MOBILE PROGRAMMING SERIES

iOS UICollectionView

The Complete Guide

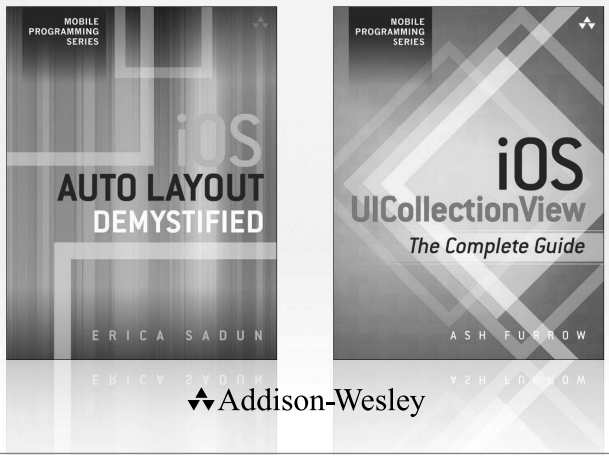
ASH FURROW

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

iOS UICollectionView: The Complete Guide

Addison-Wesley Mobile Programming Series



Visit **informit.com/mobile** for a complete list of available publications.

The **Addison-Wesley Mobile Programming Series** is a collection of digital-only programming guides that explore key mobile programming features and topics in-depth. The sample code in each title is downloadable and can be used in your own projects. Each topic is covered in as much detail as possible with plenty of visual examples, tips, and step-by-step instructions. When you complete one of these titles, you'll have all the information and code you will need to build that feature into your own mobile application.

iOS UICollectionView: The Complete Guide

Ash Furrow

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

iOS UICollectionView: The Complete Guide

ISBN-13: 978-0-13-341094-5
ISBN-10: 0-13-341094-3

Acquisitions Editors
Trina MacDonald
Angie Doyle

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Senior Project Editor
Jovana Shirley

Copy Editor
The Wordsmithery LLC

Proofreader
Sarah Kearns

Technical Editor
Richard Wardwell

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Contents at a Glance

Acknowledgments

I want to thank Angie Doyle and Trina MacDonald at Pearson Education for contacting me about writing this book. I was planning on writing an ebook about something, but with their guidance and resources, I know this book is way more awesome than anything I could have done on my own.

Rich Wardwell has been a wonderful technical editor, offering comprehensive advice concerning clarity of both my code and my prose.

I am a strong believer in the open-source community, and this book relies on some open-source software. Some of it I wrote myself, but some I didn't. I'd like to thank Mark Pospesel for his contributions to GitHub (https://github.com/mpospese/IntroducingCollectionViews) in "Introducing UICollectionViews." Mark specializes in mathematics, and while writing this book, it's been great to be able to rely on his expertise.

Speaking of the open-source community, no book discussing `UICollectionView` would be complete without a tip of the hat to Peter Steinberger's work on PSTCollectionView (https://github.com/steipete/PSTCollectionView), a 100-percent API-compatible replacement for `UICollectionView` that offers backward compatibility with iOS 4.3+. Most of the techniques discussed in this book are directly applicable to PSTCollectionView, and the project is advancing every day. If you need to support older versions of iOS, use PSTCollectionView.

Finally, I could not have completed this book without the support of my wife. Her constant prodding about deadlines made sure I was only a little late most of the time. I am lucky to have such a supportive partner who understands and encourages my compulsion to create and share.

About the Author

Ash Furrow has been developing iOS applications since 2009. He's made several of his own applications available in the App Store, and he headed the iOS team at 500px to ship its critically acclaimed app. Now he creates amazing products with Teehan+Lax.

When he's not busy writing books or blog posts (http://ashfurrow.com/), Ash enjoys photography and roasting his own coffee.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: trina.macdonald@pearson.com

Mail: Reader Feedback
Addison-Wesley's Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Preface

At WWDC 2012, Apple unveiled `UICollectionView`, enabling a new way for apps to render content to users. Collection views are a content- and layout-agnostic tool for developers to display content in apps. User interfaces created with collection views are some of the most immersive, distinctive interfaces in iOS applications.

However, the power afforded to developers by collection views is balanced by the complexity of using them. As the saying goes, Cocoa makes common things easy and uncommon things possible. `UICollectionView` embodies this sentiment.

I said earlier that collection views are layout-agnostic, and that's true: Developers write their own layouts for collection views to use to organize their content on the screen. Luckily, Apple included a sample layout that displays grids, a common request among developers.

How to Use This Book

This book is meant to tell a story; each chapter builds upon the last one to guide readers through every nook and cranny of `UICollectionView`. I strongly encourage readers to read each chapter in sequence and follow along with the code samples.

The first chapter makes sure that readers have a common vocabulary when discussing the organization of code in iOS applications. Even if you're a seasoned developer, it's worth a look just to make sure you're on the same page as I am.

The code provided with this book is as valuable as the explanations in the chapters of *why* the code is written the way it is.

All of the code that appears in this book can be downloaded at http://ashfurrow.com/uicollectionview-the-complete-guide/.

Who This Book Is For

This book is for intermediate to advanced iOS developers who want to take full advantage of `UICollectionView`. If you're trying to write your first-ever iOS application, this book probably isn't for you. I've written this book with the assumption that you understand the concepts of objects and view hierarchies, as well as basic Objective-C syntax.

Organization of This Book

This book is organized into six chapters to guide readers through a comprehensive description of every aspect of collection views:

- **Chapter 1, "Understanding Model-View-Controller on iOS,"** briefly introduces the MVC paradigm of application architecture that's used throughout the remainder of the book.
- **Chapter 2, "Displaying Content Using `UICollectionView`,"** introduces readers to `UICollectionView` with some basic examples using .xib files and storyboards, as well as view setup using only code. This chapter ends with a case study on application performance tuning.
- **Chapter 3, "Contextualizing Content,"** builds on the basics of cell use from Chapter 2 to explain how to contextualize content for users by using supplementary views. The chapter explores the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols as well.
- **Chapter 4, "Organizing Content with `UICollectionViewFlowLayout`,"** introduces readers to the idea of creating their own custom layouts while relying on existing logic in `UICollectionViewFlowLayout`. The sample code from Chapter 3 is augmented with decoration views, and custom collection view attributes are used to customize cell layout. The chapter ends with a look at a Cover Flow-esque layout.
- **Chapter 5, "Crafting Custom Layouts Using `UICollectionViewLayout`,"** explains to readers who understand subclassing flow layouts that they can subclass `UICollectionViewLayout` directly for incredibly custom layouts. The chapter also covers changing layouts with animation support, as well as provides some further examples on how to use supplementary views and decoration views with completely custom layouts.
- **Chapter 6, "Adding Interactivity to `UICollectionView`,"** is the crown jewel of this book. It looks back at all the previous chapters' code samples to augment them with interactivity, mostly using gesture recognizers.

Special Thanks

I want to thank Mark Pospesel for his work in the open-source community, specifically his contributions to “Introducing UICollectionViews” available on GitHub: https://github.com/mpospese/IntroducingCollectionViews. A lot of the math in the later chapters is taken from Mark’s code. This book would not be as awesome if it weren’t for Mark's open source contributions.

This page intentionally left blank

1

Understanding Model-View-Controller on iOS

Before you dive into `UICollectionView`, you should get familiar with some of the conventions and terms used in this book. The book starts with the basics of the iOS application lifecycle and then discusses the Model-View-Controller (MVC) paradigm. Even if you're an experienced iOS developer already familiar with these topics, I encourage you to read this chapter to make sure that you're on the same page (or screen, so to speak) that I am while you're reading the rest of this book.

Basics of the Application Lifecycle

The iOS application lifecycle is a little different from typical native applications on other platforms (although recent changes to OS X show Apple is interested in making the iOS lifecycle the norm). Developers no longer have hard-and-fast rules for when their applications are terminated, suspended, and so on. Let's start with a simple scenario to describe a typical application lifecycle.

The user has just turned on his phone and no applications are running except for those that belong to the operating system. Your application is *not* running. After the user taps your app's icon, Springboard—the part of the OS that operates the Home screen of iOS—launches your app. Your app, and the shared libraries it needs to execute, are loaded into memory while Springboard animates your `Default.png` on the screen. Eventually, your app begins execution, and your application delegate receives the appropriate notification. When your application is running and in the foreground, it is in the **active** state.

On iOS, users tend to only use any given application for a few seconds before returning their phones to their pockets. After the user has put away your app by pressing the Home

button on her iPhone or iPad, your application enters the **background** state. Typically, apps have 10 seconds to complete any database saves or other long-running tasks (though applications can request additional time from the OS). When all the background processing is complete, the application finally becomes **suspended**. While suspended, applications remain in memory but may not execute code. The state of your application is persisted. If the user opens your application while it is suspended, it begins execution exactly where it left off. If memory becomes low, the OS can kill your app while it is in the suspended state. The user can also manually terminate your app from the multitasking tray. Once terminated, applications return to their initial state of not running.

But wait, it gets more complicated! If the user receives a calendar alert, opens the multitasking tray, or gets a phone call, your application can be put into the **inactive** state. Your application is still running, but it is no longer the foremost thing the user interacts with. For example, games pause themselves. As an application developer, you need to be aware of this and use it as an indication that the user might leave your application soon.

The user can open your application without tapping its icon on the Home screen. If your application receives local or push notifications, or if it is registered for custom URL scheme handling, the user can open it in any number of ways.

The application lifecycle is important to understand for all iOS developers who want to make enriched, immersive experiences. These types of applications are exactly what `UICollectionView` is great for, so no comprehensive discussion of `UICollectionView` would be complete without a summary of the application lifecycle.

If your app enters the inactive state, stop updating your interface. It would be disconcerting for a user to see your collection-view contents move about while he's deciding whether to view the details of an appointment that has popped up over your application. Likewise, don't update your app's interface while the application is in the background. The state of the user interface should remain fixed between the switch from active to background and back to active.

How to Use MVC

MVC is not a difficult concept, but there are two main reasons for emphasizing its importance in iOS:

- MVC is used by CocoaTouch (and Cocoa on OS X). If you adhere to the same paradigm as the frameworks used for writing all iOS applications, your code will flow well and not clash with the built-in classes, including `UICollectionView`.
- MVC is generally a good framework, and using it will help you make well-written, maintainable apps.

Now that you know why MVC is important, it's time to look at what MVC is. Figure 1.1 shows the basics of MVC; strong relationships are represented with solid lines, and weak relationships are represented by dashed ones. Strong and weak relationships indicate to the

compiler how to manage memory and are important to avoid memory leaks, which would eventually lead to the app being terminated.

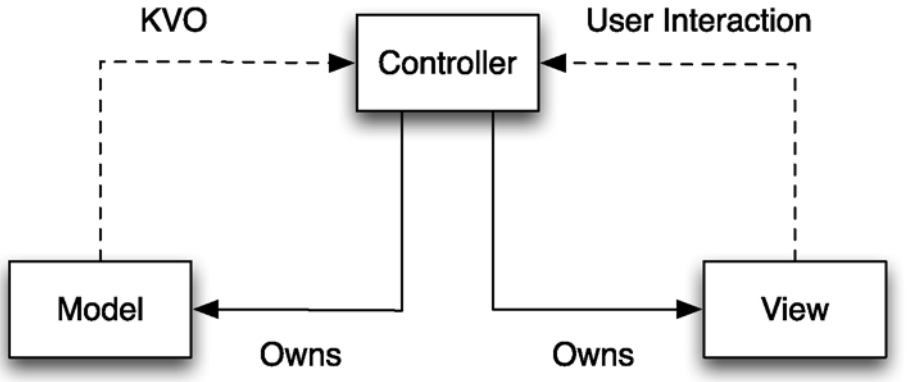


Figure 1.1 Basics of MVC

At the heart of MVC is the controller object. The controller is a View Controller—as in `UIViewController`—and it controls the view. It maintains a strong relationship to this view, which is what is presented to the user on the screen. The controller also maintains a strong relationship to the model. The model represents data that is represented in the view.

If your view ever has a reference to your model, or vice versa, you're doing it wrong. This book uses MVC and you should, too.

Most of the code in any given application resides in the controller; controllers mediate the interactions between views and models, which is why the code in controllers is often referred to as "glue code."

What sort of interactions does a controller mediate? Well, if the view contains a button, then the view controller is notified when the user taps that button. Typically, user interactions trigger actions to modify, create, or delete models belonging to the controller. The controller receives the user interaction from the view, updates the model, and then updates the view to reflect the changes made to the model.

Sometimes, the model changes without user interaction. For example, consider a view that displays a large JPEG, which is being downloaded. When the download completes, the controller should be notified so it can update the view. On iOS, you have a few different choices for how to notify the controller. My favorite is Key-Value Observation (KVO). Controllers can register themselves as observers on model objects so they are notified whenever the model's properties are changed. Other ways for models to interact with controllers on iOS include `NSNotificationCenter`, `delegation`, and `NSFetchedResultsController`. I would eschew `NSNotificationCenter` for model-controller interaction in favor of `NSFetchedResultsController` or KVO. Although this book doesn't discuss Core Data, `UICollectionView` works very well with `NSFetchedResultsController` in a similar way to `UITableViewController`.

This last example demonstrates a gaping hole in MVC: Where does the network code go? As a responsible iOS developer, you should keep the view controller to only mediating the interactions between the view and the model. If that's the case, then it shouldn't be used to house the network access code. As discussed in Chapter 6, "Adding Interactivity to `UICollectionView`," the network code should be placed *outside* of the typical MVC pyramid. Network access should not involve the view whatsoever, but it can sometimes involve the model.

Well, that's *mostly* true. In fact, a common paradigm for fetching details about a model from an application programming interface (API) involves Grand Central Dispatch blocks. A block lets developers treat anonymous functions as first-class Objective-C objects. These blocks can be invoked later. Controllers can start a network request and pass the network-fetching object a callback block that updates the view. Technically, the network code has an indirect reference to the view, but you ignore it lest you find yourself falling down a rabbit hole of pedantry.

If you are experienced in iOS development, all of this should sound familiar. `UICollectionView` and `UICollectionViewController` don't exist in silos; they are used within applications with models and with the rest of CocoaTouch. It would be irresponsible to present them in any other context than that of MVC.

MVC and `UICollectionView`

Now that you've read about the MVC paradigm, look at its application in the context of writing `UICollectionView` code.

The view component of MVC with `UICollectionView` is unsurprisingly the `UICollectionView` itself; the controller is either a subclass of `UICollectionViewController` or a subclass of `UIViewController` that conforms to the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols; the model can be anything.

Like with `UITableView`, your controller can either subclass `UIViewController` and conform to the two protocols for the collection view data source and delegate, or it can subclass `UICollectionViewController` itself. If you look in the header file of `UICollectionViewController`, you see that it's very sparse. The controller inherits from `UIViewController`—conforming to `UICollectionViewDataSource` and `UICollectionViewDelegate`—and has a convenience initializer to programmatically create an instance of it using a collection view with a specific layout. It contains a property to access the collection view and another property to specify whether the selection in a collection view becomes cleared when it (re)appears.

When using a `UICollectionViewController` subclass, the `view` property of `UIViewController` points to the same object as the `collectionView` property of `UICollectionViewController`. The view *is* the collection view. If you plan to use only `UICollectionView` to display data to your user, I strongly recommend subclassing this

prebuilt controller. In my experience, you run into fewer "gotchas" using these special controllers from Apple.

In some circumstances, subclassing `UIViewController` is preferable. For example, if your view *contains* a collection view, but also contains other views, it's easier to have the collection view as a subview of the controller's view. The distinction is minor, but important.

Figures 1.2 and 1.3 demonstrate the differences in the two approaches to using collection views. `UICollectionViewController` is far simpler; it should be the approach you take first. If you find you can't solve your problem with it, switch to using the second approach. It's usually easy to switch from using the first method to the second.

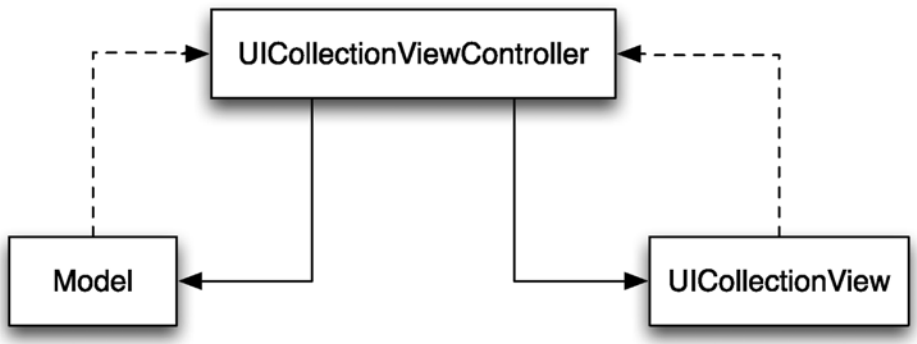


Figure 1.2 Example of MVC using `UICollectionViewController`

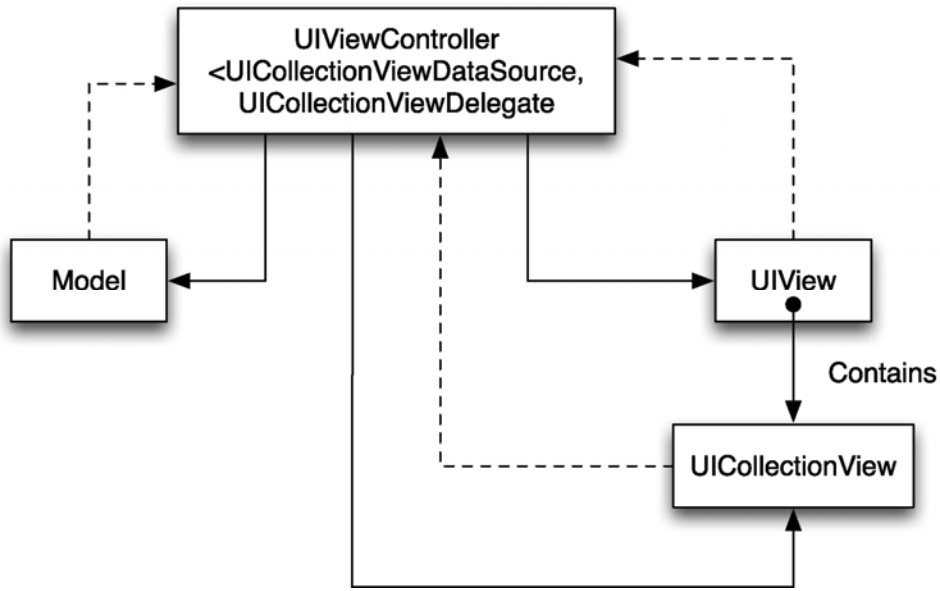


Figure 1.3 Example of MVC using `UICollectionView`'s protocols

This book uses the first approach unless there is a good reason not to. Even though the `view` property of `UICollectionViewController` is the same as its `collectionView` property, the code used in this book carefully distinguishes between the two.

Now that you've seen how collection views fit within the MVC paradigm of iOS apps, look at the following simple example. Don't worry; you experiment a lot with collection views in Chapter 2, "Displaying Content Using `UICollectionView`."

In the following example, you create a simple iPhone app that displays a bunch of cells with random colors. To get started, create a new application with the Single View template. Make sure that Use Storyboards is *unchecked*—this book focuses on collection views, and I don't want to have to diverge to discuss the peculiarities of storyboards. Delete everything in the view controller header file and replace it with the code in Listing 1.1.

Listing 1.1 Basic `UICollectionViewController` Header File

```
@interface AFViewController : UICollectionViewController

@end
```

Replace `AFViewController` with the name of your view controller. My initials are "AF," so I prefix my class names with them to avoid namespace collisions.

Next, head over to your .xib file and delete the view. Drag a collection view onto the blank canvas and connect the collection view's `delegate` and `dataSource` outlets to the File's Owner, the view controller. It should look like Figure 1.4 when you're done.

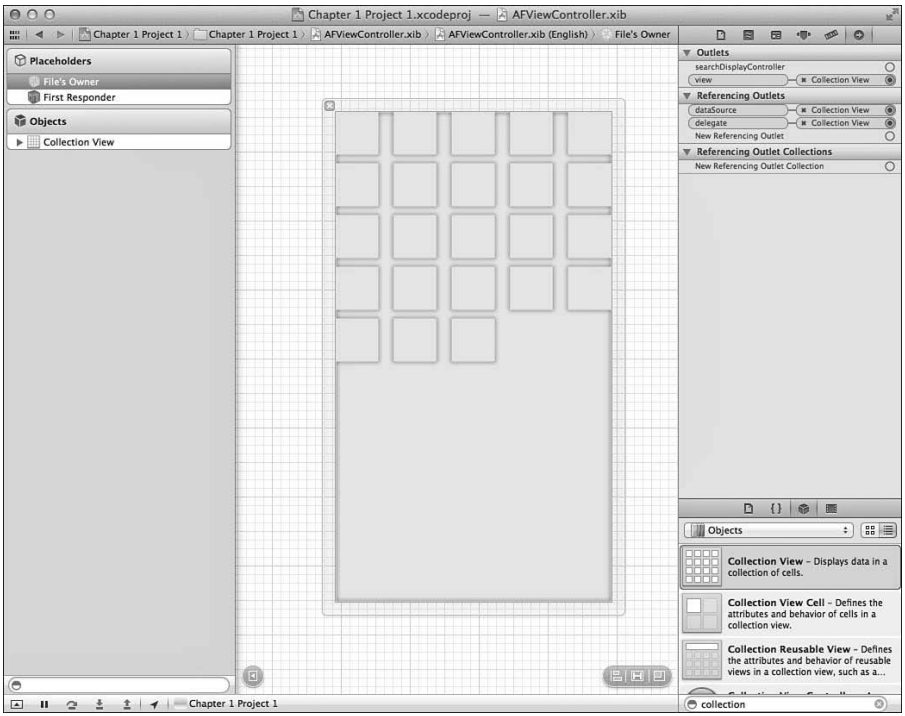


Figure 1.4 Basic `UICollectionView` setup using a .xib

Now comes the fun part: the code! `UICollectionViewDataSource` has two required methods. One returns the number of items in a section, and another configures a cell for a given index path.

If you're not familiar with these terms, don't worry. Chapter 2 explains everything in great detail. This quick example just gets your feet wet.

Following MVC, you need a model. Use a basic array that you'll populate with a bunch of randomly generated colors. The top of your implementation file should look something like Listing 1.2.

Listing 1.2 Setting Up the Model

```
static NSString *kCellIdentifier = @"Cell Identifier";


@implementation AFViewController
{
```

```
    NSArray *colorArray;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.collectionView registerClass:[UICollectionViewCell class] for-
CellWithReuseIdentifier:kCellIdentifier];

    const NSInteger numberOfColors = 100;

    NSMutableArray *tempArray = [NSMutableArray arrayWithCapac-
ity:numberOfColors];

    for (NSInteger i = 0; i < numberOfColors; i++)
    {
        CGFloat redValue = (arc4random() % 255) / 255.0f;
        CGFloat blueValue = (arc4random() % 255) / 255.0f;
        CGFloat greenValue = (arc4random() % 255) / 255.0f;

        [tempArray addObject:[UIColor colorWithRed:redValue
green:greenValue blue:blueValue alpha:1.0f]];
    }

    colorArray = [NSArray arrayWithArray:tempArray];
}
```

The `kCellIdentifier` string is used to register a plain `UICollectionViewCell` as the cell for the collection view to use, so don't pay much attention to it. The part that involves the model is the instance variable called `colorArray`. In `viewDidLoad`, you use a `for` loop to populate this array with random colors.

Now that you have the model set up, you need to configure your view to represent it. For this, use the two `UICollectionViewDataSource` methods mentioned earlier (see Listing 1.3).

Listing 1.3 Configuring the View

```
-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    return colorArray.count;
}

- (UICollectionViewCell *)collectionView:(UICollectionView
*)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewCell *cell = [collectionView dequeueReusableCell-
WithReuseIdentifier:kCellIdentifier forIndexPath:indexPath]; //Discussed
in Chapter 2 - pay no attention

    cell.backgroundColor = colorArray[indexPath.item];
```

```
    return cell;
}
```

The first method—`collectionView:numberOfItemsInSection:`—lets the collection view know how many cells it's going to display. You rely on the model to let the controller know what number to return. Next, you have `collectionView:cellForItemAtIndexPath:`, which returns a cell that you are responsible for configuring in a way that represents your model. To do this, you grab the model at the given index and use that color as the background color for the cell. If you run the app, you get something like what you see in Figure 1.5. Because the colors are randomly generated, of course, your app will look different.

So, this simple example demonstrates how a model can represent a view and how you can configure a view to represent that model without either being aware of the other. This example demonstrates the platonic ideal of what you should strive for: clear separation between model, view, and controller.

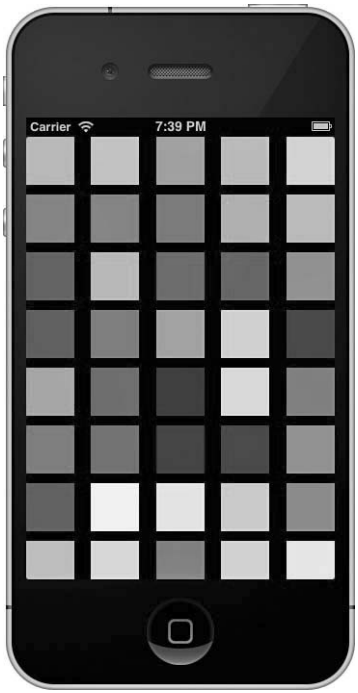


Figure 1.5 First run of the basic app