and drag a line over the view you want to connect. In our case, that means dragging over the image view in our custom cell. Now connect name to the label, and you're almost done with the storyboard.



The last thing to do is make sure the collection view stays edge to edge with the view controller regardless of device. With the collection view selected, click the `Pin button`, which is the third of four buttons in the bottom-right corner of the `Interface Builder pane`.

Deselect `“Constrain to margins”` then click all four dotted red lines near the top of the popup and click `“Add 4 Constraints”`. We did this in `Project 1`, but I appreciate that was some time ago!

Enough storyboarding: time for some code…

Data sources and delegates: UICollectionViewDataSource

In `Project 4` we made ourselves the delegate of a `WKWebView's` navigation, and as soon as that happened there were compiler problems. We also had to tell `Swift` that we conformed to the `WKNavigationDelegate` protocol in order to make the code work.

`WKNavigationDelegate` is a very easy protocol to conform to, because all its methods are optional. That means you don't need to implement anything, you just need to say that you conform to the protocol.

When you make your view controller to be the data source and delegate of a collection using a storyboard, you won't get any compiler problems, but the code won't work. You don't get any compiler problems because the compiler doesn't check storyboard connections, but the code won't work because the `UICollectionViewDataSource` protocol has two non-optional methods that we need to implement.

The best next step from here is to tell Swift in code that your view controller conforms to the UICollectionViewDataSource and UICollectionViewDelegate protocols, because that way it can make sure your code is valid. So, go to ViewController.swift and modify your view controller's class definition to this:

```
class ViewController: UIViewController,
UICollectionViewDataSource, UICollectionViewDelegate {
```

Once you've made this change, you'll see compiler errors because we don't fully conform to the UICollectionViewDataSource protocol. Specifically, you must tell the collection view how many items of data it should expect and what each item should contain. To begin with, let's put together the most basic implementation that allows our code to build cleanly. Add these two methods:

```
func collectionView(collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
    return 10
}

func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {
    let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Person
", forIndexPath: indexPath) as! PersonCell
    return cell
}
```

We haven't looked at any of this code before, so I want to pull it apart in detail before continuing:

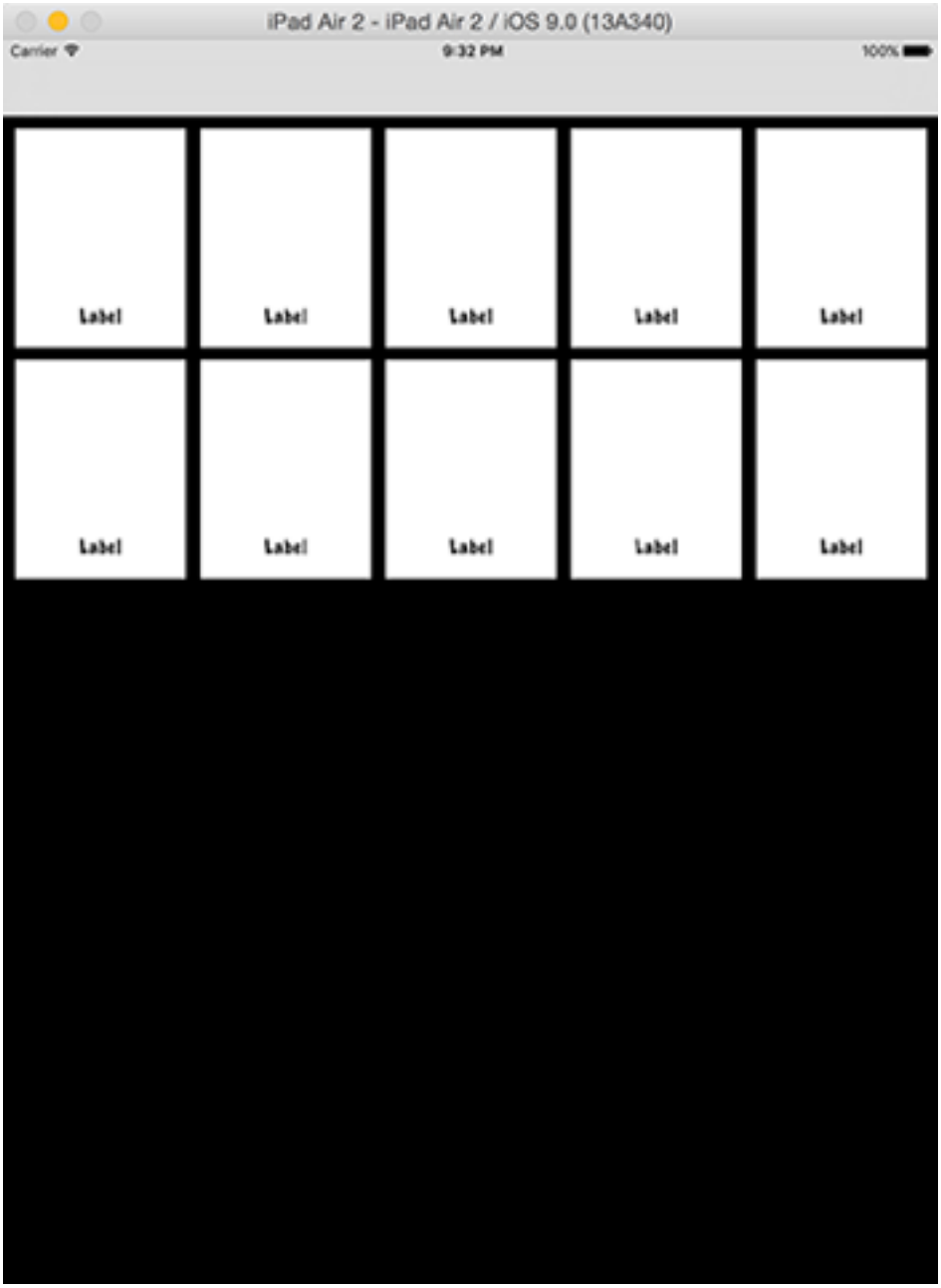
- `collectionView(_:numberOfItemsInSection:)` This must return an integer, and tells the collection view how many items you want to show in its grid. I've returned `10` from this method, but soon we'll switch to using an array.
- `collectionView(_:cellForItemAtIndexPath:)` This must return an object of type `UICollectionViewCell`. We already designed a prototype in `Interface Builder`, and configured the `PersonCell` class for it, so we need to create and return one of these.
- `dequeueReusableCellWithReuseIdentifier()` This crazy long method name does something really important: it creates a collection view cell using the reuse identified we specified, in this case `"Person"` because that was what we typed into `Interface Builder` earlier. But even better, this method will automatically try to reuse collection view cells, so as soon as a cell scrolls out of view it can be recycled so that we don't have to keep creating new ones.

Note that we need to typecast our collection view cell as a `PersonCell` because we'll soon want to access its `imageView` and name outlets.

These two new methods both come from the `UICollectionViewDataSource` protocol, but both are remarkably similar to the `UITableViewDataSource` delegate that is the equivalent for table views – you can go back and open project 1 again to see just how similar!

Press `Cmd + R` to run your project now, and you'll see the beginning of things start to come together: the prototype cell you designed in `Interface Builder` will appear `10` times, and you can scroll up and down to view them all. As you'll see, you can fit two cells across the screen, which is what makes the collection view different to the table view. Plus, if you rotate to landscape you'll see it

automatically (and beautifully) animates the movement of cells so they take up the full width.



Importing photos with UIImagePickerController

There are lots of events that make up the `UICollectionViewDelegate` protocol to handle when the user interacts with a cell, but we'll come back to that

later. For now, let's look at how to import pictures using UIImagePickerController. This new class is designed to let users select an image from their camera to import into an app. When you first create a UIImagePickerController, iOS will automatically ask the user whether the app can access their photos.

First, we need to create a button that lets users add people to the app. This is as simple as putting the following into the viewDidLoad() method:

```
navigationItem.leftBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Add, target: self,
action: #selector(addNewPerson))
```

The addNewPerson() method is where we need to use the UIImagePickerController, but it's so easy to do I'm just going to show you the code:

```
func addNewPerson() {
    let picker = UIImagePickerController()
    picker.allowsEditing = true
    picker.delegate = self
    presentViewController(picker, animated: true,
completion: nil)
}
```

There are only two interesting things in there. First, we set the allowsEditing property to be true, which allows the user to crop the picture they select. Second, when you set self as the delegate, you'll need to conform not only to the UIImagePickerControllerDelegate protocol, but also the UINavigationControllerDelegate protocol.

The first of those protocols is useful, telling us when the user either selected a picture or cancelled the picker. The second, UINavigationControllerDelegate, really is quite pointless here, so don't worry about it beyond just modifying your class declaration to include the protocol.

When you conform to the UIImagePickerControllerDelegate protocol, you don't need to add any methods because both are optional. But they aren't really – they are marked optional for whatever reason, but your code isn't much good unless you implement them! Let's tackle them individually, starting with imagePickerControllerDidCancel(). Add this to your class:

```
func imagePickerControllerDidCancel(picker:
UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
```

So, if the user cancels the image picker, we dismiss it. This is required because the default UIImagePickerController behaviour is to take up the full screen, so we need to hide it to return to our view controller.

The much more complicated delegate method is imagePickerController(_, didFinishPickingMediaWithInfo:), which returns when the user selected an image and it's being returned to you. This method needs to do several things:

- Extract the image from the dictionary that is passed as a parameter.
- Generate a unique filename for it.
- Convert it to a JPEG, then write that JPEG to disk.
- Dismiss the view controller.

To make all this work you're going to need to learn a few new things.

First, it's very common for `Apple` to send you a dictionary of several pieces of information as a method parameter. This can be hard to work with sometimes because you need to know the names of the keys in the dictionary in order to be able to pick out the values, but you'll get the hang of it over time.

`This dictionary parameter will contain one of two keys:` `UIImagePickerControllerEditedImage` (the image that was edited) or `UIImagePickerControllerOriginalImage`, and realistically it should only ever be the former unless you change the `allowsEditing` property.

The problem is, we don't know if these values exist as `UIImages`, so we can't just extract them straight into `UIImages`. Instead, we need to use an optional method of typecasting, `as?`, along with `if/let` syntax. Using this method, we can be sure we always get the right thing out.

Second, we need to generate a unique filename for every image we import. This is so that we can copy it to our app's space on the disk without overwriting anything, and if the user ever deletes the picture from their photo library we still have our copy. We're going to use a new class for this, called `NSUUID`, which generates a `Universally Unique Identifier` and is perfect for a random filename.

Third, once we have the image, we need to write it to disk. You're going to need to learn two new pieces of code: `UIImageJPEGRepresentation()` converts a `UIImage` to an `NSData`, and there's a method on `NSData` called `writeToFile()` that, well, writes its data to disk.

Writing information to disk is easy enough, but finding where to put it is tricky. All apps that are installed have a directory called `Documents` where you can save

private information for the app, and it's also automatically synchronised with `iCloud`. The problem is, it's not obvious how to find that directory, so I have a method I use called `getDocumentsDirectory()` that does exactly that – you don't need to understand how it works, but you do need to copy it into your code.

With all that in mind, here are the new methods:

```
func imagePickerController(picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    var newImage: UIImage

    if let possibleImage =
info[UIImagePickerControllerEditedImage] as? UIImage {
        newImage = possibleImage
    } else if let possibleImage =
info[UIImagePickerControllerOriginalImage] as? UIImage {
        newImage = possibleImage
    } else {
        return
    }

    let imageName = NSUUID().UUIDString
    let imagePath =
getDocumentsDirectory().stringByAppendingPathComponent(imageN
ame)

    if let jpegData = UIImageJPEGRepresentation(newImage, 80)
{
        jpegData.writeToFile(imagePath, atomically: true)
    }
```

```
    dismissViewControllerAnimated(true, completion: nil)
}

func getDocumentsDirectory() -> NSString {
    let paths =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .User
DomainMask, true)
    let documentsDirectory = paths[0]
    return documentsDirectory
}
```

Again, it doesn't matter how `getDocumentsDirectory()` works, but if you're curious: its first parameter asks for the documents directory, and its second parameter adds that we want the path to be relative to the user's home directory. This returns an array that nearly always contains only one thing: the user's documents directory. So, we pull out the first element and return it.

Now onto the code that matters: as you can see I've declared a `newImage` variable up front, then it gets assigned to whichever value in the dictionary has an image. If we can't find an image for some reason, we exit the method. We then create an `NSUUID` object, and use its `UUIDString` property to extract the unique identifer as a string data type.

The code then creates a new constant, `imagePath`, which takes the string result of `getDocumentsDirectory()` and calls a new method on it: `stringByAppendingPathComponent()`. This is used when working with file paths, and adds one string (`imageName` in our case) to a path, including whatever path separator is used on the platform.

Now that we have a `UIImage` containing an image and a path where we want to save it, we need to convert the `UIImage` to an `NSData` object so it can be saved. To do that, we use the `UIImageJPEGRepresentation()` function, which takes two parameters: the `UIImage` to convert to `JPEG` and a quality value between `0` and `100`.

Once we have an `NSData` object containing our `JPEG` data, we just need to unwrap it safely then write it to the file name we made earlier. That's done using the `writeToFile()` method, which takes a filename as its first parameter and a boolean as its second. That second parameter, `"atomically"`, should generally be `true`. It means `"write to a temporary file first, then rename it to be the file you asked"` which has the benefit that the file doesn't appear to exist until it has been fully written.

So: users can pick an image, and we'll save it to disk. But this still doesn't do anything – you won't see the picture in the app, because we aren't doing anything with it beyond writing it to disk. To fix that, we need to create a custom class to hold custom data…

Custom subclasses of NSObject

You already created your first custom class when you created the collection view cell. But this time we're going to do something very simple: we're going to create a class to hold some data for our app. So far you've seen how we can create arrays of strings by using `[String]`, but what if we want to hold an array of people?

Well, the solution is to create a custom class. Create a new file (you remember my explicit, `"read this carefully"` instructions, right?) and choose `Cocoa`

Touch Class. Click Next and name the class Person, type NSObject for "Subclass of", then click Next and Create to create the file.

NSObject is what's called a universal base class for all Cocoa Touch classes. That means all UIKit classes ultimately come from NSObject, as do all NS objects like NSString. You don't have to inherit from NSObject in Swift, but you did in Objective C and in fact there are some behaviors you can only have if you do inherit from it. More on that in Project 12, but for now just make sure you inherit from NSObject.

We're going to add two properties to our class: a name and a photo for every person. So, add this inside the Person definition:

```
var name: String
var image: String
```

When you do that, you'll see errors: "Class 'Person' has no initializers". This is a term I've skipped over so far, but now is a good time to introduce it: an initializer method is something that creates instances of a class. You've been using these all along: the contentsOfFile method for NSString is an initializer, as is UIAlertController(title:message:preferredStyle:).

Swift is telling you that you aren't satisfying one of its core rules: objects of type String can't be empty. Remember, String! and String? can both be nil, but plain old String can't – it must have a value. Without an initializer, it means the object will be created and these two variables won't have values, so you're breaking the rules.

To fix this problem, we need to create an `init()` method that accepts two parameters, one for the name and one for the image. We'll then save that to the object so that both variables have a value, and `Swift` is happy.

Doing this gives you the chance to learn something else: another required usage of the self keyword. Here's the code:

```
init(name: String, image: String) {
    self.name = name
    self.image = image
}
```

As you can see, the method takes two parameters: `name` and `image`. These are perfectly valid parameter names, but also happen to be the same names used by our class. So if we were to write something like this…

```
name = name
```

…then it would be confusing. Are you assigning the parameter to itself? Are we assigning the class's property to the parameter? To solve this problem, you use `self.` to clarify which is which, so `self.image = image` can only mean one thing: assign the parameter to the class's property.

Our custom class is done; it's just a dumb data store for now. If you're the curious type, you might wonder why I used a class here rather than a struct. This question is even more pressing once you know that structs have an automatic initializer method made for them that looks exactly like ours. Well, the answer is: you'll have to wait and see. All will become clear in `Project 12`!

With that custom class done, we can start to make our project much more useful: every time a picture is imported, we can create a `Person` object for it and add it to an array to be shown in the collection view.

So, go back to `ViewController.swift`, and add this declaration for a new array:

```
var people = [Person]()
```

Every time we add a new person, we need to create a new `Person` object with their details. This is as easy as modifying our initial image picker success method so that it creates a `Person` object, adds it to our people array, then reloads the collection view. Put this code before the call to `dismissViewControllerAnimated()`:

```
let person = Person(name: "Unknown", image: imageName)
people.append(person)
collectionView.reloadData()
```

That stores the image name in the `Person` object and gives them a default name of "Unknown", before reloading the collection view.

Can you spot the problem? If not, that's OK, but you should be able to spot it if you run the program.

The problem is that although we've added the new person to our array and reloaded the collection view, we aren't actually using the people array with the collection view – we just return `10` for the number of items and create an empty collection view cell for each one! Let's fix that…

Connecting up the people

We need to make three final changes to this project in order to finish: show the correct number of items, show the correct information inside each cell, then make it so that when users tap a picture they can set a person's name.

Those methods are all increasingly difficult, so we'll start with the first one. Right now, your collection view's `numberOfItemsInSection` method just has return `10` in there, so you'll see `10` items regardless of how many people are in your array. This is easily fixed:

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

Next, we need to update the collection view's `cellForItemAtIndexPath` method so that it configures each `PersonCell` cell to have the correct name and image of the person in that position in the array. This takes a few steps:

- Pull out the person from the people array at the correct position.
- Set the name label to the person's name.
- Create a `UIImage` from the person's image filename, adding it to the value from `getDocumentsDirectory()` so that we have a full path for the image.

We're also going to use this opportunity to give the image views a border and slightly rounded corners, then give the whole cell matching rounded corners, to make it all look a bit more interesting. This is all done using `CALayer`, so that

means we need to convert the `UIColor` to a `CGColor`. Anyway, here's the new code:

```
func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {
    let cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Person
", forIndexPath: indexPath) as! PersonCell

    let person = people[indexPath.item]

    cell.name.text = person.name

    let path =
getDocumentsDirectory().stringByAppendingPathComponent(person
.image)
    cell.imageView.image = UIImage(contentsOfFile: path)

    cell.imageView.layer.borderColor = UIColor(red: 0, green:
0, blue: 0, alpha: 0.3).CGColor
    cell.imageView.layer.borderWidth = 2
    cell.imageView.layer.cornerRadius = 3
    cell.layer.cornerRadius = 7

    return cell
}
```

The only new thing in there is setting the `cornerRadius` property, which rounds the corners of a `CALayer` – or in our case the `UIView` being drawn by the `CALayer`.

With that done, the app works: you can run it with Cmd + R, import photos, and admire the way they all appear correctly in the app. But don't get your hopes up, because we're not done yet – you still can't assign names to people!

For this last part of the project, we're going to recap how to add text fields to a UIAlertController, just like you did in Project 5. All of the code is old, but I'm going to go over it again to make sure you fully understand.

First, the delegate method we're going to implement is the UICollectionView's didSelectItemAtIndexPath method, which is triggered when the user taps a cell. This method needs to pull out the Person object at the array index that was tapped, then show a UIAlertController asking users to rename the person.

Adding a text field to an alert controller is done with the addTextFieldWithConfigurationHandler() method. We'll also need to add two actions: one to cancel the alert (with a nil handler), and one to save the change. To save the changes, we need to add a closure that pulls out the text field value and assigns it to the person's name property, then we'll also need to reload the collection view to reflect the change.

That's it! The only thing that's new, and it's hardly new at all, is the setting of the name property. Put this new method into your class:

```
func collectionView(collectionView: UICollectionView,
didSelectItemAtIndexPath indexPath: NSIndexPath) {
    let person = people[indexPath.item]

    let ac = UIAlertController(title: "Rename person",
message: nil, preferredStyle: .Alert)
```

```
    ac.addTextFieldWithConfigurationHandler(nil)

    ac.addAction(UIAlertAction(title: "Cancel",
style: .Cancel, handler: nil))

    ac.addAction(UIAlertAction(title: "OK", style: .Default)
{ [unowned self, ac] _ in
        let newName = ac.textFields![0]
        person.name = newName.text!

        self.collectionView.reloadData()
    })

    presentViewController(ac, animated: true, completion:
nil)
}
```

Finally, the project is complete: you can import photos of people, then tap on them to rename. Well done!

Wrap up

`UICollectionView` and `UITableView` are the most common ways of showing lots of information in `iOS`, and you now know how to use both. You should be able to go back to `Project 1` and recognise a lot of very similar code, and that's by intention – `Apple` has made it easy to learn both view types by learning either one.

You've also learned another batch of iOS development, this time UIImagePickerController, NSUUID, custom classes and more. You might not realise it yet, but you have enough knowledge now to make a huge range of apps!

If you wanted to take this app further you could add a second UIAlertController that is shown when the user taps a picture, and asks them whether they want to rename the person or delete them.

You could also try picker.sourceType = .Camera when creating your image picker, which will tell it to create a new image by taking a photo. This is only available on devices (not on the simulator!) so you might want to check the return value of UIImagePickerController.isSourceTypeAvailable() before trying to use it!

Before we finish, you may have spotted one problem with this app: if you quit the app and relaunch, it hasn't remembered the people you added. Worse, the JPEGs are still stored on the disk, so your app takes up more and more room without having anything to show for it!