CHAPTER 20

ADAPTIVE COLLECTION VIEW

ADAPTIVE COLLECTION VIEW

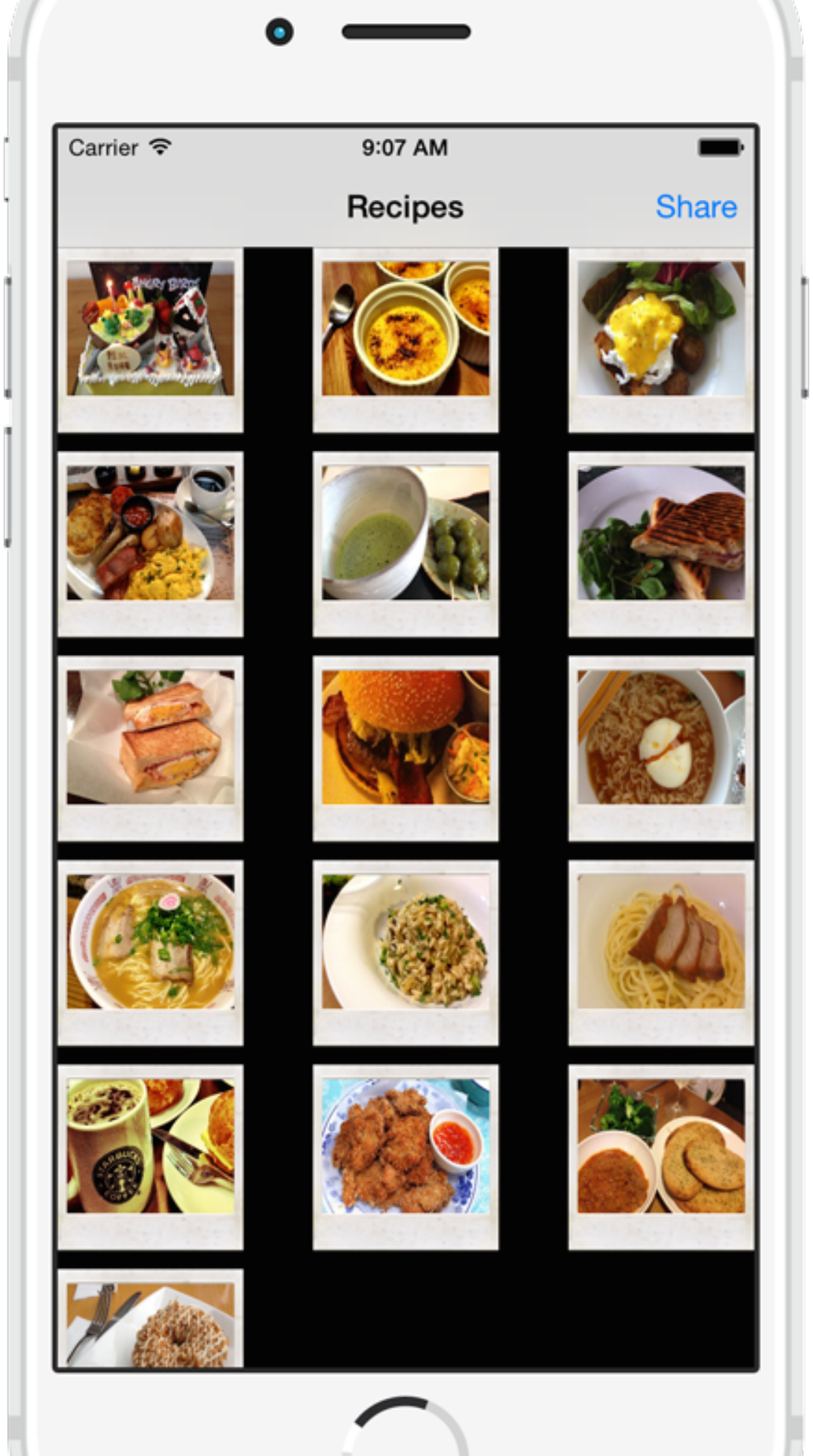
In the previous two chapters, you learned to build a demo app using a collection view. The app works perfectly on iPhone 5/5s. But if you run the app on iPhone 6/6 Plus, your screen should look like the screenshot shown on your right. The recipes are displayed in grid format but with a large space between items.

The screen of iPhone 6 and 6 Plus is wider than that of their predecessors. As the size of the collection view cell was fixed, the app rendered extra space between cells according to the screen width of the test device.

So how can we fix the issue? As mentioned in the first chapter of the book, iOS 8 comes with a new concept called Adaptive User Interfaces. You will need to make use of Size Classes and UITraitCollection to adapt the collection view to a particular device and device orientation. If you haven't read Chapter 1, I would recommend you to pause here and go back to the first chapter. Everything I will cover here will be based on the material covered in the very beginning of the book.

As usual, we will build a demo app to walk you through the concept. You are going to create an app similar to the one before but with the following changes:

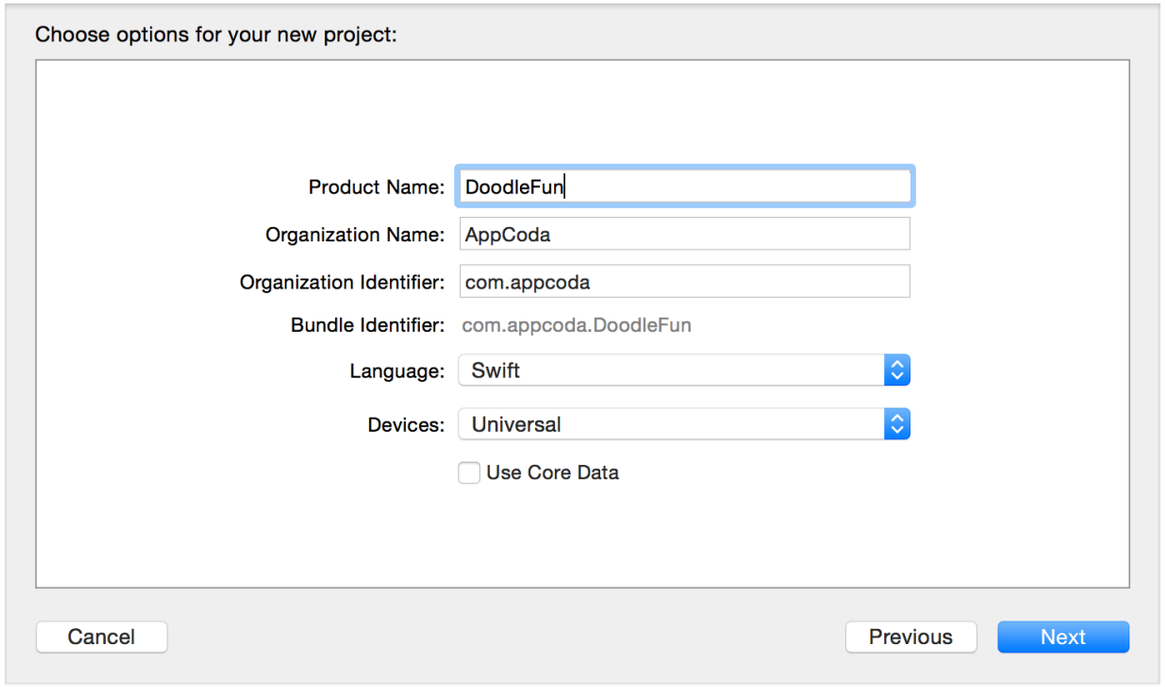
- The cell is adaptive - The size of the collection view cell changes according to a particular device and orientation. You will learn how to use size classes and UITraitCollection to make the collection view adaptive.
- The app is universal - It is a universal app that supports both iPhone and iPad.
- We will use UICollectionView - Instead of using UICollectionViewController, you will learn how to use UICollectionView to build a similar UI.



CREATING THE DEMO PROJECT

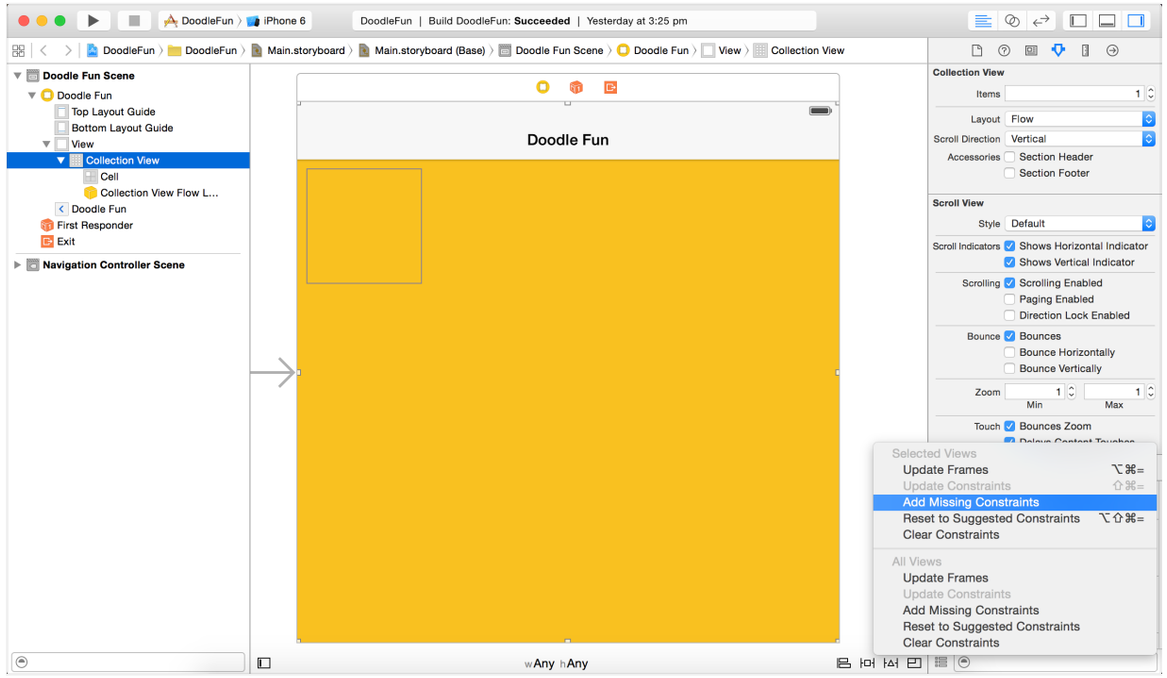
To get started, download the project template called DoodleFun from https://www.dropbox.com/s/jbntfxvg4vthwm7/DoodleFunTemplate.zip?dl=0. I have included a set of Doodle images (provided by the team at RoundIcons) and prebuilt the storyboard for you.

If you prefer to create the project from scratch, make sure to select Universal for the Devices option as we're going to build a universal app.



Assuming you've downloaded the template, open the project in Xcode and go to Main.storyboard. Drag a Collection View object from the Object Library to the View Controller.

In the Attributes inspector, change the background color to yellow. Next, go to the Size inspector. Set the cell size to 128 by 128. Change the values of Section Insets (Top, Bottom, Left and Right) to 10 points. The insets define the margins applied to the content of the section.

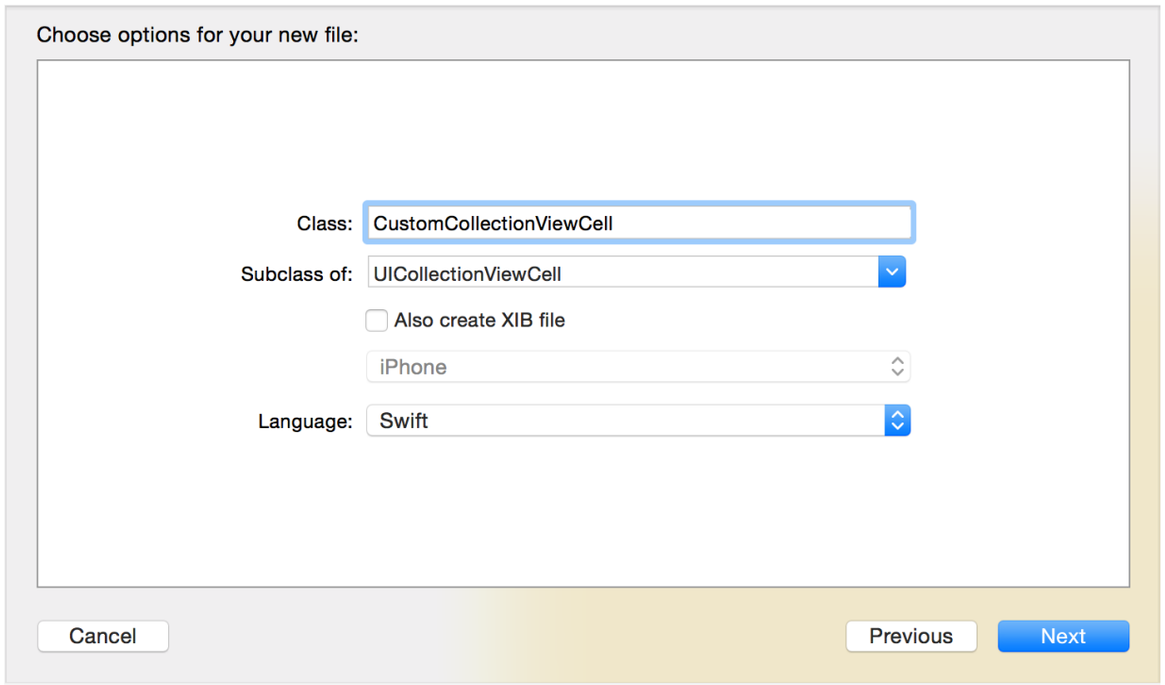


Because we are using a Collection View instead of Collection View Controller, we have to deal with auto layout constraints on our own. The simplest way to do this is to select the collection view and then click the Issues button of the auto layout menu, followed by selecting the "Add Missing Constraints" option. Xcode automatically defines the constraints for you.

Next, select the collection view cell and set its identifier to "Cell" under the Attributes inspector. Drag an image view to the cell. Again, click the Issues button of the auto layout menu and select the "Add Missing Constraints" option to define the layout constraints.

DIVING INTO THE CODE

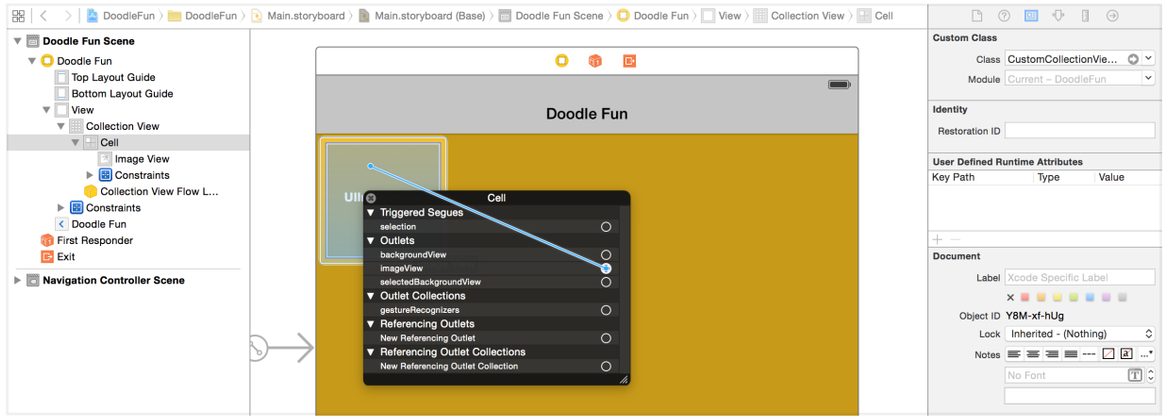
Now that you've created the collection view in the storyboard, let's move on to the coding part. First create a new file named CustomCollectionViewCell and set it as a subclass of UICollectionViewCell.



Once the file was created, declare an outlet variable for the image view:

```
class CustomCollectionViewCell: UICollectionViewCell {
    @IBOutlet weak var imageView:UIImageView!
}
```

Switch to the storyboard. Select the collection view cell and change its custom class (under Identity inspector) to CustomCollectionViewCell. Then right click the cell and connect the imageView outlet variable with the image view.



The ViewController.swift file is associated with the main view controller in the storyboard. As we want to present a set of images using the collection view, we have to implement both the UICollectionViewDataSource and UICollectionViewDelegate protocols.

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

Next, declare an array for the images and an outlet variable for the collection view:

```
var doodleImages = ["DoodleIcons-1", "DoodleIcons-2",
"DoodleIcons-3", "DoodleIcons-4", "DoodleIcons-5", "DoodleIcons-6",
"DoodleIcons-7", "DoodleIcons-8", "DoodleIcons-9", "DoodleIcons-10",
"DoodleIcons-11", "DoodleIcons-12", "DoodleIcons-13",
"DoodleIcons-14", "DoodleIcons-15", "DoodleIcons-16",
"DoodleIcons-17", "DoodleIcons-18", "DoodleIcons-19",
"DoodleIcons-20"]

@IBOutlet weak var collectionView:UICollectionView!
```

Like what we did before, we will have to implement two required methods of the UICollectionViewDataSource protocol:

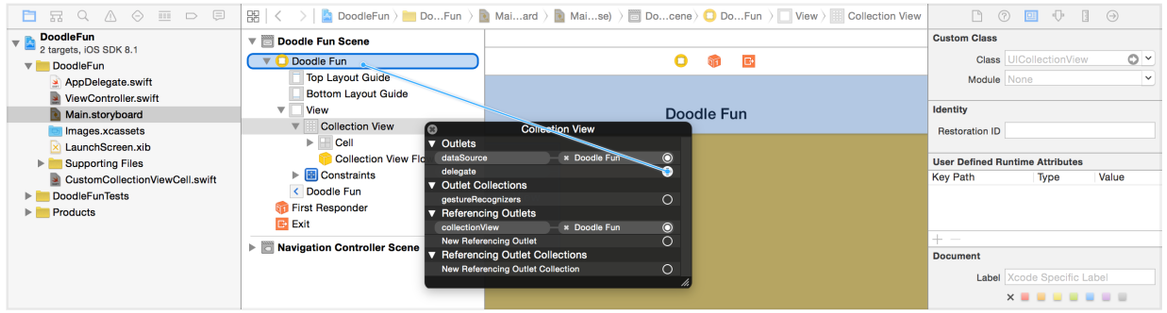
- func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int
- func collectionView(_ collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell

Okay, let's implement the methods like this:

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

func collectionView(collectionView: UICollectionView,
cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell
{
    var cell =
collectionView.dequeueReusableCellWithReuseIdentifier("Cell",
forIndexPath: indexPath) as! CustomCollectionViewCell

    cell.imageView.image = UIImage(named:
doodleImages[indexPath.row])

    return cell

}
```

The above code is very straightforward. We return the total number of images in the first method and set the image of the image view in the latter method.

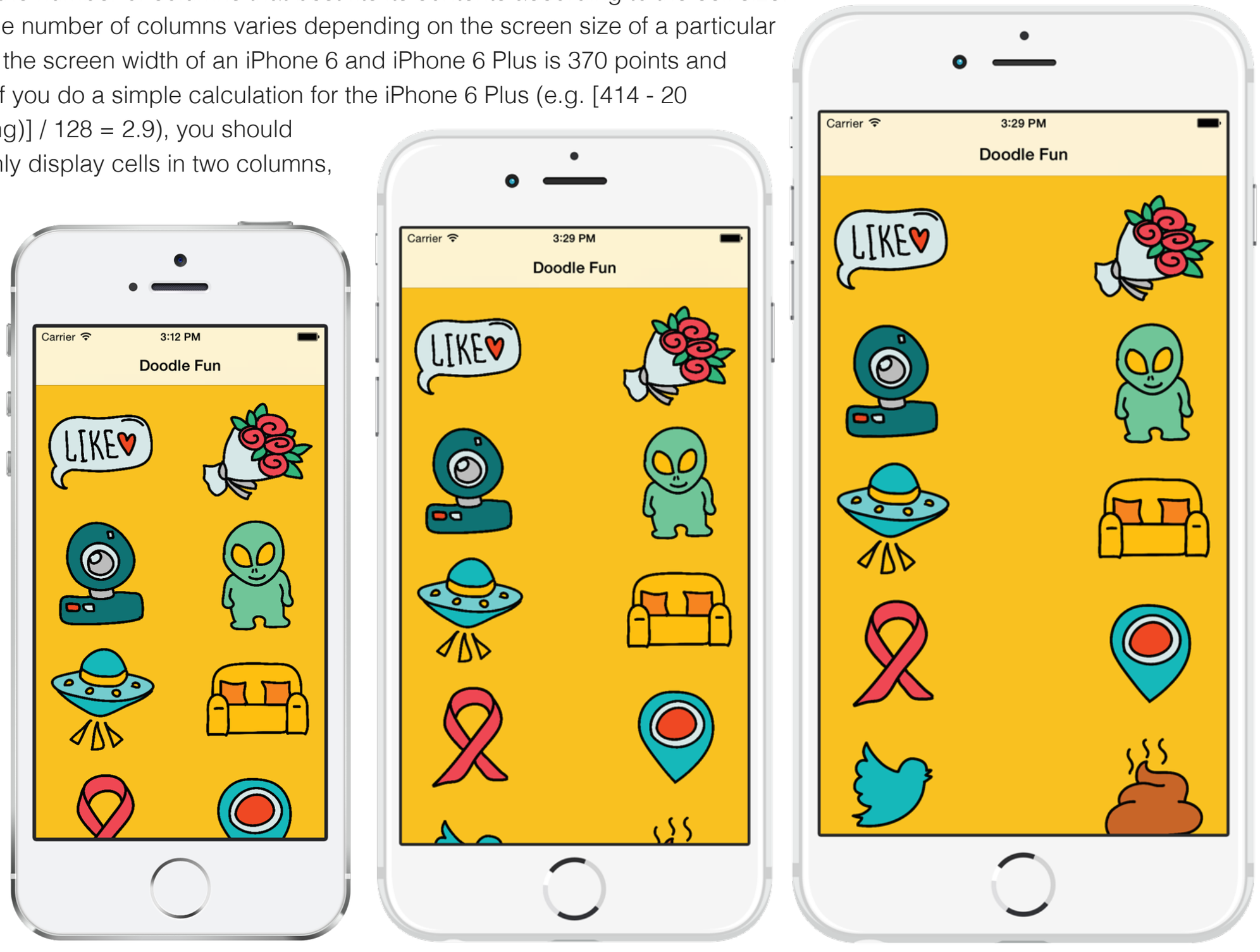
Now switch over to the storyboard. Establish a connection between the collection view and the collectionView outlet variable. Also, connect the dataSource and delegate with the view controller.



That's it! We're ready to test the app.

Compile and run the app on iPhone 5/5s simulator. The app looks pretty good, right? Now try to test the app on other iOS devices including the iPad and in landscape orientation. The app looks great on most devices but falls short on iPhone 6 and 6 Plus. UICollectionView can automatically determine the number of columns that best fits its contents according to the cell size.

As you can see below, the number of columns varies depending on the screen size of a particular device. In portrait mode, the screen width of an iPhone 6 and iPhone 6 Plus is 370 points and 414 points respectively. If you do a simple calculation for the iPhone 6 Plus (e.g. [414 - 20 (margin) - 20 (cell spacing)] / 128 = 2.9), you should understand why it can only display cells in two columns, leaving a large gap between columns.



DESIGNING FOR SIZE CLASSES

So how can you fix this issue on the iPhone 6 and 6 Plus? Obviously you can reduce the cell size so that it fits well on all Apple devices. A better way to resolve the issue, however, is to make the cell size adaptive.

The collection view works pretty well in landscape orientation regardless of device types. To fix the display issue, we are going to keep the size of the cell the same (i.e. 128x128 points) for devices in landscape mode but minimize the cell for iPhones in portrait mode.

The real question is how do you find out the current device and its orientation?

In the past, you would determine the device type and orientation using code like this:

```
let device = UIDevice.currentDevice()
let orientation = device.orientation
let isPhone = (device.userInterfaceIdiom == UIUserInterfaceIdiom.Phone)
? true : false

if isPhone {
    if orientation.isPortrait {
        // Change cell size
    }
}
```

In iOS 8, the above code is not ideal. You're discouraged from using UIUserInterfaceIdiom to verify the device type. Instead, you should use size classes to handle issues related to idiom and orientation.

I covered size classes in Chapter 1, so I won't go into the details here. In short, it boils down to this two by two grid:

Vertical Size Class	Horizontal Size Class: Regular	Horizontal Size Class: Compact
Regular	iPad Portrait iPad Landscape	iPhone Portrait
Compact	iPhone 6 Plus Landscape	iPhone 4/5/6 Landscape

There is no concept of orientation. For iPhones in portrait mode, it is indicated by a compact horizontal class and regular vertical class.

So how can you access the current size class from code?

UNDERSTANDING TRAIT COLLECTION

Well, you use a new system called Traits. The horizontal and vertical size classes are considered traits. Together with other properties like userInterfaceIdiom and display scale they make up a so-called trait collection.

In iOS 8, Apple introduced trait environments (i.e.UITraitEnvironment). This is a new protocol that is able to return the current trait collection. Because UIViewController conforms to the UITraitEnvironment protocol, you can access the current trait collection using the *traitCollection* property.

If you put the following line of code in the *viewDidLoad* method to print its content to console:

```
println("\(traitCollection)")
```

You should have something like this when running the app on an iPhone 6 Plus:

```
<UITraitCollection: 0x7f857b5b6860; _UITraitNameUserInterfaceIdiom = Phone, _UITraitNameDisplayScale = 3.000000,
_UITraitNameHorizontalSizeClass = Compact, _UITraitNameVerticalSizeClass = Regular, _UITraitNameTouchLevel = 0, _UITraitNameInteractionModel =
1>
```

From the above information, you discover the device is an iPhone that is in the Compact horizontal and Regular vertical size classes. The display scale of 3x indicates a Retina HD 5.5 display.

ADAPTIVE COLLECTION VIEW

With a basic understanding of trait collection, you should know how to determine the current size class of a device. Now it's time to make the collection view adaptive. The UICollectionViewDelegateFlowLayout protocol provides an optional method for specifying the size of a cell:

- collectionView(_:layout:sizeForItemAtIndexPath:)

All you need to do is override the method and return the cell size during runtime. Recall that we only want to alter the cell size for iPhones in portrait mode, so we will implement the method like this:

```
func collectionView(collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAtIndexPath indexPath:
NSIndexPath) -> CGSize {
    let sideSize = (traitCollection.horizontalSizeClass == .Compact && traitCollection.verticalSizeClass == .Regular) ? 80.0 : 128.0
    return CGSize(width: sideSize, height: sideSize)
}
```

For devices with a Compact horizontal and a Regular vertical size class (i.e. iPhone Portrait), we set the size of the cell to 80x80 points. Otherwise, we just keep the cell size the same.

Run the app again on an iPhone 6 / 6 Plus. It should look much better now.


Carrier
12:50 AM
Doodle Fun
Carrier
12:53 AM
Doodle Fun
Carrier
12:54 AM
Doodle Fun
Carrier
3:30 PM
100%
Doodle Fun

RESPOND TO THE CHANGE OF SIZE CLASS

Did you try to test the app in landscape mode? When you turned the iPhone sideways, the cell size was unchanged. There is one thing missing in the current implementation; we have not implemented the method that responds to size and trait changes. Insert the following method in the View Controller class:

```
override func viewWillTransitionToSize(size: CGSize,
    withTransitionCoordinator coordinator: UIViewControllerTransitionCoordinator) {
        collectionView.reloadData()
}
```

When the size of the view is about to change (e.g. rotation), UIKit will call the method. Here we simply update the collection view by reloading its data. Now test the app again. When your iPhone is put in landscape mode, the cell size should be changed accordingly.

For your reference, you can download the complete project from https://www.dropbox.com/s/wgt056afop0raq4/DoodleFunFinal.zip?dl=0.

YOUR EXERCISE

In some scenarios, you may want all images to be visible in the collection view without scrolling. In this case, you'll need to perform some calculations to adjust the cell size based on the area of the collection view. To calculate the total area of the collection view, you can use the code like this:

```
let collectionViewSize = collectionView.frame.size
let collectionViewArea = Double(collectionViewSize.width * collectionViewSize.height)
```

With the total area and total number of images, you can calculate the new size of a cell. For the rest of the implementation, I will leave it as an exercise for you. Take some time and try to figure it out on your own before checking out the solution at https://www.dropbox.com/s/562118zu12z10yd/DoodleFunExercise.zip?dl=0.