Creating iOS Apps

SECOND EDITION

DEVELOP AND **DESIGN**

Richard Warren

Creating iOS Apps

Second Edition

DEVELOP AND DESIGN

Richard Warren

PEACHPIT PRESS
WWW.PEACHPIT.COM

Creating iOS Apps: Develop and Design, Second Edition
Richard Warren

Peachpit Press
www.peachpit.com

To report errors, please send a note to errata@peachpit.com
Peachpit Press is a division of Pearson Education.

Copyright 2014 by Richard Warren

Editor: Kim Wimpsett
Production editor: Katerina Malone
Proofreader: Darren Meiss
Technical editors: Seulgi Kim and Aaron Daub
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry
Cover design: Aren Straiger
Interior design: Mimi Heft

Notice of Rights
All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability
The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the authors nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13: 978-0-321-93413-0
ISBN-10: 0-321-93413-X

Printed and bound in the United States of America

9 8 7 6 5 4 3 2 1

To my wife, Mika; my daughter, Haruko; and my son, Kai.
Thank you for your understanding and patience.
I owe each of you more than I could possibly express.
You’ve put up with a lot of extremely long hours and
general grumpiness over the last few months.
I really appreciate everything you’ve done to support me.
I couldn’t have made it without you.

ACKNOWLEDGEMENTS

I'd like to start by thanking all the students who have taken my classes and all the people who have visited my web site and posted questions or comments about the first edition of this book. You've given me a lot to think about, and a lot of the changes that I've made have come from your suggestions.

I would also like to thank the developers and Apple engineers on the Apple Developer forums. When working with new technologies, we often don't have an established set of best practices to fall back on. Having such an active group of people to discuss ideas and share notes with makes a huge difference.

Additionally, the book would not have been possible without the editors and production people at Peachpit. You've helped clarify my intentions and made the meaning behind my words come through clear. The layout and design of the new book look stunning, and I'm sure there are a thousand other things you've done that I know nothing about.

Mostly, I'd like to thank everyone who read and enjoyed the first edition. You made the second edition possible.

ABOUT THE AUTHOR

Richard splits his time between software development, teaching, and technical writing. He has a strong background in artificial intelligence research (machine learning and probabilistic decision making) and software engineering. Richard previously worked as a scientist for a small R&D company, developing Java software focused on computer vision and data analysis. Since escaping the Java world, Richard has spent the last three years focusing on hands-on, freelance consulting work for the iOS platform. During that time, he has developed a broad range of iOS apps with an emphasis on casual gaming. Richard is also the author of *Creating iOS 5 Apps: Develop and Design* (December, 2011, Peachpit Press) and *Objective-C Boot Camp: Foundation and Patterns for iOS Development* (July 2011, Peachpit Press). Additionally, he has written articles for *MacTech* magazine over the last seven years. Finally, Richard is one of the senior iOS and Objective-C instructors for About Objects.

CONTENTS

Welcome to Creating iOS 7 Apps viii

CHAPTER 1 **HELLO, iOS** **2**

An Introduction to iOS 4
Getting Started with iOS 6
Exploring Xcode 10
Surveying the Project 17
Modifying the Template 38
Wrapping Up 49
Other Resources 49

CHAPTER 2 **OUR APPLICATION'S ARCHITECTURE** **50**

Starting with the Basic Building Blocks 52
Kicking Off the Health Beat Project 55
Building the Model 63
Laying Out the Storyboard 80
Connecting the Model 89
Wrapping Up 106
Other Resources 107

CHAPTER 3 **DEVELOPING VIEWS AND VIEW CONTROLLERS** **108**

Initial Housekeeping 110
Rotations, Resizing, and Redrawing 111
Updating Health Beat 136
Wrapping Up 188
Other Resources 188

CHAPTER 4 **CUSTOM VIEWS AND CUSTOM TRANSITIONS** **190**

Custom Drawing 192
Custom Transitions 221
Wrapping Up 256
Other Resources 257

CHAPTER 5 **LOADING AND SAVING DATA** **258**

The iOS File System 260
Managing User Preferences 270
Saving the Weight History Document 282
Wrapping Up 310
Other Resources 311

CHAPTER 6 iCLOUD SYNCING . . . 312
What Is iCloud Syncing? . . . 314
iCloud APIs . . . 314
Adding Key-Value Syncing to Health Beat . . . 323
Adding Document Syncing to Health Beat . . . 332
Debugging iCloud Document Storage . . . 359
Wrapping Up . . . 361
Other Resources . . . 361

CHAPTER 7 CORE DATA . . . 362
Introducing Core Data . . . 364
Architecture Overview . . . 365
Core Data Performance . . . 386
iCloud Support . . . 389
Converting Health Beat . . . 393
Wrapping Up . . . 428
Other Resources . . . 429

CHAPTER 8 ADVANCED USER INTERFACES . . . 430
Creating Motion Effects . . . 432
Creating Physically Real Animations . . . 438
Wrapping Up . . . 468
Other Resources . . . 469

CHAPTER 9 THE LAST MILE . . . 470
The Final Touches . . . 472
Building for Distribution . . . 480
Wrapping Up . . . 482
Other Resources . . . 483

Index . . . 485

WELCOME TO CREATING iOS 7 APPS

This book serves two goals: introducing new developers to iOS development and educating experienced developers about the changes in iOS 7. We will examine a wide range of subjects—some new and some old—covering everything from building an initial iOS project to submitting your app to the iTunes App Store. For more information, including bonus chapters, sample code, and FAQs, check out `www.freelancemadscience.com/books`.

IOS 7 TECHNOLOGIES AND TOOLS

Over the course of this book, we will discuss the key technologies and development tools you will need to create high-quality iOS applications. Let's take a look at the most important of these.

STORYBOARDS

Storyboards allow us to rapidly design the relationships and segues between different scenes in our application. This lets us sketch out and edit our entire application's workflow very early in the design process. We can also show the storyboard to our clients, getting approval and feedback before we invest a significant amount of time in writing code.

AUTO LAYOUT

Creating a user interface that can automatically adapt to changes in size has become increasingly important. Whether you're responding to rotation events, trying to design an interface that works on both 3.5- and 4-inch phones or adjusting the user interface because of changes from Dynamic Type, Auto Layout makes creating truly adaptive interfaces possible. Furthermore, iOS 7 and Xcode 5 offer significantly better support for Auto Layout. If you haven't tried Auto Layout lately, you should look at it again.

ANIMATION

While visually stunning animations have always been part of the iOS experience, iOS 7's new user interface places a much greater emphasis on attractive, functional animations. The new, "flatter" look de-emphasizes visual ornamentation but replaces it with custom transitions and interactive animations. Furthermore, new technologies like `UIMotionEffects` and UIKit Dynamics help give our applications depth.

XCODE

Xcode provides an integrated development environment, giving us a wide range of tools to both create and manage our projects. It handles everything from running and debugging our code to designing our UI and data models. As iOS developers, we will spend most of our time in Xcode. Having a well-grounded understanding of its features is essential to our success.

iOS SIMULATOR

The iOS Simulator application lets us run, debug, test, and profile our code directly on a Mac. While the simulator will never replace testing on an actual device, it has a number of advantages. For example, developers can begin using the simulator before they join the iOS Developer Program. It's also easier and faster to run quick tests in the simulator, letting us perform rapid develop/test/iterate cycles as we add new features to our applications. The simulator also lets us trigger system events, such as memory warnings, that wouldn't be possible when testing on a device.

INSTRUMENTS

Apple's premier profiling tool, Instruments lets us dynamically trace and analyze code running either in the simulator or directly on an iOS device. It lets us track and record everything from object allocations and timer-based samples to file access and energy use. It also provides the tools necessary for sifting through and analyzing the data that it generates. In experienced hands, Instruments can help us to fix bugs and polish our code.

This page intentionally left blank

CHAPTER 4

Custom Views and Custom Transitions

This chapter will focus on customizing the user interface. First, we will look at drawing. By drawing our own views, we can create a wide range of visual effects—and by extension, a wide range of custom controls. Then we will look at setting the transitions between scenes and providing animations for those transitions.

In previous versions of iOS, custom drawing grew to become a large part of many applications. As developers tried to give their applications a unique look and feel or tried to push beyond the boundaries of the UIKit controls, they began to increasingly rely on custom drawing to make their applications stand out.

However, iOS 7 is different. In iOS 7, Apple recommends pushing the application's user interface into the background. It should, for the most part, disappear, focusing attention on the user's data rather than drawing attention to itself. As a result, the developers are encouraged to create their own unique look and feel through less visually obvious techniques. In this new paradigm, custom animations may replace custom interfaces as the new distinguishing feature.

CUSTOM DRAWING

In previous chapters, we set up the `GraphViewController` and made sure that it received a copy of our `weightHistoryDocument`. However, it is still just displaying an empty, white view. Let's change that. We're going to create a custom view with custom drawing—actually, to more easily handle the tab bar and status bar, we will be creating two custom views and split our drawing between them.

Let's start with the background view. This will be the backdrop to our graph. So, let's make it look like a sheet of graph paper.

DRAWING THE BACKGROUND VIEW

Create a new Objective-C class in the Views group. Name it **BackgroundView**, and make it a subclass of **UIView**. You should be an expert at creating classes by now, so I won't walk through all the steps. However, if you need help, just follow the examples from previous chapters.

There are a number of values we will need when drawing our graph grid. For example, how thick are the lines? How much space do we have between lines, and what color are the lines?

We could hard-code all this information into our application; however, I prefer to create properties for these values and then set up default values when the view is instantiated. This gives us a functional view that we can use right out of the box—but also lets us programmatically modify the view, if we want.

Open `BackgroundView.h`. Add the following properties to the header file:

```
@property (assign, nonatomic) CGFloat gridSpacing;
@property (assign, nonatomic) CGFloat gridLineWidth;
@property (assign, nonatomic) CGFloat gridXOffset;
@property (assign, nonatomic) CGFloat gridYOffset;
@property (strong, nonatomic) UIColor *gridLineColor;
```

Now, open `BackgroundView.m`. The template automatically creates an `initWithFrame:` method for us, as well as a commented-out `drawRect:`. We will use both of these.

Let's start by setting up our default values. Views can be created in one of two ways. We could programmatically create the view by calling `alloc` and one of its `init` methods. Eventually, this will call our `initWithFrame:` method, since that is our designated initializer. More likely, however, we will load the view from a nib file or storyboard. In this case, `initWithFrame:` is not called. The system calls `initWithCoder:` instead.

To be thorough, we should have both an `initWithFrame:` method and an `initWithCoder:` method. Furthermore, both of these methods should perform the same basic setup steps. We will do this by moving the real work to a private method that both init methods can then call.

AWAKEFROMNIB

When loading objects from a nib, you may want to perform any additional configuration and setup in the object's `awakeFromNib` method. Remember, `initWithCoder:` is called when the object is first instantiated. Then all of its outlets and actions are connected. Finally, `awakeFromNib` is called. This means you cannot access any of the object's outlets in the `initWithCoder:` method. They are, however, available in `awakeFromNib`.

In our case, we don't have any outlets. Also, I like the symmetry between `initWithFrame:` and `initWithCoder:`. Still, `awakeFromNib` can be a safer location for additional configuration.

Add the following line to the existing `initWithFrame:` method:

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self setDefaults];
    }
    return self;
}
```

This calls our (as yet undefined) `setDefaults` method. Now, just below `initWithFrame:`, create the `initWithCoder:` method as shown here:

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
    self = [super initWithCoder:aDecoder];
    if (self) {
        [self setDefaults];
    }
    return self;
}
```

We discussed `initWithCoder:` in the "Secret Life of Nibs" section of Chapter 2, "Our Application's Architecture." It is similar to, but separate from, our regular initialization chain. Generally speaking, `initWithCoder:` is called whenever we load an object from disk, and nibs and storyboards get compiled into binary files, which the system loads at runtime. Not surprising, we will see `initWithCoder:` again in Chapter 5, "Loading and Saving Data."

Now, we just need to create our `setDefaults` method. Create the following code at the bottom of our implementation block:

```
#pragma mark - Private Methods

- (void)setDefaults
{
    self.backgroundColor = [UIColor whiteColor];
    self.opaque = YES;

    self.gridSpacing = 20.0;

    if (self.contentScaleFactor == 2.0)
    {
        self.gridLineWidth = 0.5;
        self.gridXOffset = 0.25;
        self.gridYOffset = 0.25;
    }
    else
    {
        self.gridLineWidth = 1.0;
        self.gridXOffset = 0.5;
        self.gridYOffset = 0.5;
    }

    self.gridLineColor = [UIColor lightGrayColor];
}
```

We start by setting our view's background color to clear. This will override any background color we set in Interface Builder. I often use this trick when working with custom views, since they will not get drawn in Interface Builder. By default, all views appear as a rectangular area on the screen. Often we need to set a temporary background color, since the views otherwise match the parent's background color or are clear (letting the parent view show through). This, effectively, makes them invisible in Interface Builder.

By setting this property when the view is first loaded, we can freely give it any background color we want in Interface Builder. This make our custom views easier to see while we work on them. Plus, by setting the value when the view is created, we can still override it programmatically in the controller's `viewDidLoad` if we want.

Next, we set the view to opaque. This means we promise to fill the entire view from corner to corner with opaque data. In our case, we're providing a completely opaque background color. If we programmatically change the background color to a translucent color, we should also change the opaque property.

OPAQUE VIEWS

Each view has an `opaque` property. This property acts as a hint to the drawing system. If the property is set to `YES`, the drawing system will assume the view is completely opaque and will optimize the drawing code appropriately. If it is set to `NO`, the drawing system will composite this view with any underlying views. Typically, this means checking pixel by pixel and blending multiple values for each pixel, if necessary. This allows for more elaborate visual effects—but also increases the computational cost.

By default, `opaque` is set to `YES`. We should avoid changing this unless we actually need transparent or translucent areas within our view.

However, having an opaque view means our custom drawing code must completely fill the provided rectangle. We cannot leave any transparent or semi-transparent regions that might let underlying views show through. The simplest solution is to assign an opaque `backgroundColor`, which will implicitly fill the entire view's rectangle.

Next, we check the view's scale factor. The default value we use for our line width, the x-offset and the y-offset, all vary depending on whether we are drawing to a regular or a Retina display.

All the iPhones and iPod touches that support iOS 7 have Retina displays. However, the iPad 2 and the iPad mini still use regular displays—so it's a good idea to support both resolutions. After all, even if we are creating an iPhone-only app, it may end up running on an iPad.

Additionally, Apple may release new devices that also have non-Retina displays. The iPad mini is a perfect example. It has the same resolution as a regular size iPad 2; it just shrinks the pixel size down by about 80 percent. Also, like the iPad 2, it uses a non-Retina display, presumably to reduce cost and improve performance and energy use.

We want to draw the smallest line possible. This means the line must be 1-pixel wide, not 1-point wide. We, therefore, have to check the scale and calculate our line width appropriately. Furthermore, we also need to make sure our lines are properly aligned with the underlying pixels. That means we have to offset them by half the line's width (or, in this case, half a pixel).

Now we need to draw our grid. Uncomment the `drawRect:` method. The system calls this method during each display phase, as discussed in Chapter 3, "Developing Views and View Controllers." Basically, each time through the run loop, the system will check all the views in the view hierarchy and see whether they need to be redrawn. If they do, it will call their `drawRect:` method.

Also as mentioned in the previous chapter, our views aggressively cache their content. Therefore, most normal operations (changing their frame, rotating them, covering them up, uncovering them, moving them, hiding them, or even fading them in and out) won't force the views to be redrawn. Most of the time this means our view will be drawn once when it is first created. It may be redrawn if our system runs low on memory while it's offscreen. Otherwise, the system will just continue to use the cached version.

DRAWING FOR THE RETINA DISPLAY

As you probably know, the iPhone's Retina display has a 960x640 display. This is four times the number of pixels as the 3GS and earlier models. This could result in a lot of complexity for developers—if we had to constantly test for the screen size and alter our drawing code to match. Fortunately, Apple has hidden much of this complexity from us.

All the native drawing functions (Core Graphics, UIKit, and Core Animation) use a logical coordinate system measured in points. A point is approximately 1/160 of an inch. It's important to note that these points may or may not be the same as the screen's pixels. Draw a 1-point-wide line on a regular display, and you get a 1-pixel-wide line. Draw the same line on a Retina display, and it is now 2 pixels wide. This also means a full-screen frame is the same size, regardless of the display type. We still have three screen sizes to deal with: iPhone 3.5-inch, iPhone 4-inch, and iPad. But, three is better than five or six.

A device's scale factor gives us the conversion between points and pixels. You can access the `scale` property from the `UIScreen`, `UIView`, `UIImage`, or `CALayer` classes. This allows us to perform any resolution-dependent processing. However, we actually get a lot of support for free.

- All standard UIKit views are automatically drawn at the correct resolution.
- Vector-based drawings (e.g., `UIBezierPath`, `CGPathRef`, and PDFs) automatically take advantage of the higher resolution to produce smoother lines.
- All text is automatically rendered at the higher resolution.

There are, however, some steps we still need to take to fully support multiple screen resolutions. One obvious example occurs when loading and displaying images and other bitmapped art. We need to create higher-resolution copies of these files for the Retina display. Fortunately, UIKit supports automatically loading the correct resolution from the assets category.

Let's say you have a 20-pixel by 30-pixel image named `stamp.png`. You also need to create a Retina version that is 40 x 60 pixels. Now open the `Images.xcassets`. The catalog editor has two parts: a list of all your assets on the left and a detail view on the right.

Drag and drop your original image into the assets list. Xcode will automatically make a new entry for it called stamp. Now select the stamp entry. The detail view will show you the 1x version of your image, with a placeholder for the 2x. Drag the Retina version into the 2x position. The image is now ready to use. You just need to request the image by the asset's name, as shown here:

```
UIImage* stampImage =
[UIImage imageNamed:@"stamp"];
```

UIKit will automatically load the correct version for your device.

Similarly, if we are creating bitmaps programmatically, we will want to make sure we give them the correct scale factor. The `UIGraphicsBeginImageContext` function creates a bitmap-based graphics context with a 1.0 scale factor. Instead, use the `UIGraphicsBeginImageContextWithOptions` function and pass in the correct scale (which you can access by calling `[[UIScreen mainScreen] scale]`).

Core Animation layers may also need explicit support. Whenever you create a new `CALayer` (one that is not already associated with a view), it comes with a default scale value of 1.0. If you then draw this layer onto a Retina display, it will automatically scale up to match the screen. You can prevent this by manually changing the layer's scale factor and then providing resolution-appropriate content.

Finally, OpenGL ES also uses a 1.0 scale by default. Everything will still draw on the Retina display, but it will be scaled up and may appear blocky (especially when compared to properly scaled drawings). To get full-scale rendering, we must increase the size of our render buffers. This, however, can have severe performance implications. Therefore, we must make these changes manually. Changing a view's `contentScaleFactor` will automatically alter the underlying `CAEAGLLayer`, increasing the size of the render buffers by the scale factor.

It is also important to test your drawing code on all the devices you intend to support. Drawing a 0.5-point-wide line may look fine on a retina display but appear large and fuzzy on a regular display. Similarly, higher-resolution resources use more memory and may take longer to load and process. This is particularly true for full-screen images.

Even with all these exceptions and corner cases, iOS's logical coordinate system greatly simplifies creating custom drawing code that works across multiple resolutions. Most of the time we get automatic Retina support without doing anything at all.

SUBPIXEL DRAWING

All of our drawing routines use `CGFloat`, `CGPoint`, `CGSize`, and `CGRect`, and all of these use floating-point values. That means we can draw a 0.34-pixel-wide line at y = 23.428. Obviously, however, our screen cannot display partial pixels.

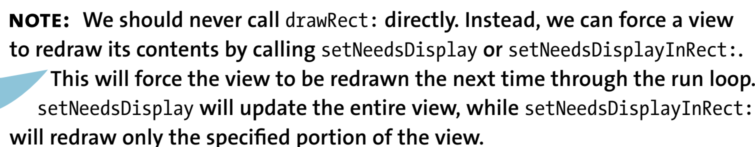
Instead, our drawing engine simulates subpixel resolutions using anti-aliasing—proportionally blending the drawn line with the background color in each partial pixel. Anti-aliasing makes lines look smoother, especially curved or diagonal lines. It fills in the otherwise harsh, jagged edges. Unfortunately, it can also make our drawings appear softer or somewhat fuzzy.

By default, anti-aliasing is turned on for any window or bitmap context. It is turned off for any other graphics contexts. However, we can explicitly set our context's anti-aliasing using the `CGContextSetShouldAntiAlias()` function.

It's also important to realize that the graphics context's coordinates fall in between the pixels, and the drawing engine centers the lines on their endpoints. For example, if you draw a 1-point-wide horizontal line at y = 20, on a non-Retina display (iPad 2 or iPad mini, for example), the line will actually be drawn half in pixel row 19 and half in row 20. If anti-aliasing is turned on, this will result in two rows of pixels, each with a 50 percent blend. If anti-aliasing is off, you will get a 2-pixel-wide line. You can fix this by offsetting the drawing coordinates by half a point (y = 20.5). Then the line will fall exactly along the row of pixels.

For a Retina display, the 1-point line will result in a 2-pixel line that properly fills both rows on either side of the y-coordinate. A 0.5 pixel-wide line, though, will similarly need to be offset by 0.25 pixels. For more information on how Retina displays and scale factors work, see the "Drawing for the Retina Display" sidebar.

Additionally, when the system does ask us to draw or redraw a view, it typically asks us to draw the entire view. The `rect` passed into `drawRect:` will be equal to the view's bounds. The only time this isn't true is when we start using tiled layers or when we explicitly call `setNeedsDisplayInRect:` and specify a subsection of our view.



NOTE: We should never call `drawRect:` directly. Instead, we can force a view to redraw its contents by calling `setNeedsDisplay` or `setNeedsDisplayInRect:`. This will force the view to be redrawn the next time through the run loop. `setNeedsDisplay` will update the entire view, while `setNeedsDisplayInRect:` will redraw only the specified portion of the view.

Drawing is expensive. If you're creating a view that updates frequently, you should try to minimize the amount of drawing your class performs with each update. Write your `drawRect:` method so that it doesn't do unnecessary drawing outside the specified rect. You should also use `setNeedsDisplayInRect:` to redraw the smallest possible portion. However, for Health Beat, our views display largely static data. The graph is updated only when a new

entry is added. As a result, we can take the easier approach, disregard the `rect` argument, and simply redraw the entire view.

Implement `drawRect:` as shown here:

```
- (void)drawRect:(CGRect)rect
{
    CGFloat width = CGRectGetWidth(self.bounds);
    CGFloat height = CGRectGetHeight(self.bounds);

    UIBezierPath *path = [UIBezierPath bezierPath];
    path.lineWidth = self.gridLineWidth;

    CGFloat x = self.gridXOffset;
    while (x <= width)
    {
        [path moveToPoint:CGPointMake(x, 0.0)];
        [path addLineToPoint:CGPointMake(x, height)];
        x += self.gridSpacing;
    }

    CGFloat y = self.gridYOffset;
    while (y <= height)
    {
        [path moveToPoint:CGPointMake(0.0, y)];
        [path addLineToPoint:CGPointMake(width, y)];
        y += self.gridSpacing;
    }

    [self.gridLineColor setStroke];
    [path stroke];
}
```

We start by using some of the CGGeometry functions to extract the width and height from our view's bounds. Then we create an empty `UIBezierPath`.

Bezier paths let us mathematically define shapes. These could be open shapes (basically lines and curves) or closed shapes (squares, circles, rectangles, and so on). We define the path as a series of lines and curved segments. The lines have a simple start and end point. The curves have a start and end point and two control points, which define the shape of the curve. If you've ever drawn curved lines in a vector art program, you know what I'm talking about. We can use our `UIBezierPath` to draw the outline (stroking the path) or to fill in the closed shape (filling the path).

FIGURE 4.1 Our background view

UIBezierPath also has a number of convenience methods to produce ready-made shapes. For example, we can create paths for rectangles, ovals, rounded rectangles, and arcs using these convenience methods.

For our grid, the path will just be a series of straight lines. We start with an empty path and then add each line to the path. Then we draw it to the screen.

Notice, when we create the path, we set the line's width. Then we create a line every 20 points, both horizontally and vertically. We start by moving the cursor to the beginning of the next line by calling `moveToPoint:`, and then we add the line to the path by calling `addLineToPoint:`. Once all the lines have been added, we set the line color by calling `UIColor`'s `setStroke` method. Then we draw the lines by calling the path's `stroke` method.

We still need to tell our storyboard to use this background view. Open `Main.storyboard` and zoom in on the Graph View Controller scene. Select the scene's `view` object (either in the Document Outline or by clicking the center of the scene), and in the Identity inspector, change Class to **BackgroundView**.

Just like in Chapter 3, the class should be listed in the drop-down; however, sometimes Xcode doesn't correctly pick up all our new classes. If this happens to you, you can just type in the name, or you can quit out of Xcode and restart it—forcing it to update the class lists.

Run the application and tap the Graph tab. It should display a nice, graph paper effect (**Figure 4.1**).

FIGURE 4.2 The background view is stretched when rotated.

There are a couple of points worth discussing here. First, notice how our drawing stretches up under the status bar. Actually, it's also stretched down below the tab bar, but the tab bar blurs and fades the image quite a bit, so it's not obvious.

One of the biggest changes with iOS 7 is the way it handles our bars. In previous versions of iOS, our background view would automatically shrink to fit the area between the bars. In iOS 7, it extends behind the bars by default.

We can change this, if we want. We could set the view controller's `edgesForExtendedLayout` to `UIRectEdgeNone`. In our view, this would prevent our background view from extending below the tab bar. It will also make the tab bar opaque. However, this won't affect the status bar.

We actually want our background view to extend behind our bars. It gives our UI a sense of context. However, we want the actual content to be displayed between the bars. To facilitate this, we will draw our actual graph in a separate view, and we will use auto layout to size that view properly between the top and bottom layout guides.

Also notice, the status bar does not have any background color at all. It is completely transparent. We can set whether it displays its contents using a light or a dark font—but that's it. If we want to blur or fade out the content behind the bar, we need to provide an additional view to do that for us. However, I will leave that as a homework exercise.

There is one slight problem with our background view, however. Run the app again and try rotating it. When we rotate our view, auto layout will change its frame to fill the screen horizontally. However, as I said earlier, this does not cause our view to redraw. Instead, it simply reuses the cached version, stretching it to fit (**Figure 4.2**).

We could fix this by calling `setNeedsDisplay` after our view rotates—but there's an easier way. Open `BackgroundView.m` again. Find `setDefaults`, and add the following line after `self.opaque = YES`:

```
self.contentMode = UIViewContentModeRedraw;
```

The content mode is used to determine how a view lays out its content when its bounds change. Here, we are telling our view to automatically redraw itself. We can still move the view, rotate it, fade it in, or cover it up. None of that will cause it to redraw. However, change the size of its `frame` or `bounds`, and it will be redrawn to match.

Run the app again. It will now display correctly as you rotate it.

DRAWING THE LINE GRAPH

Now let's add the line graph. Let's start by creating a new `UIView` subclass named `GraphView`. Place this in the Views group with our other views.

Before we get to the code, let's set up everything in Interface Builder. Open `Main.storyboard`. We should still be zoomed into the Graph View Controller scene.

Drag a new view out and place it in the scene. Shrink it so that it is a small box in the middle of the scene and then change its Class to **GraphView** and its background to **lightGray**. This will make it easier to see (**Figure 4.3**).

We want to resize it so that it fills the screen from left to right, from the top layout guide to the bottom. However, it will be easier to let Auto Layout do that for us. With the graph view selected, click the Pin tool. We want to use the Add New Constraints controls to set the constraints shown in **Table 4.1**. Next, set Update Frames to **Items of New Constraints**, and click the Add 4 Constraints button. The view should expand as desired.

Run the app and navigate to our graph view. It should fill the screen as shown in **Figure 4.4**. Rotate the app and make sure it resizes correctly.

TABLE 4.1 Graph View's Constraints

SIDE	TARGET	CONSTANT
Left	Background View	0
Top	Top Layout Guide	0
Right	Background View	0
Bottom	Background View*	49*

* Ideally, we would like to pin the bottom of our graph view to the Bottom Layout Guide with a spacing constant of 0. This should align the bottom of our graph view with the top of the tab bar. If the tab bar's height changes or if we decide to remove the tab bar, the Bottom Layout Guide would automatically adjust.

However, as of the Xcode 5.0 release, there is a bug in the Bottom Layout Guide. When the view first appears it begins aligned with the bottom of the background view, not the top of the tab bar. If you rotate the view, it adjusts to the correct position. Unfortunately, until this bug is fixed, we must hard-code the proper spacing into our constraints. That is the only way to guarantee that our graph view gets laid out properly.

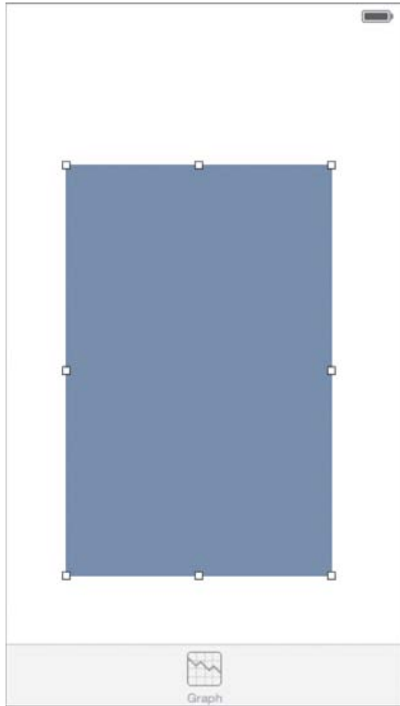


FIGURE 4.3 Adding the view

FIGURE 4.4 The properly sized graph view

There's one last step before we get to the code. We will need an outlet to our graph view. Switch to the Assistant editor, Control-drag from the graph view to `GraphViewController.m`'s class extension, and create the following outlet:

```
@property (weak, nonatomic) IBOutlet GraphView *graphView;
```

Unfortunately, this creates an "Unknown type name 'GraphView'" error. We can fix this by scrolling to the top of `GraphViewController.m` and importing `GraphView.h`.

Now, switch back to the Standard editor, and open `GraphView.h`. We want to add a number of properties to our class. First, we need an `NSArray` to contain the weight entries we intend to graph. We also need to add a number of properties to control our graph's appearance. This will follow the general pattern we used for the background view.

Add the following properties to GraphView's public header:

```
// This must be an array sorted by date in ascending order.
@property (strong, nonatomic) NSArray *weightEntries;

@property (assign, nonatomic) CGFloat margin;

@property (assign, nonatomic) CGFloat guideLineWidth;
@property (assign, nonatomic) CGFloat guideLineYOffset;
@property (strong, nonatomic) UIColor *guideLineColor;

@property (assign, nonatomic) CGFloat graphLineWidth;
@property (assign, nonatomic) CGFloat dotSize;
```

Notice that we're assuming our weight entries are sorted in ascending order. When we implement our code, we'll add a few sanity checks to help validate the incoming data—but we won't do an item-by-item check of our array. Instead, we will assume that the calling code correctly follows the contract expressed in our comment. If it doesn't, the results will be unpredictable.

We've seen this methodology before. Objective-C often works best when you clearly communicate your intent and assume other developers will follow that intent. Unfortunately, in this case, we don't have any syntax tricks that we can use to express our intent. We must rely on comments. This is, however, an excellent use case for documentation comments. I'll leave that as an exercise for the truly dedicated reader.

Switch to `GraphView.m`. Just like our background view, we want the pair of matching `init`… methods, with a private `setDefaults` method. The `init`… methods are shown here:

```
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        [self setDefaults];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)aDecoder
{
    self = [super initWithCoder:aDecoder];
    if (self) {
        [self setDefaults];
    }
    return self;
}
```

And, at the bottom of the implementation block, add the `setDefaults` method.

```
#pragma mark - Private Methods

- (void)setDefaults
{
    self.contentMode = UIViewContentModeRedraw;
    self.backgroundColor = [UIColor clearColor];
    self.opaque = NO;

    self.margin = 20.0;

    if (self.contentScaleFactor == 2.0)
    {
        self.guideLineYOffset = 0.0;
    }
    else
    {
        self.guideLineYOffset = 0.5;
    }

    self.guideLineWidth = 1.0;
    self.guideLineColor = [UIColor darkGrayColor];

    self.graphLineWidth = 2.0;
    self.dotSize = 4.0;
}
```

This code is almost the same as our background view. Yes, the property names and values have changed, but the basic concept is the same. Notice that we are using a clear `backgroundColor`, letting our background view show through. We have also set the opaque property to YES.

Additionally, since our graph line's width is an even number of pixels, we don't need to worry about aligning it to the underlying pixels. If it's drawn as a vertical or horizontal line, it will already correctly fill 1 pixel above and 1 pixel below the given coordinates (in a non-Retina display). By a similar logic, the 1-point guidelines need to be offset only in non-Retina displays.

The app should successfully build and run. However, notice that our previously gray graph view has completely vanished. Now we just need to add some custom drawing, to give it a bit of content.

For the purpose of this book, we want to keep our graph simple. So, we will just draw three horizontal guidelines with labels and the line graph itself. Also, we're not trying to match the graph with the background view's underlying graph paper. In a production app, you would probably want to add guidelines or labels for the horizontal axis, as well as coordinate the background and graph views, giving the graph paper actual meaning.

PASSING THE DATA

The graph will be scaled to vertically fill our space, with the maximum value near the top of our view and the minimum near the bottom. It will also horizontally fill our view, with the earliest dates close to the left side and the latest date close to the right.

But, before we can do this, we need to pass in our weight entry data. Open `GraphViewController.m`. First things first—we need to import `WeightHistoryDocument.h`. We already added a forward declaration in the `.h` file, but we never actually imported the class. It hasn't been necessary, since we haven't used it—until now.

Then, add a `viewWillAppear:` method, as shown here:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    self.graphView.weightEntries =
    [[[self.weightHistoryDocument allEntries]
       reverseObjectEnumerator] allObjects];
}
```

As our `GraphView`'s comments indicated, we want to pass our weight entries in ascending order. However, our document stores them in descending order. So, we have to reverse this array.

We grab all the entries from our weight history document. We then access the reverse enumerator for these entries. The reverse enumerator lets us iterate over all the objects in the reverse order. Finally we call the enumerator's `allObjects` method. This returns the remaining objects in the collection. Since we haven't enumerated over any objects yet, it returns the entire array. This is a quick technique for reversing any array.

Of course, we could have just passed the entire weight history document to our graph view, but isolating an array of entries has several advantages. First, as a general rule, we don't want to give any object more access to our model than is absolutely necessary. More pragmatically, writing the code this way gives us a lot of flexibility for future expansions. We could easily modify our code to just graph the entries from last week, last month, or last year. We'd just need to filter our entries before we passed them along.

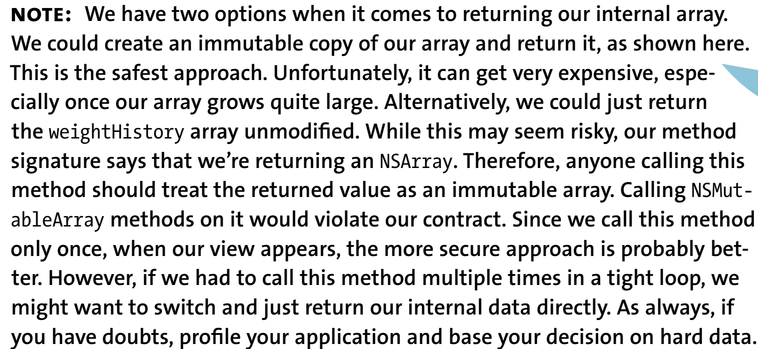
Unfortunately, there is one small problem with this approach. `WeightHistoryDocument` doesn't have an `allEntries` method. Let's fix that. Open `WeightHistoryDocument.h` and declare `allEntries` just below `weightEntriesAfter:before:`.

```
- (NSArray *)allEntries;
```

Now switch to WeightHistoryDocument.m. Implement allEntries as shown here:

```
- (NSArray *)allEntries
{
    return [self.weightHistory copy];
}
```

Here, we're making a copy of our history array and returning the copy. This avoids exposing our internal data, and prevents other parts of our application from accidentally altering it.



NOTE: We have two options when it comes to returning our internal array. We could create an immutable copy of our array and return it, as shown here. This is the safest approach. Unfortunately, it can get very expensive, especially once our array grows quite large. Alternatively, we could just return the weightHistory array unmodified. While this may seem risky, our method signature says that we're returning an NSArray. Therefore, anyone calling this method should treat the returned value as an immutable array. Calling NSMutableArray methods on it would violate our contract. Since we call this method only once, when our view appears, the more secure approach is probably better. However, if we had to call this method multiple times in a tight loop, we might want to switch and just return our internal data directly. As always, if you have doubts, profile your application and base your decision on hard data.

Now, switch back to GraphView.m. There are a number of private properties we want to add. Some of these will hold statistical data about our entries. Others will hold layout data—letting us easily calculate these values once and then use them across a number of different drawing methods.

At the top of GraphView.m, import WeightEntry.h, and then add a class extension as shown here:

```
#import "WeightEntry.h"

@interface GraphView ()

@property (assign, nonatomic) float minimumWeight;
@property (assign, nonatomic) float maximumWeight;
@property (assign, nonatomic) float averageWeight;
@property (strong, nonatomic) NSDate *earliestDate;
@property (strong, nonatomic) NSDate *latestDate;

@property (assign, nonatomic) CGFloat topY;
@property (assign, nonatomic) CGFloat bottomY;
@property (assign, nonatomic) CGFloat minX;
@property (assign, nonatomic) CGFloat maxX;

@end
```

Now, let's add a custom setter for our `weightEntries` property. This will set the statistical data any time our entries change.

```
#pragma mark - Accessor Methods

- (void)setWeightEntries:(NSArray *)weightEntries
{
    _weightEntries = weightEntries;

    if ([_weightEntries count] > 0)
    {
        self.minimumWeight =
        [[weightEntries valueForKeyPath:@"@min.weightInLbs"] floatValue];

        self.maximumWeight =
        [[weightEntries valueForKeyPath:@"@max.weightInLbs"] floatValue];

        self.averageWeight =
        [[weightEntries valueForKeyPath:@"@avg.weightInLbs"] floatValue];

        self.earliestDate =
        [weightEntries valueForKeyPath:@"@min.date"];

        self.latestDate =
        [weightEntries valueForKeyPath:@"@max.date"];

        NSAssert([self.latestDate isEqualToDate:
                 [[self.weightEntries lastObject] date]],
                @"The weight entry array must be "
                @"in ascending chronological order");

        NSAssert([self.earliestDate isEqualToDate:
                 [self.weightEntries[0] date]],
                @"The weight entry array must be "
                @"in ascending chronological order");
    }
    else
    {
        self.minimumWeight = 0.0;
        self.maximumWeight = 0.0;
        self.averageWeight = 0.0;
```

```
        self.earliestDate = nil;
        self.latestDate = nil;
    }


    [self setNeedsDisplay];
}
```

Ideally, nothing in this method comes as a surprise. We assign the incoming value to our property. Then we check to see whether our new array has any elements. If it does, we set up our statistical data. Here, we're using the KVC operators we used in the "Updating Our View" section of Chapter 3. Notice that we can use them for both the weight and the date properties. `@min.date` gives us the earliest date in the array, while `@max.date` gives us the latest. We also check the first and last object and make sure the first object is our earliest date and the last object is our latest date. This acts as a quick sanity check. Obviously it won't catch every possible mistake—but it is fast, and it will catch most obvious problems (like forgetting to reverse our array). As a bonus, it works even if our array has only one entry.

If our array is empty (or has been set to `nil`), we simply clear out the stats data.

Now, let's draw the graph. Drawing code can get very complex, so we're going to split it into a number of helper methods.

Start by adding methods to calculate our graph size. We'll be storing these values in instance variables. I'm always a bit uncomfortable when I store temporary data in an instance variable. I like to clear it out as soon as I'm done—preventing me from accidentally misusing the data when it's no longer valid. So, let's make a method to clean up our data as well.

Add these private methods to the bottom of our implementation block:

```
- (void)calculateGraphSize
{
    CGRect innerBounds =
    CGRectInset(self.bounds, self.margin, self.margin);
    self.topY = CGRectGetMinY(innerBounds) + self.margin;
    self.bottomY = CGRectGetMaxY(innerBounds);
    self.minX = CGRectGetMinX(innerBounds);
    self.maxX = CGRectGetMaxX(innerBounds);
}

- (void)clearGraphSize
{
    self.topY = 0.0;
    self.bottomY = 0.0;
    self.minX = 0.0;
    self.maxX = 0.0;
}
```

The calculateGraphSize shows off more of the CGGeometry functions. Here, we use CGRectInset to take a rectangle and shrink it. Each side will be moved in by our margin. Given our default values, this will create a new rectangle that is centered on the old one but is 40 points shorter and 40 points thinner.

You can also use this method to make rectangles larger, by using negative inset values.

Then we use the CGRectGet… methods to pull out various bits of data. The topY value deserves special attention. We want the top margin to be twice as big as the bottom, left, and right. Basically, we want all of our content to fit inside the innerBounds rectangle. We will use the topY and bottomY to draw our minimum and maximum weight guidelines. However, the top guideline needs a little extra room for its label.

The clearGraphSize method is much simpler; we just set everything to 0.0.

Now, let's add a private drawGraph method.

```
- (void)drawGraph
{
    // if we don't have any entries, we're done
    if ([self.weightEntries count] == 0) return;

    UIBezierPath *barGraph = [UIBezierPath bezierPath];
    barGraph.lineWidth = self.graphLineWidth;

    BOOL firstEntry = YES;
    for (WeightEntry *entry in self.weightEntries) {

        CGPoint point =
        CGPointMake([self calculateXForDate:entry.date],
                    [self calculateYForWeight:entry.weightInLbs]);

        if (firstEntry)
        {
            [barGraph moveToPoint:point];
            firstEntry = NO;
        }
        else
        {
            [barGraph addLineToPoint:point];
        }

        if (entry.weightInLbs == self.maximumWeight)
        {
```

```
            [self drawDotForEntry:entry];
        }

        if (entry.weightInLbs == self.minimumWeight)
        {
            [self drawDotForEntry:entry];
        }
    }

    [self.tintColor setStroke];
    [barGraph stroke];
}
```

We start by checking to see whether we have any entries. If we don't have any entries, we're done. There's no graph to draw.

If we do have entries, we create an empty Bezier path to hold our line graph. We iterate over all the entries in our array. We calculate the correct x- and y-coordinates for each entry, based on their weight and date. Then, for the first entry we move the cursor to the correct coordinates. For every other entry, we add a line from the previous coordinates to our new coordinates. If our entry is our minimum or maximum value, we draw a dot at that location as well. Finally we draw the line. This time, we're using our view's `tintColor`. We discussed `tintColor` in the "Customizing Our Appearance" section of Chapter 3. Basically, `graphView`'s `tintColor` will default to our window's tint color—but we could programmatically customize this view by giving it a different `tintColor`.

Let's add the methods to calculate our x- and y-coordinates. Start with the x-coordinate.

```
- (CGFloat) calculateXForDate:(NSDate *)date
{
    NSAssert([self.weightEntries count] > 0,
             @"You must have more than one entry "
             @"before you call this method");

    if ([self.earliestDate compare:self.latestDate] == NSOrderedSame )
    {
        return (self.maxX + self.minX) / (CGFloat)2.0;
    }

    NSTimeInterval max =
    [self.latestDate timeIntervalSinceDate:self.earliestDate];

    NSTimeInterval interval =
```

```
    [date timeIntervalSinceDate:self.earliestDate];

    CGFloat width = self.maxX - self.minX;
    CGFloat percent = (CGFloat)(interval / max);
    return percent * width + self.minX;
}
```

We start with a quick sanity check. This method should never be called if we don't have any entries. Then we check to see whether our earliest date and our latest date are the same. If they are (for example, if we have only one date), they should be centered in our graph. Otherwise, we need to convert our dates into numbers that we can perform mathematical operations on. The easiest way to do this is using the `timeIntervalSinceDate:` method. This will return a `double` containing the number of seconds between the current date and the specified dates.

We can then use these values to calculate each date's relative position between our earliest and latest dates. We calculate a percentage from 0.0 (the earliest date) to 1.0 (the latest date). Then we use that percentage to determine our x-coordinate.

Note that our literal floating-point values and our time intervals are `doubles`. If we compile this on 32-bit, we want to convert them down to `floats`. If we compile on 64-bit, we want to leave them alone. We can accomplish this by casting them to `CGFloat`—since its size changes with the target platform.

Now, let's calculate the y-values.

```
- (CGFloat)calculateYForWeight:(CGFloat)weight
{
    NSAssert([self.weightEntries count] > 0,
             @"You must have more than one entry "
             @"before you call this method");

    if (self.minimumWeight == self.maximumWeight)
    {
        return (self.bottomY + self.topY) / (CGFloat)2.0;
    }

    CGFloat height = self.bottomY - self.topY;
    CGFloat percent = (CGFloat)1.0 - (weight - self.minimumWeight) /
    (self.maximumWeight - self.minimumWeight);

    return height * percent + self.topY;
}
```

The Y value calculations are similar. On the one hand, the code is a little simpler, since we can do mathematical operations on the weight values directly. Notice, however, that the math to calculate our percentage is much more complicated. There are two reasons for this. First, in our dates, the earliest date will always have a value of 0.0. With our weights, this is not true. We need to adjust for our minimum weight. Second, remember that our y-coordinates increase as you move down the screen. This means our maximum weight must have the smallest y-coordinate, while our minimum weight has the largest. We do this by inverting our percentage, subtracting the value we calculate from 1.0.

Now, let's add the method to draw our dots.

```
- (void)drawDotForEntry:(WeightEntry *)entry
{
    CGFloat x = [self calculateXForDate:entry.date];
    CGFloat y = [self calculateYForWeight:entry.weightInLbs];
    CGRect boundingBox =
    CGRectMake(x - (self.dotSize / (CGFloat)2.0),
               y - (self.dotSize / (CGFloat)2.0),
               self.dotSize,
               self.dotSize);

    UIBezierPath *dot =
    [UIBezierPath bezierPathWithOvalInRect:boundingBox];

    [self.tintColor setFill];
    [dot fill];
}
```

Here we get the x- and y-coordinates for our entry and calculate a bounding box for our dot. Our bounding box's height and width are equal to our `dotSize` property, and it is centered on our x- and y-coordinates. We then call `bezierPathWithOvalInRect:` to calculate a circular Bezier path that will fit inside this bounding box. Then we draw the dot, using our view's tint color as the fill color.

Now, we just need to call these methods. Replace the commented out `drawRect:` with the version shown here:

```
- (void)drawRect:(CGRect)rect
{
    [self calculateGraphSize];
    [self drawGraph];
    [self clearGraphSize];
}
```

FIGURE 4.5 A graph with four entries

Build and run the application. When it has zero entries, the graph view is blank. If you add one entry, it will appear as a dot, roughly in the center of the screen. Add two or more entries, and it will draw the graph's line (**Figure 4.5**). Rotate it, and the graph will be redrawn in the new orientation.

Now we just need to add our guidelines. Let's start by making a private method to draw a guideline.

```
- (void)drawGuidelineAtY:(CGFloat)y withLabelText:(NSString *)text
{
    UIFont *font =
    [UIFont preferredFontForTextStyle:UIFontTextStyleCaption1];

    NSDictionary *textAttributes =
    @{NSFontAttributeName:font,
      NSForegroundColorAttributeName:self.guideLineColor};
```

```
    CGSize textSize = [text sizeWithAttributes:textAttributes];
    CGRect textRect = CGRectMake(self.minX,
                                 y - textSize.height - 1,
                                 textSize.width,
                                 textSize.height);

    textRect = CGRectIntegral(textRect);

    UIBezierPath *textbox = [UIBezierPath bezierPathWithRect:textRect];
    [self.superview.backgroundColor setFill];
    [textbox fill];

    [text drawInRect:textRect withAttributes:textAttributes];

    CGFloat pattern[] = {5, 2};

    UIBezierPath *line = [UIBezierPath bezierPath];
    line.lineWidth = self.guideLineWidth;
    [line setLineDash:pattern count:2 phase:0];

    [line moveToPoint:CGPointMake(CGRectGetMinX(self.bounds), y)];
    [line addLineToPoint:CGPointMake(CGRectGetMaxX(self.bounds), y)];

    [self.guideLineColor setStroke];;
    [line stroke];
}
```

We start by getting the Dynamic Type font for the Caption 1 style. Then we create a dictionary of text attributes. These attributes are defined in NSAttributedString UIKit Addons Reference. They can be used for formatting parts of an attributed string; however, we will be using them to format our entire string. We're setting the font to our Dynamic Type font and setting the text color to the guideline color.

Once that's done, we use these attributes to calculate the size needed to draw our text. Then we create a rectangle big enough to hold the text and whose bottom edge is 1 point above the provided y-coordinate. Notice that we use `CGRectIntegral()` on our rectangle. This will return the smallest integral-valued rectangle that contains the source rectangle.

The `sizeWithAttributes:` method typically returns sizes that include fractions of a pixel. However, we want to make sure our rectangle is properly aligned with the underlying pixels. Unlike the custom line drawing, when working with text, we should not offset our rectangle—the text-rendering engine will handle that for us.

We first use the `textRect` to draw a box using our `BackgroundView`'s background color. This will create an apparently blank space for our label, making it easier to read. Then we draw our text into the same rectangle.

Next, we define our line. Again, we start with an empty path. We set the line's width, and we set the dash pattern. Here, we're using a C array. This is one of the few places where you'll see a C array in Objective-C. Our dashed line pattern will draw a segment 5 points long, followed by a 2-point gap. It then repeats this pattern.

We draw this line from the left edge of our view to the right edge of our view along the provided y-coordinate. This means our line will lay just below the bottom edge of our text's bounding box. Finally, we set the line color and draw the line.

Now we need to create our guidelines. We will have three: one for the max weight, one for the minimum weight, and one for the average weight. Add the following method to our implementation file:

```
- (void) drawLabelsAndGuides
{
    if ([self.weightEntries count] == 0) return;

    WeightUnit unit = getDefaultUnits();

    if (self.minimumWeight == self.maximumWeight)
    {
        float weight = [self.weightEntries[0] weightInLbs];

        NSString * weightLabel =
        [NSString stringWithFormat:@"Weight: %@",
         [WeightEntry stringForWeightInLbs:weight inUnit:unit]];

        [self drawGuidelineAtY:[self calculateYForWeight:weight]
                 withLabelText:weightLabel];
    }
    else
    {
        NSString *minLabel =
        [NSString stringWithFormat:@"Min: %@",
         [WeightEntry stringForWeightInLbs:self.minimumWeight
                                    inUnit:unit]];

        NSString *maxLabel =
```

```
        [NSString stringWithFormat:@"Max: %@",
         [WeightEntry stringForWeightInLbs:self.maximumWeight
                                    inUnit:unit]];

        NSString *averageLabel =
        [NSString stringWithFormat:@"Average: %@",
         [WeightEntry stringForWeightInLbs:self.averageWeight
                                    inUnit:unit]];

        [self drawGuidelineAtY:self.topY
                 withLabelText:maxLabel];

        [self drawGuidelineAtY:self.bottomY
                 withLabelText:minLabel];

        [self drawGuidelineAtY:
         [self calculateYForWeight:self.averageWeight]
                 withLabelText:averageLabel];
    }
}
```

In this method, we simply check to see whether we don't have any entries. If there are no entries, then we don't need to draw any guidelines, so we just return. Next, we check to see whether all of our entries have the same weight (either because we have only one entry or because the user entered the same weight for all entries). If we have only one weight value, we create a single weight label and draw our line at the calculated y-coordinate (which should be the vertical center of the graph). Our line should match up with the dot or line that we drew in our drawGraph method.

If we have more than one weight value, we create a minimum, maximum, and average strings and then draw lines corresponding to the appropriate values.

Now, add this method to drawRect: as shown here:

```
- (void)drawRect:(CGRect)rect
{
    [self calculateGraphSize];
    [self drawGraph];
    [self drawLabelsAndGuides];
    [self clearGraphSize];
}
```

FIGURE 4.6 Our completed graph view

Run the application. With no entries, the graph view is still blank. Add a single entry. This will appear in roughly the middle of the screen, with a guideline running right through our dot. Add multiple lines. You should see the Max, Average, and Min guidelines (**Figure 4.6**).

Our graph is almost done. We just need to update the fonts when the user changes them. We will use the same basic procedure we used in the previous chapter for our history and entry detail views.

Open `GraphViewController.m` and import `UIViewController+HBTUpdates.h`. Now we need to override our `HBT_updateFonts` method as shown here:

```
- (void)HBT_updateFonts
{
    [super HBT_updateFonts];
    [self.graphView setNeedsDisplay];
}
```

If the user changes the font size, this method will mark our graph view as needing to be redrawn. The next time through the run loop, the system will redraw the graph, picking up the new fonts in the process.

Run the app. It will now update correctly when the user changes the font size.

Now, as you can see, there's a lot of room for improvement in this graph. Ideally, however, you've learned a lot along the way. We've covered drawing lines (the grid, graph, and guide lines), shapes (the dots), and text. There are similar methods for drawing images as well—though, it is often easier to just insert a `UIImageView` into your application. The `UIImageView`'s drawing code is highly optimized, and you're unlikely to match its performance.

You should be able to use these, and the other UIKit drawing methods, to create your own custom views. However, it doesn't end here. There are a couple of other graphic techniques worth considering.

OTHER GRAPHICS TECHNIQUES

Under the hood, all of the UIKit drawing methods are actually calling Core Graphics methods. Core Graphics is a low-level C-based library for high-performance drawing. While, most of the time, we can do everything we want at the UIKit level—if you need special features (e.g., gradients or shadows), you will need to switch to Core Graphics.

All core graphics functions take a `CGContextRef` as their first argument. The context is an opaque type that represents the drawing environment. Our UIKit methods also draw into a context—however, the context is hidden from us by default. We can access it, if we need, by calling `UIGraphicsGetCurrentContext()`. This lets us freely mix UIKit and Core Graphics drawing code.

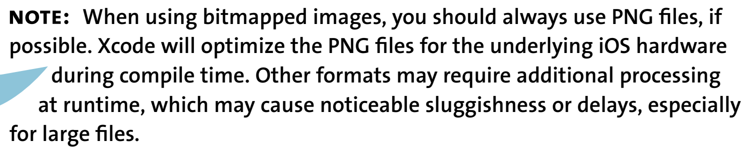
In the `drawRect:` method, our code is drawn into a context that is then displayed on the screen, we can also create graphics contexts to produce JPEG or PNG images or PDFs. We may even use graphics contexts when printing.

However, custom drawing isn't the only way to produce a custom user interface. We can also drop bitmap images into our layout. There are two main techniques for adding bitmaps. If it's simply artwork, we can just place a `UIImageView` in our view hierarchy and set its `image` property. If, however, our image needs to respond to touches, we can place a custom `UIButton` and then call its `backgroundImageForState:` method (or set the background image in Interface Builder).

Bitmaps are often easier than custom drawing. You simply ask your graphic designer to create them for you. You place them in the assets catalog, and you drop them into your interface.

However, custom drawn interfaces have a number of advantages. The biggest is their size. Custom drawn views require considerably less memory than bitmaps. The custom drawn views can also be redrawn to any arbitrary size. If we want to change the size of a bitmapped image, we usually need to get our graphic designer to make a new one. Finally, custom drawn views can be dynamically generated at runtime. Bitmaps can only really present static, unchanging pieces of art.

Often, however, we can get many of the advantages of bitmaps while sidestepping their weaknesses by using resizeable images or image patterns. For more information, check out `-[UIImage resizableImageWithCapInsets:]`, `+[UIColor colorWithPatternImage:}`, and the Asset Catalog Help.



NOTE: When using bitmapped images, you should always use PNG files, if possible. Xcode will optimize the PNG files for the underlying iOS hardware during compile time. Other formats may require additional processing at runtime, which may cause noticeable sluggishness or delays, especially for large files.

UPDATING OUR DRAWING WHEN THE DATA CHANGES

There's one last modification to our graph view. We need to make sure our graph is updated whenever our data changes. This isn't so important now, but it will become extremely important once we implement iCloud syncing.

To do this, we need to listen for our model's notifications. However, unlike our history view, we don't need to track each change individually. Instead, we will update our entire graph once all the changes are complete. This means we need to track only a single notification.

Open `GraphViewController.m`. In the class extension, add the following property:

```
@property (strong, nonatomic) id modelChangesCompleteObserver;
```

Now, add a dealloc method just below `initWithNibName:bundle:`.

```
- (void)dealloc
{
    NSNotificationCenter *center = [NSNotificationCenter defaultCenter];

    if (self.modelChangesCompleteObserver)
    {
        [center removeObserver:self.modelChangesCompleteObserver];
    }
}
```

Next, let's set up our observer. Since there's only one and the code is so simple, let's just put this in `viewDidLoad:`.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.tabBarItem.selectedImage =
    [UIImage imageNamed:@"graph_selected"];
```

```
    NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
    NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
    __weak GraphViewController *_self = self;

    self.modelChangesCompleteObserver =
    [center
     addObserverForName:WeightHistoryDocumentChangesCompleteNotification
     object:self.weightHistoryDocument
     queue:mainQueue
     usingBlock:^(NSNotification *note) {
         _self.graphView.weightEntries =
         [[[_self.weightHistoryDocument allEntries]
           reverseObjectEnumerator] allObjects];
     }];
}
```

This just resets our graph view's weight entry array whenever our document changes. Unfortunately, there's no way to test this code yet. We will get a chance to test it later this chapter, however.

CUSTOM TRANSITIONS

iOS 7 provides classes and methods to support adding custom transitions to our scenes. Specifically, we can customize the transitions when presenting modal views, when switching between tabs, when pushing or popping views onto a navigation controller, and during layout-to-layout transitions between collection view controllers.

In this chapter, we will look at custom transitions when presenting view controllers and when switching between tabs.

For modal segues, the transitions give us a lot of control beyond just modifying the transition's animation. We can also define the final shape and size of our presented view. Since the presented view no longer needs to cover the entire screen, our presenting scene remains in the view hierarchy. We may even be able to interact with any exposed portions of that original scene.

This is an important change, especially for the iPhone. In previous versions of the iOS SDK, presented views had to fill the entire screen on the iPhone (we had a few additional options on the iPad, but they were still relatively limited).

The ability to modify the tab bar animation is also a nice improvement. Before iOS 7, tab bar controllers could not use any animation at all. Touching a tab bar would instantaneously change the content.

However, we're not just going to add an animation sequence between our tabs; we will also create an interactive transition, letting the users drive the transition using a pan gesture. As the user slides her finger across the screen, we will match the motion, sliding from one tab to the next.

Before we can do any of this, however, we need to understand how Core Animation works. Core Animation is the underlying technology behind all of our transitions.

CORE ANIMATION

I won't lie to you: Core Animation is a rich, complex framework. Entire books have been written on this topic. There are lots of little knobs to tweak. However, UIKit has implemented a number of wrapper methods around the basic Core Animation functionality. This lets us easily animate our views, without dropping into the full complexity of the Core Animation framework.

The bad news is UIKit actually exposes two different animation APIs. The old API consists of calling a number of `UIView` methods. However, these methods must be called in the correct order or you will get unexpected behaviors. Furthermore, coordinating between different animations proved difficult.

While these methods have not been officially deprecated yet, their use is no longer recommended for any applications targeting iOS 4.0 or beyond. Since our minimum deployment target is iOS 4.3, there is no good reason to use these methods anymore.

Instead, we want to use `UIView`'s block-based animation API. UIKit provides a number of class methods in the block-based animation API. Some of these let us animate our views—moving them, resizing them, and fading them in and out. Others let us transition between views. They will remove a view from the view hierarchy, replacing it with a different view.

All `UIViews` have a number of animatable properties. These include `frame`, `bounds`, `center`, `transform`, `alpha`, `backgroundColor`, and `contentStretch`. To animate our view, we call one of the animation methods. Inside the animation block, we change one or more of these properties. Core Animation will then calculate the interpolated values for that property for each frame over the block's duration—and it will smoothly animate the results.

If I want to move the view, I just change the `frame` or the `center`. If I want to scale or rotate the view, I change the `transform`. If I want it to fade in or fade out, I change the `alpha`. Everything else is just bells and whistles.

`UIView`'s animation methods are listed here:

- `animateWithDuration:animations:` This is the simplest animation method. We provide a duration and the animation block. It animates our changes over the given duration.
- `animateWithDuration:animations:completion:` This adds a completion block. This block runs after our animation has ended. We can use this to clean up resources, set final positions, trigger actions, or even chain one animation to the next.

- `animateWithDuration:delay:options:animations:completion:` This is the real workhorse. It provides the most options for simple animations. We can set the duration, a delay before the animation begins, a bitmask of animation options, our animation block, and a completion block.
- `animateWithDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:` This is a newer animation method. It's used to create simple, spring-like effects. If you want to give the animation a slight bounce before it settles into the final state, this is your method. You can control the spring's behavior by modifying the dampening and initial velocity arguments. We will look at other techniques for creating realistic, physics-based animations when we talk about `UIDynamics` in Chapter 8.
- `animateKeyframesWithDuration:delay:options:animations:completion:` We can use this method to run keyframe animations. These are more complex animations tied to specific points in time. To set up keyframe animations, you must call `addKeyframeWithRelativeStartTime:relativeDuration:animations:` inside this method's animation block.
- `addKeyframeWithRelativeStartTime:relativeDuration:animations:` Call this method inside `animateKeyframesWithDuration:delay:options:animations:completion:`'s animation block to set up keyframe animations.
- `performSystemAnimation:onViews:options:animations:completion:` This method lets us perform a predefined system animation on an array of views. We can also specify other animations that will run in parallel. At this point, this lets us only delete views. The views are removed from the view hierarchy when the animation block is complete.
- `transitionWithView:duration:options:animations:completion:` We can use this method to add, remove, show, or hide subviews of the specified container view. The change between states will be animated using the transition style specified in the options. This gives us more flexibility than `transitionFromView:toView:duration:options:completion:`, but it requires more work to set up properly.
- `transitionFromView:toView:duration:options:completion:` This method replaces the "from view" with the "to view" using the transition animation specified in the options. By default the "from view" is removed from the view hierarchy, and the "to view" is added to the view hierarchy when the animation is complete.
- `performWithoutAnimation:` You can call methods that include animation blocks inside this method's block. The changes in those animation blocks will take effect immediately with no animation.

AUTO LAYOUT AND CORE ANIMATION

The easiest way to move views around using animation is to modify the view's `frame` or `center` inside an animation block. This works fine as long as our view only has implicit auto layout constraints. In other words, we added a view to our view hierarchy, either programmatically or using Interface Builder, but we don't provide any constraints for that view.

As long as a view has implicit constraints, we can continue to programmatically alter its position and size using the `frame` and `center` properties. However, once we start adding explicit constraints, things get more difficult. We can no longer think in terms of the `frame` or the `center`. If we try, we may get the view to move, but Auto Layout and Core Animation code will eventually fight for control over the view's position. This can result in unexpected results. The animation may run to completion, only to have the view snap back to its original position. The animation may seem to work properly, only to revert after our device rotates or after some other call updates our constraints. Sometimes it can even cause views to flicker back to an old position briefly at the end of the animation. It all depends on the exact nature of the animation and the timing of the updates.

So, if we want to animate changes in size or position of a view with explicit constraints, we have two options.

- We can modify one of the existing constraints inside the animation block. This is the easiest option, but it is also more limited. We can change only the constraint's `constant` property. Nothing else.
- We can remove the old constraint, add a new constraint, and then call `layoutIfNeeded` inside the animation block. This lets us make any arbitrary changes to our constraints but requires more work.

For the purpose of the Health Beat project, we will perform animation on views only with implicit constraints. These are easier to write and easier to understand. I recommend shying away from animating Auto Layout constraints until you have gotten very comfortable with both Core Animation and Auto Layout.

CUSTOMIZING VIEW PRESENTATION

When creating our own transitions, we have two options: custom transitions and interactive transitions. Custom transitions work like most transitions we've seen. We trigger the transition, and then it automatically runs to completion. Most transitions occur rather rapidly, usually in 0.25 seconds.

On the other hand, the user drives interactive transitions. Before iOS 7, we had only one example of an interactive transition—the `UIPageViewController`. This is the controller behind iBook's page flip animation. As you drag your finger across the screen, the page seems to curl, following your finger. You can move the finger forward or backward. You can stop, speed up, or slow down. The animation follows along. Lift your finger and the page either flips to its original position or continues to flip to the new page.

We will start with a custom transition since it is simpler. The actual transition isn't too difficult—though it does have a number of moving parts to keep track of. However, it radically changes the behavior of our application, which will require some additional work to support.

CUSTOMIZING OUR ADD ENTRY VIEW'S TRANSITION

Our application presents our Add Entry View controller as a modal view. It slides in from the bottom and then slides back out again when we are done. We want to change this. Instead of covering the entire screen, we'd like to cover just a small rectangle in the center. We'd also like to fade in and out, instead of sliding.

Doing this requires three steps.

1. We set the presented controller's `modalPresentationStyle` to `UIModalPresentationCustom`.
2. We provide a transitioning delegate.
3. The transitioning delegate creates an object, which implements the actual animation sequences for us. It will actually provide two separate animation sequences: one to present our view, one to dismiss it.

Open `HistoryTableViewController.m` and add the following code to the bottom of its `prepareForSegue:sender:` method. This should go after the `if` block but before the closing curly bracket.

```
if ([segue.identifier isEqualToString:@"Add Entry Segue"])
{
    UIViewController *destinationController =
    segue.destinationViewController;

    destinationController.modalPresentationStyle =
    UIModalPresentationCustom;

    destinationController.transitioningDelegate = self;
}
```

Next, we need to have our history table view controller adopt the `UIViewController TransitioningDelegate` protocol. Scroll up to the top of the file, and modify the class extension's declaration as shown here:

```
@interface HistoryTableViewController ()
<UIViewControllerTransitioningDelegate>
```

This should get rid of our errors, but if you run the app and bring up the add entry view, you'll see that nothing has changed. Actually, the rotations no longer work properly—so things are worse than before.

We need to implement two of `UIViewControllerTransitioningDelegate`'s methods: `animationControllerForPresentedController:presentingController:sourceController:` and `animationControllerForDismissedController:`. However, before we do this, we need to create our animation controller class.

ANIMATOR OBJECT

We need to create an object, which will manage our transition's animation. In the Controller's group, create a new Objective-C class named `AddEntryAnimator`. It should be a subclass of `NSObject`. Open `AddEntryAnimator.h`, and modify it as shown here:

```
@interface AddEntryAnimator : NSObject <UIViewControllerAnimatedTransitioning>
@property (assign, nonatomic, getter = isPresenting) BOOL presenting;
@end
```

We're simply adopting the `UIViewControllerAnimatedTransitioning` protocol and declaring a `presenting` property.

Our `AddEntryAnimator` will be responsible for actually animating the appearance and disappearance of our Add Entry view. To do this, we must implement two methods: `animateTransition:` and `transitionDuration:`.

Switch to `AddEntryAnimator.m`. At the top of the file, let's import `AddEntryViewController.h`. Next, let's define a constant value before the `@implementation` block.

```
static const NSTimeInterval AnimationDuration = 0.25;
```

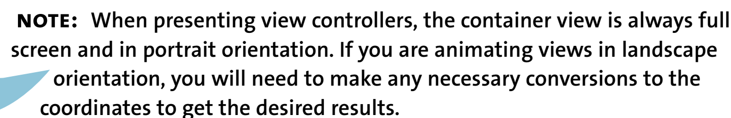
This is the duration we will use for our animations. We can now implement the `transitionDuration:` method as shown here:

```
- (NSTimeInterval)transitionDuration:
(id<UIViewControllerContextTransitioning>)transitionContext
{
    return AnimationDuration;
}
```

This method just returns our constant.

Now, let's look at `animateTransition:` method. This method takes a single argument, our transition context. This is an opaque type, but it adopts the `UIViewControllerContextTransitioning` protocol. As we will see, the context provides a wide range of important information about our transition. This includes giving us access to our view controllers, the beginning and ending frames for their views (if defined by the transition), and our container view.

Our views and view controller hierarchies must be in a consistent state before our transition begins. They must also return to a consistent state after our transition ends. However, during the transition, they may pass through a temporary inconsistent state. To help manage this, we use a container view. The system will create the container and add it to our view hierarchy. It then acts as our superview during our animation sequence.



NOTE: When presenting view controllers, the container view is always full screen and in portrait orientation. If you are animating views in landscape orientation, you will need to make any necessary conversions to the coordinates to get the desired results.

As part of animateTransition:, we need to perform the following steps:

1. Make sure both views have been added to the container view.
2. Animate the transition from one view to the next.
3. Make sure each view is in its final position, if defined.
4. Call completeTransition: to end the transition and put things back into a consistent state.

Since we are using the same animator for both presenting and dismissing our views, our animateTransition: method must handle both cases. To simplify things, we will do some preprocessing in animateTransition:, but then call separate private methods to handle the actual animations.

Add the animateTransition: method, as shown here:

```
-(void)animateTransition:
(id<UIViewControllerContextTransitioning>)transitionContext
{
    id fromViewController =
    [transitionContext viewControllerForKey:
     UITransitionContextFromViewControllerKey];

    id toViewController =
    [transitionContext viewControllerForKey:
     UITransitionContextToViewControllerKey];

    UIView *containerView = [transitionContext containerView];

    if (self.presenting)
    {
        [self
         presentAddEntryViewController:toViewController
         overParentViewController:fromViewController
         usingContainerView:containerView
         transitionContext:transitionContext];
    }
    else
    {
        [self
         dismissAddEntryViewController:fromViewController
         fromParentViewController:toViewController
         usingContainerView:containerView
         transitionContext:transitionContext];
    }
}
```

We start by requesting the "from" and "to" view controllers from our transition context. Then we ask for our container view. We check to see whether we're presenting or dismissing our Add Entry view, and then we call the corresponding helper method.

When presenting view controllers, the from- and to-controllers can be a common source of confusion and errors. When our Add Entry View is appearing, the from-controller will be our `RootTabBarController`. This is the presenting view controller as defined by our current presentation context. The to-controller is our `AddEntryViewController`. However, when we're dismissing the Add Entry View, these roles are reversed.

This means, sometimes our `AddEntryViewController` is the to-controller. Sometimes it's the from-controller. By using helper functions, we can pass the `toViewController` and `fromViewController` in as more clearly named arguments. This lets us work with the `parentController` and `addEntryController`—instead of trying to remember which is "from" and which is "to" in each particular case.

Next, add the `presentAddEntryViewController:overParentViewController:usingContainerView:transitionContext:` method as shown here:

```
#pragma mark - private

- (void) presentAddEntryViewController:
(AddEntryViewController *)addEntryController
overParentViewController:(UIViewController *)parentController
usingContainerView:(UIView *)containerView
transitionContext:
(id<UIViewControllerContextTransitioning>)transitionContext
{

    [containerView addSubview:parentController.view];
    [containerView addSubview:addEntryController.view];

    UIView *addEntryView = addEntryController.view;
    UIView *parentView = parentController.view;
    CGPoint center = parentView.center;

    UIInterfaceOrientation orientation =
    parentController.interfaceOrientation;
    if (UIInterfaceOrientationIsPortrait(orientation))    {
        addEntryView.frame = CGRectMake(0.0, 0.0, 280.0, 170.0);
    }
    else
    {
```

```
        addEntryView.frame = CGRectMake(0.0, 0.0, 170.0, 280.0);
    }

    addEntryView.center = center;
    addEntryView.alpha = 0.0;

    [UIView
     animateWithDuration:AnimationDuration
     animations:^{
         addEntryView.alpha = 1.0;
     }
     completion:^(BOOL finished) {
         [transitionContext completeTransition:YES];
     }];
}
```

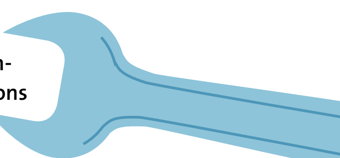
We start by adding both views to the container view. We want to make sure the parent view is added first, so our Add Entry view appears above it when drawn. According to the documentation, the system usually adds the from-controller to the container for us, but we typically have to add the to-controller ourselves. Personally, I find it easiest to just add both views. This won't hurt anything, and it guarantees that both views are properly added and that they will be added in the correct order.

Next, we need to make sure our views end up in the proper position. Since we are presenting a view controller (instead of using the navigation controller or tab bar), our to-controller doesn't have a preset destination. We get to define its final size and location. However, our parent view controller must not move. We could verify this by calling our transition context's `initialFrameForViewController:` and `finalFrameForViewController:` methods for both view controllers. If it has a required position, it will return the frame for that location. If not, it returns `CGRectZero` instead—freeing us to do whatever we want.

We're going to make our Add Entry view 280 points wide and 170 points tall. We are also going to center it in its superview. However, we need to make sure all of these coordinates are in the container view's coordinate system, which is always in a portrait orientation.

Finally we perform the actual animation. We set our view's `alpha` to `0.0`, making it invisible. Then we use an animation block to fade it in, by setting the `alpha` to `1.0` inside the animation block. When the animation is complete, we call `completeTransition:`. This puts our view and view controller hierarchies into their final state and performs other cleanup tasks.

NOTE: `completeTransition:`**'s argument should be** `YES` **if our transition completed successfully,** `NO` **if the animation was canceled. Most custom transitions cannot be canceled, so we almost always just pass** `YES`**. However, this argument becomes more important when writing interactive transitions.**



As you will see, the dismissal code is similar but much simpler. Add the following method, as shown here:

```
- (void) dismissAddEntryViewController:
(AddEntryViewController *)addEntryController
fromParentViewController:(UIViewController *)parentController
usingContainerView:(UIView *)containerView
transitionContext:
(id<UIViewControllerContextTransitioning>)transitionContext
{
    [containerView addSubview:parentController.view];
    [containerView addSubview:addEntryController.view];

    [UIView
     animateWithDuration:AnimationDuration
     animations:^{
         addEntryController.view.alpha = 0.0;
     } completion:^(BOOL finished) {
         [transitionContext completeTransition:YES];
     }];
}
```

This time, we don't need to worry about calculating anyone's frame. Again, the parent view should not move, while the Add Entry view will be removed from the view hierarchy and deallocated. So, we can safely ignore both of their frames.

We just add both views to the container view and then create our animation block. This time, we just set the Add Entry view's `alpha` to `0.0` inside the animation block and then call `completeTransition:` in the completion block.

IMPLEMENTING THE DELEGATE METHODS

Now, open `HistoryTableViewController.m` and import `AddEntryAnimator.h`. Then add the following methods after the storyboard unwinding methods:

```
#pragma mark - UIViewControllerTransitioningDelegate Methods

- (id<UIViewControllerAnimatedTransitioning>)
animationControllerForPresentedController:(UIViewController *)presented
presentingController:(UIViewController *)presenting
sourceController:(UIViewController *)source
{
    AddEntryAnimator *animator = [[AddEntryAnimator alloc] init];
    animator.presenting = YES;
```

```
    return animator;
}

- (id<UIViewControllerAnimatedTransitioning>)
animationControllerForDismissedController:(UIViewController *)dismissed
{
    AddEntryAnimator *animator = [[AddEntryAnimator alloc] init];
    animator.presenting = NO;
    return animator;
}
```

The first method is called when performing the modal segue to our `AddEntryViewController`. The second is called when our segue unwinds and our `AddEntryViewController` is dismissed. In both cases, we instantiate an animator. We set its `presenting` property, and we return the animator. The animator does all the real work.

Run the application, and display our Add Entry view. As you can see, there are a couple of problems. Our keyboard covers our view. Our weight label is truncated—we really want our text fields to shrink instead. Our view is not easy to see against the background. It does not rotate properly. When the Cancel or Save button is pressed, it is not dismissed. And, we can still use the controls of our background view. Click the Graph tab or the Edit button, and they operate normally.

As you can see, switching from presenting a modal view to presenting a custom pop-up can involve quite a few changes to our code.

GENERAL CLEANUP

Let's fix the easy things first. For whatever reason, when we use the regular modal presentation style, the segue unwinding methods will automatically dismiss our presented view controller. We didn't have to do anything. However, with the custom presentation style, we need to programmatically dismiss our presented controller.

Still in `HistoryTableViewController.m`, navigate to our segue unwinding methods. Add the following line to the bottom of both `addNewEntrySaved:` and `addNewEntryCanceled:`:

```
[self dismissViewControllerAnimated:YES completion:nil];
```

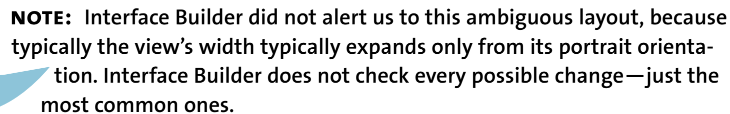
Our Add Entry view will now be properly dismissed whenever the Save or Cancel button is tapped.

Next, open `Main.storyboard` and zoom in on the Add Entry View Controller scene. Our Auto Layout constraints did not work as we expected. Our labels are supposed to remain the same size, while our text fields grow and shrink. This worked properly when we expanded the view, but didn't work now that we shrunk the view.

There were no unusual warnings in the console, so either we have a bug in our constraint logic or we have managed to create an ambiguous constraint. According to Interface Builder, everything's OK, so let's take a look at the compression resistance and hugging priorities for our labels and text fields.

Our labels have a content hugging of 251 and a compression resistance of 750. Our text fields have a content hugging of 100 and a compression resistance of 750. We reduced their content hugging to make sure they stretched when the view grew wider during rotations; however, we never thought about our views becoming smaller. Both the text field and the label have a content hugging of 750, which means we have an ambiguous layout. The system doesn't know which element to shrink.

Drop the Horizontal Compression Resistance on both text fields to 100. This breaks the tie. Now, when we run the app, our text fields shrink instead of our labels.



NOTE: **Interface Builder did not alert us to this ambiguous layout, because typically the view's width typically expands only from its portrait orientation. Interface Builder does not check every possible change—just the most common ones.**

This, however, introduces another problem. Our text fields are now too small for our date. Open `AddEntryViewController.m` and modify `updateDateText` as shown here:

```
- (void)updateDateText
{
    if (self.date == nil)
    {
        self.dateTextField.text = @"";
    }
    else
    {
        self.dateTextField.text =
        [NSDateFormatter
         localizedStringFromDate:self.date
         dateStyle:NSDateFormatterShortStyle
         timeStyle:NSDateFormatterShortStyle];

        self.dateTextField.textColor =
        [UIColor darkTextColor];
    }
}
```

Now, back in `Main.storyboard`, select the date text field, and change its Min Font Size attribute to **12**. That should both give us more space, and let the font shrink if necessary.

While we're in Interface Builder, let's do something about our view's appearance. Select the View, and then click the Background attribute. In the pop-up menu, select Other.... This brings up the color picker. I want a very light blue—almost, but not quite, white. Select the Color

Sliders tab, and set it to **RGB Sliders**. Set the Red to **240**, Green to **245**, and Blue to **255**. Then close the color picker. This will help our Add Entry view stand out against the background.

Finally, let's fix the rotation. Switch back to AddEntryViewController.m and add the following method just under HBT_updateFonts:

```
- (void)willAnimateRotationToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
duration:(NSTimeInterval)duration
{
    [self centerViewInParent];
}
```

This method is called from within our rotation's animation block, so any animatable properties that we change here will be animated along with the rotation. We will be updating our view's position in a few different methods, so we should extract that code into its own, private method. Add the centerViewInParent method to the bottom of our private methods, just before the @end directive.

```
- (void)centerViewInParent
{
    UIView *parentView = self.view.superview;
    CGPoint center = parentView.center;
    center = [parentView convertPoint:center
                             fromView:parentView.superview];
    self.view.center = center;
}
```

Here, we calculate our center's new position. Again, we grab the superview's center and then convert it to the correct coordinate system. Finally, we assign it to our view.

This will cause our view's location to update properly when rotated.

AVOIDING THE KEYBOARD

We need to listen to the keyboard notifications and move out of the way of our incoming keyboard. We also need to adjust our position if the keyboard is being displayed when a new text field is selected or when we rotate. This means we need to know the keyboard's state.

Let's start by listening to some of the keyboard appearance and disappearance notifications. Open AddEntryViewController.m. In the class extension, we need to add two properties for our observers. We also need a property to hold our keyboard's frame and a property to record whether it's currently being displayed.

```
@property (strong, nonatomic) id keyboardWillAppearObserver;
@property (strong, nonatomic) id keyboardWillDisappearObserver;
@property (assign, nonatomic) CGRect keyboardFrame;
@property (assign, nonatomic) BOOL keyboardIsShown;
```

Now, after `initWithNibName:bundle:`, add a `dealloc` method to remove these observers.

```
- (void)dealloc
{
    NSNotificationCenter *center = [NSNotificationCenter defaultCenter];

    if (self.keyboardWillAppearObserver)
    {
        [center removeObserver:self.keyboardWillAppearObserver];
    }

    if (self.keyboardWillDisappearObserver)
    {
        [center removeObserver:self.keyboardWillDisappearObserver];
    }
}
```

Most of the time, we've removed our observers just because it's a good habit to get into. However, since the view controllers in question would last throughout our application's entire life cycle, it was not vital. In this case, however, it is required. Our Add Entry View Controller will be deallocated after it is dismissed. We must make sure it removes all its observers before it disappears.

Now, navigate to `viewDidLoad:` We're going to add the following line to the bottom of the method (just after the call to `setupInputAccessories`):

```
[self setupNotifications];
```

Next, we need to implement this method. Add `setupNotifications` just after our `centerViewInParent` method:

```
- (void)setupNotifications
{
    NSNotificationCenter *center = [NSNotificationCenter defaultCenter];
    NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];
    UIWindow *window =
    [[[UIApplication sharedApplication] delegate] window];

    __weak AddEntryViewController *_self = self;

    self.keyboardWillAppearObserver =
    [center
     addObserverForName:UIKeyboardWillShowNotification
```

```
         object:nil
         queue:mainQueue
         usingBlock:^(NSNotification *note) {
             _self.keyboardIsShown = YES;
             CGRect frame =
             [note.userInfo[UIKeyboardFrameEndUserInfoKey] CGRectValue];
             frame = [window convertRect:frame fromWindow:nil];
             frame = [_self.view.superview convertRect:frame fromView:window];
             _self.keyboardFrame = frame;

             NSTimeInterval duration =
             [note.userInfo[UIKeyboardAnimationDurationUserInfoKey]
              doubleValue];

             [UIView animateWithDuration:duration animations:^{
                 [_self placeFirstResponderAboveKeyboard];
             }];
         }];


        self.keyboardWillDisappearObserver =
        [center
         addObserverForName:UIKeyboardWillHideNotification
         object:nil
         queue:mainQueue
         usingBlock:^(NSNotification *note) {
             _self.keyboardIsShown = NO;
             _self.keyboardFrame = CGRectZero;

             NSTimeInterval duration =
             [note.userInfo[UIKeyboardAnimationDurationUserInfoKey]
              doubleValue];

             [UIView animateWithDuration:duration animations:^{
                 [_self centerViewInParent];
             }];
         }];
    }
```

When the keyboard appears, we set keyboardIsShown to YES, and then we grab the keyboard's end frame from the notification's userInfo dictionary. We convert it into our superview's coordinate system. Then we call a private method to update our position relative to the keyboard. We do the update in an animation block and set the animation duration to our keyboard's appearance duration—this helps match our keyboard's animation.

When the keyboard will disappear notification is received, we set keyboardIsShown to NO. We set the keyboard's frame to all zeros, and we center our view in the frame. Again, we use an animation block to sync with our keyboard's disappearance.

We still need to implement placeFirstResponderAboveKeyboard. Add this after our setupNotifications method.

```
- (void)placeFirstResponderAboveKeyboard
{
    UIView *parentView = self.view.superview;
    CGPoint currentCenter = self.view.center;

    UIView *firstResponder = [self.dateTextField isFirstResponder] ?
    self.dateTextField : self.weightTextField;

    CGRect textFrame = [parentView convertRect:firstResponder.frame
                                      fromView:self.view];

    // our superview is always in portrait orientation
    UIInterfaceOrientation orientation = self.interfaceOrientation;
    if (UIInterfaceOrientationIsPortrait(orientation))
    {
        CGFloat textBottom = CGRectGetMaxY(textFrame);
        CGFloat targetLocation =
        CGRectGetMinY(self.keyboardFrame) - (CGFloat)8.0;
        currentCenter.y += targetLocation - textBottom;
    }
    else if (orientation == UIInterfaceOrientationLandscapeLeft)
    {
        CGFloat textBottom = CGRectGetMaxX(textFrame);
        CGFloat targetLocation =
        CGRectGetMinX(self.keyboardFrame) - (CGFloat)8.0;
        currentCenter.x += targetLocation - textBottom;
    }
    else
    {
```

```
        CGFloat textBottom = CGRectGetMinX(textFrame);
        CGFloat targetLocation =
        CGRectGetMaxX(self.keyboardFrame) + (CGFloat)8.0;
        currentCenter.x += targetLocation - textBottom;
    }

    self.view.center = currentCenter;
}
```

We start by grabbing our parent view and our current center. We want to make sure that all coordinates are in our parent view's coordinate system, so we will convert our coordinates when necessary.

Next, we determine who our first responder is. Here, we use C's ternary operator. Basically, it has three parts. It checks to see whether the first part (before the question mark) is true. If it is true, it returns the second part (between the question mark and the colon). If it's not true, it returns the third part (after the colon).

We can then convert our first responder's frame to the parent view's coordinate system and calculate the distance we need to move our view. This gets a bit complicated, as you can see.

Here's the problem. You might think that our Add Entry view's superview would be the presenting controller's view (e.g., the `RootTabBarController`'s view). This is not the case. It's actually kept in the container view from our transition. And, just like inside our transition, the container view is always kept in portrait orientation. This means we have to adjust our coordinates appropriately.

Conceptually, we get the bottom of the selected text view and the top of our keyboard. We calculate the difference between them, adding in an 8-point margin. Then we offset our view's center by that difference. Of course, the details vary depending on our current orientation.

This will shift our view so that the currently selected text field is just above the keyboard. Notice that it also works when we rotate, since our keyboard is removed and re-added during rotations.

Now, our keyboard moves out of the way—most of the time. However, there's a problem when our view first appears. The keyboard appears as well, but the timing doesn't quite work. The simplest solution is to move our call to `becomeFirstResponder` from `viewWillAppear:` to `viewDidAppear:`. This delays the appearance of our keyboard, making sure our view can move out of the way as expected.

Delete the following line from `viewWillAppear:`:

```
[self.weightTextField becomeFirstResponder];
```

And, just after that method, implement `viewDidAppear:` as shown here:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [self.weightTextField becomeFirstResponder];
}
```

One last tweak. If we are currently editing one text field and we select the other text field, our view should shift appropriately. Let's start by adding an action method that the views can trigger when they begin editing.

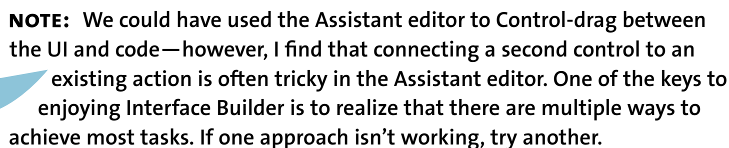
Add this method right after the `unitButtonTapped` method:

```
- (IBAction)textFieldSelected:(id)sender {
    if (self.keyboardIsShown)
    {
        [UIView animateWithDuration:0.25 animations:^{
            [self placeFirstResponderAboveKeyboard];
        }];
    }
}
```

Here, we just check to make sure the keyboard is already showing. If it is, we adjust our view's positioning inside an animation block with a quarter-second duration.

NOTE: If in doubt, use a quarter-second animations. These usually match the system animations nicely.

Now, let's switch to `Main.storyboard`. Control-click the weight text field to bring up its connections HUD. Drag from Editing Did Begin to our Add Entry View Controller, and select textFieldSelected:. Do the same thing for our date text field.



NOTE: We could have used the Assistant editor to Control-drag between the UI and code—however, I find that connecting a second control to an existing action is often tricky in the Assistant editor. One of the keys to enjoying Interface Builder is to realize that there are multiple ways to achieve most tasks. If one approach isn't working, try another.

That's it. Our pop-up view now automatically adjusts its positioning relative to the keyboard. Run the app and test it. It would be nice to have a Next button in our toolbar to toggle between the text fields, but I'll leave that as an exercise for the reader.

ROTATING THE CONTAINER VIEW

The fact that the container view is always in portrait orientation may seem odd—but it makes sense once we dig a bit deeper. Remember, our transition's container view acts as the superview for both our presenting and our presented view controller. And, as we saw, the presenting view controller is our `RootTabBarController`.

Now, before the transition began, the `RootTabBarController`'s superview was our window, and the window is always displayed as full screen and in portrait orientation, so the container view simply matches that.

While this makes sense, it's often harder to work with. Ideally we would like to rotate the container view to the interface's current orientation and just present our view controllers inside it. This is possible, but it may be more work than is worth.

The basic procedure is simple. Grab the transform from the from-controller. Assign that to the container view. Then set the transform for both the from-controller and the to-controller to `CGAffineTransformIdentity`. Update the frames as necessary.

The code should look like the following:

```
// Adjust the coordinate system of all three views
containerView.transform = fromViewController.view.transform;
containerView.frame = window.bounds;

fromViewController.view.transform = CGAffineTransformIdentity;
fromViewController.view.frame = containerView.bounds;

toViewController.view.transform = CGAffineTransformIdentity;
```

This, however, isn't enough. The system expects our views to be in the original orientation. It will also "helpfully" reset some (but not all) of the transforms. To make things work completely correctly, you'll need to set everything up before each animation block and then reset it during the completion handler. That also means we'd have to do a similar shuffle-and-reset in our `placeFirstResponderAboveKeyboard` method.

However, if all of that is done correctly, we wouldn't need to check the orientation and modify our math. Of course, I think the modified math is simpler—but your mileage may vary.

MAKING THE VIEW MODAL AGAIN

Currently, the users can interact with the scenes behind our Add Entry view. This doesn't cause any real problems. You can actually use this to make sure our graph view is updating properly. With the History view showing, bring up the Add Entry view. Then tap the Done button to dismiss the keyboard. With our Add Entry view still showing, select the Graph tab. Now enter a weight. It should appear on the graph.

While this is somewhat interesting, I'd rather make our Add Entry scene behave like a modal view again. I think that would be less surprising and confusing to the users. Fortunately, this is rather easy to do. However, the solution is not obvious or intuitive.

We just need to modify the way our Add Entry view performs hit testing.

When a finger touches the screen, the system uses hit testing to determine exactly which element was touched. For each touch event, it asks the window if the touch was within its bounds. If it was, the window asks all of its children. If the touch was within any of their bounds, they ask all their children. This repeats all the way down the view hierarchy.

The system does this using the `hitTest:withEvent:` method. By default, if the touch event is outside the receiver's `bounds`, it returns `nil`. Otherwise, it calls `hitTest:withEvent:` on all of its subviews. If any of them return a non-`nil` value, it returns the results from the subview at the highest index (the one that would lie on top of all the others). Otherwise, it returns itself.

So, we've already seen that our Add Entry view's superview is our transition's container view. This view fills the entire screen. So, no matter where the user touches on the screen, it will be inside the container's bounds. The container will then call our view's `hitTest:withEvent:` method. We just need to guarantee that our view never returns `nil`. This will allow our view to grab all the touch events whenever it is displayed.

So, let's start by adding a new `UIView` subclass to the Views group. Name this class `AddEntryView`. Then open `AddEntryView.m`. We don't need the `initWithFrame:` or the commented-out `drawRect:` methods. Delete them both. Then add the following method:

```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event
{
    UIView *view = [super hitTest:point withEvent:event];
    if (view == nil) return self;
    return view;
}
```

Here, we call super's `hitTest:withEvent:`. If this returns a value, we simply pass that value along. Otherwise, we return self. This lets our view's subviews continue to respond to touch events but grabs and ignores all other touch events.

Now, open `Main.storyboard`. Zoom in on the Add Entry View Controller scene. Select the scene's view, and in the Identity inspector, change its class to **AddEntryView**.

Run the application. When you bring up the Add Entry view, you can no longer touch any of the controls below our view. However, the keyboard still works, since it is displayed above our view.

LAST THOUGHTS ON PRESENTING CUSTOM VIEWS

The actual custom animation had a number of cogs and gears that needed to be connected correctly, but it really wasn't that hard. We spent most of our time writing custom code to deal with the fact that our Add Entry view was no longer a full-screen, modal view.

In fact, we've written all of the custom code with the basic assumption that our Add Entry View Controller will always be presented using the `UIModalPresentationCustom` presentation style.

If we ever plan on pushing this controller onto a navigation stack, adding it to our tab bar, or even presenting it full screen using the `UIModalPresentationFullScreen` style, we would need to check and see how our view is being displayed and then enable or disable the code as appropriate.

We could use our view controller's `presentingViewController`, `navigationController`, `tabBarController`, `modalPresentationStyle`, and similar properties to determine this. However, going through all the different corner cases gets quite involved.

The good news is if you mess this up, it will be immediately, visibly obvious. This is not one of those bugs that can sneak up and bite you unexpectedly.

INTERACTIVE TRANSITIONS

Now, let's look at interactive transitions.

There's good news, bad news, and good news again. Interactive transitions build upon everything we learned about custom transitions. They are a bit more complex, since we have to sync them with a gesture (or really anything that can programmatically drive our percentages). We also need to determine whether our transition is finished or whether it has been canceled.

Fortunately, we are just adding animation and a bit of additional interactivity to the tab bar controller—we're not making fundamental changes to the way these scenes are presented, like we did with our Add Entry scene. So, once the animation is working, we're done.

When presenting a view controller, we had to set the controller's `transitioningDelegate`. Fortunately, when customizing the transitions for tab bars or navigation bars, the animation methods have already been added to the `UITabBarControllerDelegate` and `UINavigationControllerDelegate` protocols. So, we just need to create an object that adopts `UITabBarControllerDelegate`. Assign it as our tab bar's `delegate` and implement the animation methods

Just like before, our delegate will return an animator object that adopts the `UIViewControllerAnimatedTransitioning` protocol. This will manage the noninteractive portions of our animation; for example, if you select a new tab, the animator object will automatically animate the transition from one scene to the next.

If we want to add interactivity, we also need to return an object that adopts the `UIViewControllerInteractiveTransitioning` protocol. This is responsible for the interactive portion. Interactively driving the animation could get quite complex. Fortunately, UIKit provides a concrete implementation, `UIPercentDrivenInteractiveTransition`, that handles most of the details for us.

The percent-driven interactive transition will actually use our animator object. It can access the frames that the animator object produces and display the correct frame based on the interaction's current state.

While driving the animation interactively, we just change the percentage, and the animation updates appropriately. If we cancel the interaction, it will use the animator to roll back the animation to its beginning state. If we finish the interaction, it will use the animator to complete any remaining animation, bringing the views to their correct, final position.

Strictly speaking, we don't need to subclass `UIPercentDrivenInteractiveTransition`, but I find that it usually simplifies our application's logic. It also lets us combine our `UIPercentDrivenInteractiveTransition` and our `UITabBarControllerDelegate` into a single object.

CREATING THE ANIMATOR OBJECT

So, let's start with our animator object. In the Controllers group, create a new `NSObject` subclass named `TabAnimator`.

Now that we have a couple of animators, let's give them their own group. Select `TabAnimator.h`, `TabAnimator.m`, `AddEntryAnimator.h`, and `AddEntryAnimator.m`. Control-click them and select "New Group from Selection" from the pop-up menu. Name the group **Animators**.

In my mind, these objects belong with the controllers, so I will leave this group inside the Controllers group. However, you could drag it up to the top level or into one of the other groups, if you wanted.

Now, select `TabAnimator.h`. Modify it as shown here:

```
@interface TabAnimator : NSObject <UIViewControllerAnimatedTransitioning>
@property (weak, nonatomic) UITabBarController *tabBarController;
@end
```

We are simply adopting the `UIViewControllerAnimatedTransitioning` protocol and adding a property that will refer back to our tab bar controller.

Switch to `TabAnimator.m`. At the top of the file, before the beginning of our `@implementation` block, we need to declare a constant for our duration. Add the following line of code:

```
static const NSTimeInterval AnimationDuration = 0.25;
```

Now, we can define our methods. As before, we have to implement only two methods. The first is `transitionDuration:`.

```
-(NSTimeInterval)transitionDuration:
(id<UIViewControllerContextTransitioning>)transitionContext
{
    return AnimationDuration;
}
```

Just like our `AddEntryAnimator`, this method returns our constant. After this, add the `animateTransition:` method.

```
-(void)animateTransition:
(id<UIViewControllerContextTransitioning>)transitionContext
{
    UIViewController *fromViewController =
    [transitionContext viewControllerForKey:
     UITransitionContextFromViewControllerKey];

    UIViewController *toViewController =
    [transitionContext viewControllerForKey:
     UITransitionContextToViewControllerKey];

    NSUInteger fromIndex =
    [self.tabBarController.viewControllers
     indexOfObject:fromViewController];

    NSUInteger toIndex =
    [self.tabBarController.viewControllers
     indexOfObject:toViewController];

    BOOL goRight = (fromIndex < toIndex);

    UIView *container = [transitionContext containerView];

    CGRect initialFromFrame =
    [transitionContext initialFrameForViewController:fromViewController];

    CGRect finalToFrame =
    [transitionContext finalFrameForViewController:toViewController];

    CGRect offscreenLeft =
    CGRectOffset(initialFromFrame,
                 - CGRectGetWidth(container.bounds),
                 0.0);


    CGRect offscreenRight =
    CGRectOffset(initialFromFrame,
```

```
                 CGRectGetWidth(container.bounds),
                 0.0);

CGRect initialToFrame;
CGRect finalFromFrame;
if (goRight)
{
    initialToFrame = offscreenRight;
    finalFromFrame = offscreenLeft;
} else
{
    initialToFrame = offscreenLeft;
    finalFromFrame = offscreenRight;
}

fromViewController.view.frame = initialFromFrame;
toViewController.view.frame = initialToFrame;

[container addSubview:fromViewController.view];
[container addSubview:toViewController.view];

UIViewAnimationOptions options = 0;
if ([transitionContext isInteractive])
{
    options = UIViewAnimationOptionCurveLinear;
}

[UIView
 animateWithDuration:AnimationDuration
 delay:0.0
 options:options
 animations:^{
     toViewController.view.frame = finalToFrame;
     fromViewController.view.frame = finalFromFrame;
 } completion:^(BOOL finished) {
     BOOL didComplete = ![transitionContext transitionWasCancelled];

     if (!didComplete)
     {
```

```
            toViewController.view.frame = initialToFrame;
            fromViewController.view.frame = initialFromFrame;
        }

        [transitionContext completeTransition:didComplete];
    }];
}
```

Phew, that's a lot of code! Let's walk through it. Just as before, we start by grabbing our "from" and "to" view controllers. Since we are adding a custom transition to a tab bar, we don't have the same confusion we did with the custom presentation. We are always moving explicitly from one controller to another controller. The from-controller is the controller we started with. The to-controller is the new tab's controller. We may not make it. Our transition may get canceled. But it is our intended destination.

Even if the interactive portion gets canceled, it doesn't change these objects—our interactive transition will simply run this animation in reverse.

Next, we calculate the tab index of our from- and to-controllers, and we determine whether we are sliding to the right or to the left.

We grab our container and request the initial frame for our from- and the final frame for our to-controller. In theory, these should be the same frame, since our final view is replacing our initial view. However, I like to grab them as separate local variables—just in case something changes in future implementations.

Also, notice that, unlike the presentation transition, the `transitionContext` defines a final frame for our to-controller. This means our to-controller's view must be in the specified position when the animation ends. However, the context does not specify a starting position for our to-controller. It also does not specify an ending position for our from-controller. Since we're sliding views in and out, we'd like the to-controller to start just offscreen and the from-controller to end just off the opposite side. Whether they are sliding from the left or right edge depends on the transition's direction.

Returning to our code, we take the from-controller's initial frame and use `CGRectOffset()` to create new frames that are shifted horizontally just past the edge of the screen. These are our `offscreenLeft` and `offscreenRight` rectangles.

Notice that, unlike the presenting transition, the container view will rotate to match our tab bar's orientation. This means we don't need to modify our math based on our orientation.

Next, we use the offscreen rectangles to set our to-controller's initial frame and our from-controller's final frame. We make sure our views are in the correct starting position, and we add them to our container view.

There's one last pretransition check. We see whether our animation is interactive. If it is, we're going to set the animation curve to linear. If not, we will use the default curve.

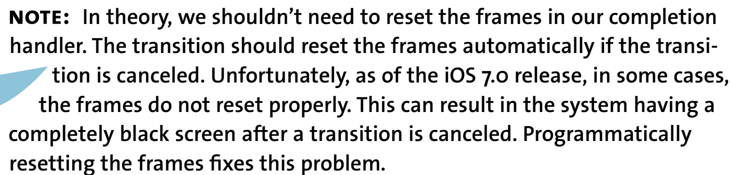
By default, our animations are not displayed at a linear speed. Rather, they ease in and ease out. This means they start at a complete stop. They accelerate up to speed, and then they decelerate back to a stop. This gives the animation a nice appearance that mimics the

way physical objects move. However, the ease-in-out animation curve is not always appropriate, so we can change the animation curve by passing the correct option to the relevant animation methods.

In this case, when we are presenting an interactive animation, we want to make sure our view stays under the user's finger. That means we must use a linear animation. Otherwise, the view and finger will move at different rates.

Finally, we perform our animation. We have to use the longer `animateWithDuration:delay:options:animations:completion:` method, since we may be changing our animation's curve.

In the animation block, we simply update the frames for both our "to" and "from" view controllers. In the completion block, we check to see whether the animation was canceled. If it was, we reset the frames to their initial positions. Finally, we call `completeTransition:`. We pass `YES` if the transition completed and `NO` if it was canceled. This should close out the transition and move the view and view controller hierarchies into their final states.



NOTE: **In theory, we shouldn't need to reset the frames in our completion handler. The transition should reset the frames automatically if the transition is canceled. Unfortunately, as of the iOS 7.0 release, in some cases, the frames do not reset properly. This can result in the system having a completely black screen after a transition is canceled. Programmatically resetting the frames fixes this problem.**

BUILDING TABINTERACTIVETRANSITION

Now let's create our `UIPercentDrivenInteractiveTransition` subclass. This object will also act as our `UITabBarControllerDelegate`. Create a new Objective-C class in the Animators group. Name it `TabInteractiveTransition` and make it a subclass of `UIPercentDrivenInteractiveTransition`.

Next, open `TabInteractiveTransition.h`. We want to adopt the `UITabBarControllerDelegate` property and declare a public method, as shown here:

```
@interface TabInteractiveTransition : UIPercentDrivenInteractiveTransition
<UITabBarControllerDelegate>
- (id)initWithTabBarController:(UITabBarController *)parent;
@end
```

Switch to `TabIntractiveTransition.m`, and import `TabAnimator.h`. And create a class extension with the following properties:

```
@interface TabInteractiveTransition ()
@property (assign, nonatomic, getter = isInteractive) BOOL interactive;
@property (weak, nonatomic) UITabBarController *parent;

@property (assign, nonatomic) NSUInteger oldIndex;
@property (assign, nonatomic) NSUInteger newIndex;
@end
```

The interactive property will track whether we're currently in an interactive transition. The parent property will hold a reference to our tab bar controller. Finally, the newIndex and oldIndex properties will hold the starting and ending tab indexes for the duration of our transition.

Now, add our init... methods. These go just after the @implementation line.

```
- (id)initWithTabBarController:(UITabBarController *)parent
{
    self = [super init];
    if (self) {
        _parent = parent;
        _interactive = NO;
    }
    return self;
}

- (id)init
{
    [self doesNotRecognizeSelector:_cmd];
    return nil;
}
```

initWithTabBarController is our designated initializer. It just sets the parent property and starts with interactive set to NO. Then, as always, we override the designated init method of its superclass. This time, we throw an exception if this method is called. This forces us to always use our designated initializer.

Now, we have two animation methods from UITabBarControllerDelegate that we need to implement. The first returns our animation object.

```
#pragma mark = UITabBarControllerDelegate Methods

-(id<UIViewControllerAnimatedTransitioning>)tabBarController:
(UITabBarController *)tabBarController
animationControllerForTransitionFromViewController:
(UIViewController *)fromVC
toViewController:(UIViewController *)toVC
{
    TabAnimator *animator = [[TabAnimator alloc] init];
    animator.tabBarController = self.parent;
    return animator;
}
```

Here, we just instantiate TabAnimator, set its tabBarController property and return it.

TESTING THE CUSTOM ANIMATION

Before we wire in our interactivity, let's test the custom animation. Switch to RootTabBarController.m, and import TabInteractiveTransition.h. Then add the following property to the class extension:

```
@property (strong, nonatomic) TabInteractiveTransition
*interactiveTransition;
```

This will hold our interactive transition object.

Finally, at the bottom of viewDidLoad, add the following two lines of code:

```
self.interactiveTransition =
[[TabInteractiveTransition alloc] initWithTabBarController:self];

self.delegate = self.interactiveTransition;
```

Here, we instantiate our TabInteractiveTransition object and assign it to our interactiveTransition property. We also assign it to our delegate property. This may seem like unnecessary duplication, but both steps are important.

We need to assign our object to our delegate property, since that's where the real work is done. Unfortunately, delegate properties are almost always weak. This helps prevent retain cycles. Unfortunately, this also means that the delegate property, alone, won't keep our TabInteractiveTransition object in memory. If we want it to last beyond this method, we need to assign it to a strong property as well. The interactiveTransition property simply exists to keep our TabInteractiveTransition object alive.

Run the application and tap the tab bars. Our custom animation will now slide between the History and Graph views.

ADDING INTERACTIVITY

We have a custom animation, but it's not yet interactive. Let's fix that.

Open TabInteractiveTransition.h and declare two additional methods.

```
- (void)handleLeftPan:(UIPanGestureRecognizer *)recognizer;
- (void)handleRightPan:(UIPanGestureRecognizer *)recognizer;
```

We will connect these to our gesture recognizer in a minute.

Switch to TabInteractiveTransition.m. We still have one more delegate method to implement.

```
- (id<UIViewControllerInteractiveTransitioning>)tabBarController:
(UITabBarController *)tabBarController
interactionControllerForAnimationController:
(id<UIViewControllerAnimatedTransitioning>)animationController
{
    if (self.interactive)
    {
```

```
        return self;
    }

    return nil;
}
```

If we want to produce an interactive transition, we must implement both `tabBarController:animationControllerForTransitionFromViewController:toViewController:` and `tabBarController:interactionControllerForAnimationController:`. The interaction controller method will not be called unless the animation controller returns a valid object.

In this case, we simply return `self` if we're in an interactive state. Otherwise, we return `nil`. Or, to put it more simply, if the user starts the animation with a gesture, this method will return `self`, and the animation will proceed interactively. If the user simply taps one of the tabs, it will return `nil`, and the animation will complete automatically.

Now, all we need to do is set up a gesture recognizer to drive the interactive transition.

GESTURE RECOGNIZERS

There are two ways to handle touch events. First, we could use low-level touch handling to monitor each and every touch. Alternatively, we could use higher-level gesture recognizers to recognize and respond to common gestures.

Let's look at the low-level approach first.

Whenever the user's finger touches the screen, this creates a touch event. The system uses hit testing to determine which view was tapped. That view is then sent the touch event. If it doesn't respond, the touch event is passed up the responder chain, giving others a chance to respond.

`UIResponder` defines a number of methods that we can override to respond to touch events. These include `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCanceled:withEvent:`. Both our views and our view controllers inherit from `UIResponder`, so we can implement these methods in either place.

If you want to receive the event and not let it pass on up the responder chain, you must implement all four methods, and you should not call `super`. If you want to just peek at the event and then place it back on the responder chain (possibly letting others respond to it as well), you can just implement the methods you are interested in—but you must call `super` somewhere inside your method.

While this lets us easily receive simple touch information, tracking touches over time to detect more-complex gestures gets very complex. Even something as simple as distinguishing between a single tap, a double tap, and a long press would involve a fair bit of code. Worse yet, if developers create their own, custom algorithms to recognize each gesture, different applications may respond slightly differently to different gestures.

To solve both of these problems, the iOS SDK provides a set of predefined gesture recognizers. Each gesture recognizer has a number of properties and methods and can be used either to request information from the recognizer or to fine-tune the recognizer's behavior.

The recognizers are listed in **Table 4.2.** We can also create our own, custom gesture recognizers—though that is an advanced topic and beyond the scope of this book.

TABLE 4.2 Gesture Recognizers

GESTURE RECOGNIZER	DESCRIPTION
`UITapGestureRecognizer`	Triggers an action once the tap gesture is recognized. You can set both the number of fingers and the number of taps, letting you distinguish between, say, a one-finger tap and a four-finger quadruple-tap.
`UIPinchGestureRecognizer`	Returns continuous pinch data. This is typically used to modify something's size.
`UIRotationGestureRecognizer`	Returns continuous rotation data. This is typically used to rotate objects.
`UISwipeGestureRecognizer`	Triggers an action when the swipe gesture is recognized. You can set the direction and the number of fingers for the gesture.
`UIPanGestureRecognizer`	Returns continuous location data as the finger is moved around the screen. You can set the minimum and maximum number of fingers.
`UIScreenEdgePanGestureRecognizer`	Like the pan gesture, but it recognizes only the gestures that originate from the edge of the screen. This gesture is new with iOS 7. We can set the edges that the gesture will monitor.
`UILongPressGestureRecognizer`	Triggers an action when the user presses and holds a finger on the screen. We can set the duration, the number of fingers, the number of taps before the hold, and the amount of motion allowed. While this is primarily used to trigger actions, it will return continuous data as well.

In many cases, we can set up our gesture recognizers in Interface Builder. We can drag them out and drop them onto a view. Then we can select the gesture recognizer icon to set its attributes. Finally, we can Control-drag from the icon to our code in the Assistant editor to create a method that will be called whenever the recognizer fires.

For discrete gesture recognizers, this method is called once, when the gesture is recognized. For continuous recognizers, it is called when the gesture begins, again for each update, and then when the gesture is canceled or ends. Our method will need to check the recognizer's state and respond accordingly.

We're going to add screen edge pan recognizers to all the view controllers managed by our tab bar. This is a continuous gesture, so we can monitor how far the user has moved their finger across the screen.

We could do this in Interface Builder; however, the screen edge pan recognizer has not been added to the library yet. So, we cannot just drag and drop it in place. More importantly, we

will want to automate adding these gesture recognizers based on the tab's index; that way the gesture recognizers will be added correctly even if we end up adding or removing tabs later.

Open `RootTabBarController.m`. Add the following method to the bottom of the private methods:

```
- (void)addPanGestureRecognizerToViewController:
(UIViewController *)controller
forEdges:(UIRectEdge)edges
{
    NSParameterAssert((edges == UIRectEdgeLeft)||
                      (edges == UIRectEdgeRight));
    SEL selector;

    if (edges == UIRectEdgeLeft)
    {
        selector = @selector(handleRightPan:);
    }
    else
    {
        selector = @selector(handleLeftPan:);
    }

    UIScreenEdgePanGestureRecognizer *panRecognizer =
    [[UIScreenEdgePanGestureRecognizer alloc]
     initWithTarget:self.interactiveTransition
     action:selector];

    panRecognizer.maximumNumberOfTouches = 1;
    panRecognizer.minimumNumberOfTouches = 1;
    panRecognizer.edges = edges;
    [controller.view addGestureRecognizer:panRecognizer];
}
```

Here we start with a parameter assert. This is just like a normal `NSAssert`, but it's specifically designed for checking incoming parameter values. It's also easier to use, since the system automatically generates the error message for us. In this case, we just want to verify that we are setting our recognizer to watch either the left or right edge (not the top, bottom, or multiple edges).

Next, we determine the selector for the method that should be called when the gesture is triggered. If we're panning from the left to the right, it's the `handleRightPan:` method. If we're panning from the right to the left, it's the `handleLeftPan:` method. The *right* or *left* in the handler's name refers to the direction of motion—not the starting position.

We then create our gesture recognizer, passing in our interactive transition as the target, and our selector as the action. When the recognizer is triggered, it will call the specified method on our interactive transition object.

Finally, we set it to respond to one, and only one finger, from the specified edge. Then we add it to our container's view.

Now we need to call this method once for each view controller. Add the following method right before `addPanGestureRecognzierToViewController:forEdges:`:

```
- (void)setupEdgePanGestureRecognizers
{
    NSUInteger count = [self.viewControllers count];
    for (NSUInteger index = 0; index < count; index++)
    {
        UIViewController *controller = self.viewControllers[index];

        if (index == 0) {
            [self addPanGestureRecognizerToViewController:controller
                  forEdges:UIRectEdgeRight];
        }
        else if (index == count - 1)
        {
            [self addPanGestureRecognizerToViewController:controller
                  forEdges:UIRectEdgeLeft];
        }
        else {
            [self addPanGestureRecognizerToViewController:controller
                  forEdges:UIRectEdgeRight];

            [self addPanGestureRecognizerToViewController:controller
                  forEdges:UIRectEdgeLeft];
        }
    }
}
```

This method iterates over all our view controllers. This time, we don't use fast enumeration, since we actually need the index number. If it's the first tab, we add a gesture recognizer on the right edge. If it's the last tab, we add a gesture recognizer on the left edge. If it's somewhere in the middle, we add them to both edges.

Now, add the following line to the bottom of `viewDidLoad`:

```
[self setupEdgePanGestureRecognizers];
```

This sets up our gesture recognizers when our tab bar controller loads.

Now we just need to implement `handleRightPan:` and `handleLeftPan:`. Switch to `TabInteractiveTransition.m` and add the following method to the bottom of the `@implementation` block:

```
- (void)handleLeftPan:(UIPanGestureRecognizer *)recognizer
{
    CGPoint translation = [recognizer translationInView:self.parent.view];

    CGFloat percent =
    -translation.x / CGRectGetWidth(self.parent.view.bounds);

    percent = MAX(percent, 0.0f);
    percent = MIN(percent, 1.0f);

    if (recognizer.state == UIGestureRecognizerStateBegan)
    {
        NSAssert(self.oldIndex == 0, @"We shouldn't already have an "
                 @"old index value");
        NSAssert(self.newIndex == 0, @"We shouldn't already have a "
                 @"new index value");

        self.oldIndex = self.parent.selectedIndex;
        NSAssert(self.oldIndex != NSNotFound,
                 @"Interactive transitions from the "
                 @"More tab are not possible");

        self.newIndex = self.oldIndex + 1;
        NSAssert(self.newIndex < [self.parent.viewControllers count],
                 @"Trying to navigate past the last tab");

    }

    [self handleRecognizer:recognizer
      forTransitionPercent:percent];
}
```

We start by getting the translation from our gesture recognizer. This is the amount our finger has moved from the original point of contact. We then convert this to a percentage. As a safety step, we clamp this value so that it cannot be lower than 0.0 or higher than 1.0. These values will make sense only during change notifications—not when the recognizer began, ended, or was canceled—but we're going to precalculate them anyway, making

the rest of our code easer to read. If it's not a change notification, the percent will simply get ignored.

Next, if this is a begin notification, we save our new and old index. Then we call handleRecognzier:forTransitionPercent:.

handleRightPan: is largely the same. The math for calculating our percentage is a little different, since the gesture is moving in the opposite direction.

```
- (void)handleRightPan:(UIPanGestureRecognizer *)recognizer
{
    CGPoint translation = [recognizer translationInView:self.parent.view];

    CGFloat percent =
    translation.x / CGRectGetWidth(self.parent.view.bounds);

    percent = MAX(percent, 0.0f);
    percent = MIN(percent, 1.0f);

    if (recognizer.state == UIGestureRecognizerStateBegan)
    {
        NSAssert(self.oldIndex == 0, @"We shouldn't already have an "
                 @"old index value");
        NSAssert(self.newIndex == 0, @"We shouldn't already have a "
                 @"new index value");

        self.oldIndex = self.parent.selectedIndex;
        NSAssert(self.oldIndex != NSNotFound,
                 @"Interactive transitions from the "
                 @"More tab are not possible");

        self.newIndex = self.oldIndex - 1;
        NSAssert(self.newIndex >= 0,
                 @"Trying to navigate past the first tab");

    }

    [self handleRecognizer:recognizer
      forTransitionPercent:percent];
}
```

Finally, add the handleRecognzier:forTransitionPercent: method after the other two.

```
- (void)handleRecognizer:(UIPanGestureRecognizer *)recognizer
    forTransitionPercent:(CGFloat)percent
{
    switch (recognizer.state)
    {
        case UIGestureRecognizerStateBegan:
            self.interactive = YES;
            self.parent.selectedIndex = self.newIndex;
            break;

        case UIGestureRecognizerStateChanged:
            [self updateInteractiveTransition:percent];
            break;

        case UIGestureRecognizerStateCancelled:
            self.completionSpeed = 0.5;
            [self cancelInteractiveTransition];
            self.interactive = NO;
            self.newIndex = 0;
            self.oldIndex = 0;
            break;

        case UIGestureRecognizerStateEnded:
            self.completionSpeed = 0.5;
            if (percent > 0.5)
            {
                [self finishInteractiveTransition];
            }
            else
            {
                [self cancelInteractiveTransition];
            }
            self.newIndex = 0;
            self.oldIndex = 0;
            self.interactive = NO;
            break;
```

```
        default:
            NSLog(@"*** Invalid state found %@ ***", @(recognizer.state));
    }
}
```

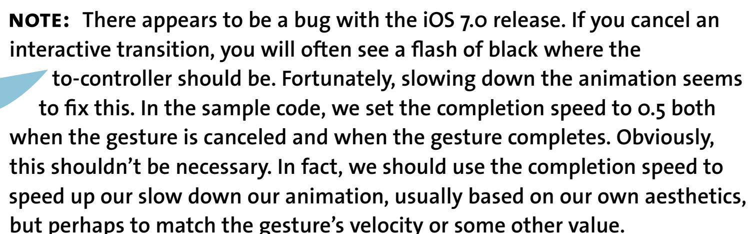
Since the edge pan recognizer is a continuous recognizer, it will send us updates as our finger moves across the screen. This method checks the recognizer's state and responds appropriately.

If the recognizer has just begun, it places us in interactive mode and then changes the tab bar's index to the new index.

Every time we get a change update, it updates our animation percentage by calling `UIPercentDrivenInteractiveTransition`'s `updateInteractiveTransition:` method.

If the recognizer is canceled, we cancel the transition, clear our index properties, and set `interactive` to `NO`.

Finally, if the gesture ends normally (because the user lifts his finger or runs past the edge of the screen), it checks to see how much we have completed. If it is greater than 50 percent, it finishes the transition. The animation will automatically complete. If it is less than 50 percent, it cancels the transition. The animation will automatically roll back to the initial state.



NOTE: There appears to be a bug with the iOS 7.0 release. If you cancel an interactive transition, you will often see a flash of black where the to-controller should be. Fortunately, slowing down the animation seems to fix this. In the sample code, we set the completion speed to 0.5 both when the gesture is canceled and when the gesture completes. Obviously, this shouldn't be necessary. In fact, we should use the completion speed to speed up our slow down our animation, usually based on our own aesthetics, but perhaps to match the gesture's velocity or some other value.

That's it. Run the app. You can now swipe your finger from the edge of the screen to trigger the transition.

WRAPPING UP

In this chapter, we examined a number of techniques that will be helpful when building your own custom views and custom controls. This includes custom drawing, custom transitions, and tracking user interaction through gesture recognizers.

We will take this a step further in Chapter 8, when we look at advanced user interface techniques. In the meantime, Chapter 5 will look at saving our application's data. Chapter 6 will look at syncing that data using iCloud, and Chapter 7 will look at replacing our custom model with Core Data.

OTHER RESOURCES

- **Quartz 2D Programming Guide**

  OS Developer's Library

  This is the complete guide for using Core Graphics. If you want to go beyond the basics, you should definitely read through this programming guide.

- **PaintCode**

  www.paintcodeapp.com

  This is a third-party application for creating custom-drawn interfaces. PaintCode is an illustrator-like drawing tool—but it saves the graphics as Objective-C Code. We can then take that code and paste it into our application. Sometimes the code can be used as-is. Sometimes it needs to be cleaned up or refactored. Regardless, it can save a significant amount of time over coding everything by hand.

  While the application may be a bit pricy, I highly recommend downloading and playing around with the demo. If nothing else, it's a great tool for learning how to generate a wide range of visual effects with Core Graphics.

- **Event Handling Guide for iOS**

  iOS Developer's Library

  This guide provides detailed information about all aspects of iOS event handling. This includes low-level event handling, hit testing, gesture recognizers, and even handling motion or remote control events.

- **Core Animation Programming Guide**

  iOS Developer's Library

  This guide provides a detailed discussion of Core Animation. Note that this goes very deep into a fairly complex topic—however, most of the time we can add a touch of animation to our user interface without getting bogged down in the Core Animation framework itself. Still, if you run into a situation where your animation isn't behaving quite the way you expect, this is a good resource to have in your back pocket.

- **Custom Transitions Using View Controllers**

  2013 WWDC Videos

  Currently, this is the only source of information about custom transitions from Apple. This presentation covers both custom transitions and interactive transitions. While this provides a nice, high-level explanation of the API, it often feels frustratingly thin on details. Hopefully Apple will supplement this presentation with either sample code or a Custom Transition Programming Guide in the near future.

A

accessibility
  resource, 483
  support, 479
accessor methods
  using with `WeightHistoryDocument`, 75
  `WeightHistoryDocument` for Core Data, 404–405
Accounts preferences, 16
ad hoc builds, creating, 481
Add Entry button, outlet for, 299–300
Add Entry scene, updating fonts for, 183–184
Add Entry View. *See also* bouncing Add Entry View
  adding motion effects to, 434–438
  animator object, 229
  bringing up, 168
  custom transition, 225
  laying out, 157–161
`addEntry:` method, adding to `WeightHistoryDocument`, 76
`addSubview:`, 116
aligning
  labels, 39
  views, 456
`animateTransition:`, 227–228
animation methods for `UIView`, 222–223
animation transition controllers
  `AbstractTransitionController`, 449
  aligning views, 456
  `animateTransition:`, 451
  calling `super`, 454
  `cancelAnimation` method, 453
  controller transitioning protocol, 449
  duration methods, 458
  horizontal duration method, 452–453
  `init` methods, 457
  instantiating bounce behavior, 454–455
  lift behavior, 458
  overriding `animateTransition:`, 453–454, 457–458
  `transitionContext`, 451–452
  `UIDynamicAnimator` methods, 452
  `velocity` property, 456
  vertical duration method, 452–453
animations
  length of, 238
  UIKit Dynamics, 438
animator, deallocating, 453
animator object
  Add Entry View, 229
  `completeTransition:`'s argument, 229
  from- and to- controllers, 228
  for custom transition, 226–230
  interactive transitions, 242–246
  positioning views, 229
App Store
  number of apps in, 4
  submitting to, 481–482
App Store Review Guidelines, 484
`AppDelegate`, 31, 57
Apple's Developer site, 481
application artwork. *See also* Health Beat project
  accessibility, 479
  deployment target, 476–477
  file sharing with applications, 479–480
  icons, 472–473
  launch images, 474–475
  localization, 478–479
  required capabilities, 475–476
applications. *See also* best practices
  organizing, 53
  presented controllers, 53
  upgrading, 264–265
apps
  bootstrapping, 30
  distribution resource, 484
  Music, 52
array, filtering in Entry Detail View, 154–155
Asset Catalog Help, 483
assets catalog, contents, 57
Assistant editor, 11, 40–41

attributes, 119
Attributes inspector, 14, 28
Auto Layout. *See also* Health Beat UI layout; layout guides
  aligning views, 132
  alignment rect, 133
  attributes, 119
  constraint API, 122–126
  constraints, 119
  constraints for table view cells, 138–139
  declaring constraints, 120
  default constraints, 122
  defining constraints, 124–126
  detecting ambiguous layouts, 133–134
  detecting conflicts, 134–135
  editing constraints, 129–132
  Editor>Canvas menu, 131–132
  explained, 118
  inequalities, 120
  intrinsic sizes, 121
  introduced, 112
  layout guides, 123
  Leading Edge, 119
  nonscrolling scenes, 124
  Not An Attribute, 119
  overriding methods, 132–133
  priority constants, 121
  pseudo-code examples, 119–120
  selecting constraints, 132
  `shouldBeArchived` property, 133
  Size inspector, 132
  size of controls, 120
  specifying constraints, 122
  timing layouts, 135–136
  Trailing Edge, 119
  using, 119–120
  visual formatting syntax, 126–129
autoresizing masks, 117–118
`awakeFromNib`, calling, 193

B

background color
  setting, 38
  setting to clear, 194
background saves, automating, 303–305
background view
  Core Graphics, 219
  displaying, 200
  extending behind bars, 201
  grid and path, 200
  redrawing, 202
  rotating, 201
  stretched and rotated, 201
  `UIBezierPath`, 200
  use by storyboard, 200
`BackgroundView` class
  creating, 192
  default values, 192
  drawing grid, 195
  `initWithCoder:`, 193
  scale factor, 195
  `setDefaults` method, 193
  setting view to opaque, 194–195
backing up data, 263–264
bars, handling, 201
Behaviors preferences, 16
best practices. *See also* applications
  closing apps, 103
  low memory warnings, 103
  opening apps, 102
bitmapped images, using, 220
bitmaps versus custom drawing, 219
bootstrapping apps, 30
bouncing Add Entry View. *See also* Add Entry View
  `AbstractBehavior` class, 441
  `addNewEntryCanceled:`, 460
  angle for push, 448
  animation transition controllers, 449–459
  `BounceBehavior` class, 444
  `BouncePresentationController`, 460
  child behaviors, 445
  collision behavior, 444–445
  `collision` property, 445
  defining up and down, 446–447
  disabling rotation, 445, 447
  dismiss gesture, 461–468
  dynamic item behavior, 443

gravity behavior, 447
HistoryTableViewController, 459
init, 443
initializer, 442–443
insets for orientation, 445
item behavior, 447
LiftBehavior class, 445–446
read-only properties, 441
read-write properties, 442
setupBehaviors, 443
setupBehaviors method, 445
subclasses, 444
bounds property, 114–115
Breakpoint tab in Navigator, 12
bringSubviewToFront:, 116
building for distribution, 480–482
buttons, resizing, 160

C

Cancel button, adding to bottom of screen, 160
center property, 114–115
center vertically constraint, adding, 39
classes, adding to Health Beat, 63
Clear Button attribute, setting, 47
closing apps, best practices for, 103
code, refactoring, 287
Code Snippet library, 14
collection operators, using, 151
companion file, selecting, 41. *See also* files
concurrency resource, 311
concurrent programming, 290–291
Connections inspector, 14, 25
constraints
Auto Layout, 119–120
center vertically, 39
defining in Auto Layout, 124–126
editing in Interface Builder, 129–132
line graph, 202
selecting in Auto Layout, 132
shouldBeArchived property, 133
specifying in Auto Layout, 122
for text field and navigation bar, 43–44
container controllers, 52
container view, rotating, 239
containers, nesting, 54
content controllers, 52
contentsForType:Error:, overriding, 289
control states, adding to Units button, 174
controllers
collection view, 54
container, 52
content, 52
table view, 53
conversion methods, implementing, 68–69
coordinate systems
considering, 114
OS X versus iOS, 115
core animation resource, 257
Core Data. *See also* Health Beat conversion to Core Data
accessing contexts, 372–381
adding versions, 369
attribute settings, 367
attribute types, 366
Binary Data attribute, 366
Boolean attribute, 366
Cascade delete rule, 368
comparing to databases, 364
configurations, 372
creating document, 423–426
Date attribute, 366
Decimal attribute, 366
delete rules, 368
Deny delete rule, 368
Double attribute, 366
entity connections, 367
entity descriptions, 366, 369
fetch requests, 371–372
fetched properties, 368–369
Float attribute, 366
formatting string placeholders, 379
iCloud and fallback store, 390–391
iCloud limitations, 392–393
iCloud syncing, 390
Indexed attribute setting, 367
inheritance, 369
instantiating entities, 374

Core Data (*continued*)
Integer attributes, 366
integration with iCloud, 389
loading document, 423–426
managed object context, 372–381
managed object model, 365–372
managed objects, 374–375
many-to-many relationships, 368
mapping model, 370
migrating data, 369–371
models, 365
No Action delete rule, 368
NSOrderedSet, 368
Nullify delete rule, 368
to-one relationships, 368
Optional attribute setting, 367
overview, 364–365
persistent stores, 382–385
predicate strings, 379
relationships, 367–368
reliability, 392–393
resources, 429
String attribute, 366
support data, 371–372
Transformable attribute, 366–367
Transient attribute setting, 367
UIManagedDocument, 385–386
Undefined attribute, 366–367
updating view controllers, 426–427
using with SQLite stores, 389
validation settings, 367
Core Data performance
faulting, 387
fetch requests, 386
large binary objects, 388
memory overhead, 387
tools, 388
Core Graphics
data types, 113
explained, 219
functions, 219
count public property
explained, 71–72
getter for, 74
making KVO compliant, 74
using, 94
createCloudURL method, 340–341
createNewFileATURL: method, 289
custom table view cells. *See also* table view controllers
Auto Layout constraints, 138–139
changing font, 137
connecting outlets, 139–140
displaying weights, 145–146
editing content, 137
formatting weight strings, 140–145
revising Weight label, 137
stretching Date label, 139
custom transitions. *See also* view presentation
Add Entry View controller, 225
animation methods for UIView, 222–223
Auto Layout and core animation, 223–224
core animation, 222–224
modal segues, 221
resource, 257
view presentation, 224

D

data
backing up, 263–264
migrating via Core Data, 369–371
passing to scenes, 36
data detector, entering for dates, 170
data objects, 282. *See also* objects
date arithmetic, substituting with NSCalendar, 155
Date label, stretching to fit, 139
dates
alert for changes, 171
creating, 105
data detector, 170
delegate for text field, 171
Done button text, 171
entering in Health Beat, 169–172
generating objects from, 105
text color, 171

dealloc method
  document state changes, 297
  user preferences, 275–276
  WeightHistoryDocument for Core Data, 404
Debug area, 11
Debug navigator, 12, 359
defaults database, accessing, 270
delegate methods
  calling, 37
  implementing, 230–231
delegates, 31
deleting
  controllers from storyboards, 58
  derived data, 287
  entries from Health Beat, 155–157
  log messages, 110
deployment target, 476–477
detail view cell, laying out, 148
dictionary, required capability, 475
dismiss gesture
  bounceBacktoCenter:, 467
  checking state, 463
  endDragWithVelocity:, 465–466
  gesture recognizers, 467–468
  implementing, 461–463
  offset center, 465
  private methods, 464–465
  static variable, 463
  superview, 463
  UIDynamicAnimatorDelegate, 464
  viewDidLoad:, 462
display resolutions, supporting, 195
distribution, building for, 480–482
Document Outline. *See also* UIDocument subclass
  collapsing, 40
  hiding and showing, 22
document start-up sequence
  createCloudURL method, 340–341
  expanding sandbox, 337–339
  loadFile, 348–350
  metadata query, 346
  migrating document, 339–342
  modifying, 333–334
  monitoring iCloud users, 334–337
  processQuery: method, 347–348
  searchForCloudDocument method, 345–346
  searching Ubiquity Container, 346
  updating, 343–350
document state changes. *See also* UIDocument subclass
  automatic background saves, 303–305
  checking state, 298–299, 301
  dealloc method, 297
  delegate approach, 295
  graph view controller, 302
  implementing subscriber method, 302–303
  initWithCoder:, 306
  NSCoding, 305–307
  NSHashTable, 296
  NSKeyedUnarchiver, 305
  outlet for Add Entry button, 299–300
  registering for notifications, 297–298
  removing observer, 297
  responding to, 294–303
  setupDocument method, 299
  subscriber notification method, 297
  subscribers, 295–296
  UIActionSheet, 299
  UIAlertView, 299
  undo action, 307–310
  updateAddButton method, 300–301
Document Storage API. *See also* sandbox
  best practices, 319
  debugging, 359–360
  extended sandbox, 317
  file coordinators, 320–321
  file presenters, 321
  file versioning, 321–322
  flow of data, 317
  flow of metadata, 317
  implementation, 318–319
  Ubiquity Container, 317–318
document storage strategies, 322–323
documentation comments
  adding, 66
  displaying, 67
documentReady method, 287
Done button, 36

done: method
  connecting Return key to, 47
  navigating to, 45
doneEditing action, creating, 166
Double Click Navigation preferences, 16
Downloads preferences, 16
Doxygen-formatted comments, adding, 66
drawing
  background view, 192
  versus bitmaps, 219
  dots for line graph, 213
  line graph, 209
  rectangle for text, 216
  subpixel, 198
  views, 198
drawRect: method, 217, 219
Dynamic Type
  categories, 178
  current state, 178
  default implementation, 179
  font for Caption 1 style, 215
  font size, 180
  function of, 177
  notification, 180–181

E

editing style, checking, 156
Editor, 11
Editor areas, displaying side by side, 41
editors, stacking vertically, 41
entries
  adding to Health Beat, 155–157
  deleting from Health Beat, 155–157
Entry Detail scene
  adding color to, 187
  updating fonts for, 184–185
Entry Detail View
  collection operators, 151
  creating, 154–155
  creating outlets, 149–150
  filtering array, 152, 154–155
  KVC operators, 152
  NSCalendar object, 155
  updating, 150–152
error messages, providing, 292. *See also* warnings
event handling resource, 257, 469
events, responding to, 164
exchangeSubviewAtIndex:withSubviewAtIndex:, 116

F

fetch requests
  performance, 386
  WeightHistoryDocument for Core Data, 409–410
file coordinators
  Document Storage API, 320–321
  resource, 360
File inspector, 13
file presenters, 321
file sharing with applications, 479–480
file system
  manipulating, 267
  resource, 311
File Template library, 14
file versioning. *See also* iCloud
  Document Storage API, 321–322
  in Health Beat, 332–333
files. *See also* companion file
  creating for Health Beat, 63
  opening, 12
  saving, 318
  sharing with applications, 479–480
Find tab in Navigator, 12
First Responder
  becomeFirstResponder, 164
  explained, 23
  resignFirstResponder, 164
flipside view controller, 26
FlipsideViewController.h, 36
font size, determining with Dynamic Type, 180
fonts
  setting, 40
  updating, 277

updating for Entry Detail scene, 184–185
updating in History Table, 181–182
Fonts & Colors preferences, 16
`frame` property, 114–115
frameworks, 5
from- controller, 228

G

General preferences, 15
gesture recognizers. *See also* touch events
calculating percentage, 254
creating, 252
getting translation from, 253–254
parameter assert, 251
`UILongPressGestureRecognizer`, 250
`UIPanGestureRecognizer`, 250
`UIPinchGestureRecognizer`, 250
`UIRotationGestureRecognizer`, 250
`UIScreenEdgePanGestureRecognizer`, 250
`UISwipeGestureRecognizer`, 250
`UITapGestureRecognizer`, 250
graph
drawing, 209
with four entries, 214
graph tab icon, adding to Health Beat, 61
graph view
completing, 218
observer, 220–221
outlet, 203
resetting weight entry, 221
resizing, 202–203
updating when data changes, 220–221
`viewDidLoad:`, 220
graph view controller, adding, 302
`GraphViewController.h`, 91
groups, adding to Health Beat, 58

H

`handleError:userInteractionPermitted:`, 293
hash methods, using with iCloud, 358–359
hash table, creating, 296
Health Beat conversion to Core Data. *See also* Core Data
fetch request, 395
managed object model, 393–395
predicate syntax expressions, 395
`WeightEntry` class, 396–400
`WeightEntry` entity, 395
Health Beat project. *See also* application artwork; projects; storyboard for Health Beat
accessory view, 168
Add Entry View, 157–161
adding classes to, 63
adding entries, 155–157
adding groups, 58
adding images, 60–61
assets catalog, 57
Cancel button, 160, 176
checking editing style, 156
cleaning up, 57–58
creating, 55–57
creating dates, 105
creating files, 63–64
custom preferences, 280
customizing appearance, 186–187
Debug navigator, 359
deleting controllers from storyboard, 58
deleting entries, 155–157
deleting log messages, 110
deleting rows, 157
Dynamic Type, 177–182
editing mode, 156
entering dates, 169–172
entries, 105
Entry Detail View, 147–152, 154–155
file versioning, 332–333
fonts for Add Entry, 183–184
fonts for Entry Detail, 184–185
fonts in History Table, 181–182
generating objects from dates, 105
graph tab icon, 61
implementing `WeightUnit`, 64
`inputAccessoryView`, 164–168

Health Beat (*continued*)
- key-value syncing, 323–332
- launch image, 474–475
- making room for keyboard, 159
- model, 52
- modules, 52
- preference page, 280
- responder chain, 164
- `Root.plist`, 279, 281
- running template, 57
- Save button, 160, 176
- saving, 56
- searching workspace, 110
- segue unwinding, 176–177
- selecting location for, 56
- setting options, 56
- setting warnings, 58–60
- tasks, 55
- testing architecture for, 104–106
- `tintColor:` method, 186
- `// TODO`: comments, 110
- `UIAppearance`, 186
- `UIDocument` subclass, 284–286
- `UINib` object, 168
- Units button, 172–174
- `updateDateText`, 163
- `updateUI` method, 163
- `updateWeightText` method, 162–163
- updating text fields, 161–164
- user preferences, 271–282
- view controllers, 52
- view hierarchies, 52
- Weight Entry scene, 106
- weight management, 55
- `WeightEntry`, 65–69
- `WeightHistoryDocument`, 70–79

Health Beat project model
- `EntryDetailViewController.h`, 92
- `GraphViewController.h`, 91
- `HistoryTableViewController`, 92–100
- modifying `RootTabBarController.h`, 89
- modifying `viewDidLoad`, 90
- `setWeightHistoryDocument:` method, 90

Health Beat UI layout. *See also* Auto Layout
- adding views, 116
- autoresizing asks, 117–118
- change in size, 116–117
- coordinate system, 114
- interface elements, 112
- main screen and window, 114
- removing views, 116
- reordering views, 116
- `size`, 114–115
- view geometry, 113
- view hierarchy, 113

Health Beat user preferences
- `dealloc` method, 275–276
- explicit, 270
- fixing warnings, 273
- Health Beat in Settings app, 280
- history view, 274
- implicit, 270
- implicit property, 272
- managing, 270–271
- `NSUserDefaults`, 270–272
- passing selector, 275
- removing observer, 275–276
- resource, 311
- saving, 270
- saving defaults, 271–272
- `setDefaultUnits()`, 272
- Settings application, 271
- settings bundle, 278
- system settings support, 278–282
- `updateUI` method, 277
- updating fonts, 277

Hello World object graph, 37

`Hello World-Info.plist` file, 18

History Table, updating fonts in, 181–182

history view, updating, 274

`HistoryTableViewController`. *See also* table view controllers
- `count` property, 94
- data source methods, 92
- Health Beat project model, 92–100
- listening for notifications, 96–100

passing data across segues, 95–96
properties in class extensions, 93
recycling cells, 94
reloading data, 92–93
return entries to history, 93
returning cells, 94

I

iCloud. *See also* file versioning
checking for conflicting state, 352
delayed updates, 351
document storage strategies, 322–323
fallback store, 390–391
hash methods, 358–359
integration with Core Data, 389
Logging Profile, 360
merging conflicting versions, 351
merging history, 354–356
migrating documents to, 339–342
moving files back from, 323
removeConflictedVersions, 356–358
`resolveConflicts` method, 352–353
resolving conflicts, 350–359
resources, 360, 429
showing versions to users, 351
three-way merges, 389
usage details, 360
iCloud APIs
Document Storage, 316–323, 359–360
Key-Value Storage, 314–316
saving files, 318
iCloud best practices
device bit size, 316
device-specific data, 316
passwords, 316
SQLite databases, 316
"sync storms," 316
iCloud syncing, 314, 390
iCloud users, monitoring, 334–337
icons
`iTunesArtwork`, 473
placeholders, 473
providing, 472–473
sizes, 472–473
Identity inspector, 14
images, adding to Health Beat, 60–61
info button, 28–29, 31, 36
`init` method
creating for `WeightHistoryDocument`, 73
using with line graph, 204
`WeightHistoryDocument` for Core Data, 404
initialization methods, adding to `WeightEntry`, 68
`initWithCoder:`, 193, 306
`insertSubview:*`, 116
inspectors, 13–14
`interactive` property, 247
interactive transitions
animator object, 242–246
canceling, 256
explained, 241
gesture recognizers, 249–256
percent-driven, 242
`TabInteractiveTransition`, 246–247
testing custom animation, 248
interactivity, adding, 248–249
Interface Builder
Align tool, 130–131
custom table view cells, 137
Dock, 23
Document Outline, 23
editing constraints, 129–132
explained, 21
Pin tool, 130–131
Resizing Behavior, 130–131
Resolve Auto Layout Issues, 130–131
segue, 23
interfaces, custom drawn, 219
internationalization, 483. *See also* localization tools
iOS
concurrent programming, 290–291
human interface guidelines, 483
versions and devices, 483
iOS 7, handling of bars, 201
iOS App Programming Guide, 49
iOS apps, state preservation and restoration, 311

iOS Developer Program, cost to join, 6
iOS devices, running and testing, 62
iOS file system
  directories, 265–270
  sandbox, 260–266
iOS Utility Application. *See also* projects
  actions, 25
  adding outlet, 40–42
  adding project to workspace, 9
  adding text field, 43–45
  aligning label, 39
  `AppDelegate`, 31
  Assistant editor, 40
  background color, 38
  bootstrapping, 30
  center vertically constraint, 39
  Class Prefix field, 8
  Clear Button attribute, 47
  collapsing Document Outline, 40
  Company Identifier field, 8
  connections, 27
  Connections inspector, 25
  constraints, 27–28
  creating, 7–9
  declaring window property, 31–33
  delegate methods, 35, 37
  delegates, 31
  displaying connection, 26
  Done button, 36
  `done:` method, 45
  examining classes in scenes, 25
  Exit, 24
  files, 18–19
  First Responder, 23
  fixing navigation bar, 43
  flipside view, 48
  flipside view controller, 26
  `FlipsideViewController.h`, 36
  functional code, 34–35
  git repository, 9
  Hello World object graph, 37
  `Hello World-Info.plist` file, 18
  implementation, 34
  info button, 28, 31
  jump bar, 45
  label and constraints, 39
  landscape orientation, 29
  layer of indirection, 20
  layout guides, 25
  `main()`, 20
  main view, 48
  `main.m` file, 18–20
  `Main.storyboard` file, 18
  `MainViewController`, 34
  method stubs, 34
  methods, 25
  modal view, 29
  modifying main view, 38–40
  passing data to scenes, 36
  prefixes for class names, 8–9
  `Prefix.pch` file, 18–19
  pressing info button, 36
  preventing naming conflicts, 8
  refining interface, 46–48
  relationships, 25
  Return key and done: method, 47
  running, 17
  Save sheet, 9
  scenes, 23
  segue attributes, 29
  segue identifiers, 35
  segues, 25, 29
  selecting project options, 9
  setting font, 40
  `showAlternate` segue, 29
  source control, 9
  Supporting Files subgroup, 18
  `text` property, 45
  `UIApplication` object, 21
  `UIApplicationMain()` function, 20, 31
  user interface, 21
  view controller, 23–25
iPhone apps, default behavior, 111
Issue tab in Navigator, 12
iTunes, sharing with, 264
iTunes Connect Developer Guide, 484
`iTunesArtwork`, 473

J

jump bar, 45

K

Key Bindings preferences, 16
keyboard
  avoiding in view presentation, 233–238
  making room for, 159
keychain data, 265
keychain services resource, 311
keyed archives, using, 283
Key-Value Storage API, 314–316
key-value syncing, file versioning, 332–333
KVC operators, using, 152

L

label
  aligning, 39
  changing text for, 44
  with constraints, 39
  editing, 39
  resizing, 39
layout guides, using, 25, 123. *See also* Auto Layout
layout timing
  Display phase, 136
  Layout phase, 135–136
  Update Constraints phase, 135
libraries, 13–14
line graph. *See also* passing data for line graph
  adding, 202–203
  adding properties to class, 203–204
  bitmaps, 219
  constraints, 202
  `drawRect:` method, 217, 219
  `init` methods, 204
  resizing, 202–203
  `setDefaults` method, 205
listening for notifications, 96–100
`loadFile`, modifying, 348–350
`loadFileAtURL:`, 288
localization, 483
localization tools
  `genstrings`, 478
  `ibtool`, 478
  `NSLocalizedString()`, 478
localized string, creating for `WeightEntry`, 69
Locations preferences, 16
log messages, deleting, 110
Logging Profile, downloading for iCloud, 360
low memory warnings, best practices for, 103

M

`main()` file, 20
main view, modifying, 38–40
`main.m` file, 18
`Main.storyboard` file, 18
`MainViewController`, 34
managed object context
  asynchronous saves, 373–374
  calling `reset`, 386
  concurrency patterns, 373
  creating projects, 374
  fetch requests, 378–381
  hierarchies, 374
  nesting, 373
  nonstandard attributes, 376–378
  `NSPredicate`, 379–381
  opening, 373
  predicates, 379–381
managed objects
  accessing, 378
  accessors, 374–375
  custom accessors, 375
  custom subclasses, 375
  declaring object properties, 375
  `NSFetchRequest`, 378
  `NSManagedObject`, 375
  overriding methods, 375
masks, autoresizing, 117–118
methods
  declaring for `WeightHistoryDocument`, 71
  defining for `WeightHistoryDocument.m`, 72–73
migrating data via Core Data, 369–371

modal view, 29
model, 52
model layer tasks. *See* Core Data
modules
  explained, 18–19
  model, 52
  view controllers, 52
  view hierarchies, 52
motion effects
  adding to Add Entry View, 434–438
  adding to views, 433
  creating, 432–433
  disabling, 433
  minimum relative value, 437
  removing, 435
  setting up, 436–437
  `setupMotionEffects`, 435
  `UIInterpolatingMotionEffect`, 434
  `UIMotionEffect`, 433
motion events resource, 469
Music app, 52

N

navigation bar
  button, 82
  fixing, 43
navigation items, adding to storyboard, 85–88
Navigation preferences, 16
navigation techniques, 83
Navigator, 11
Navigator tabs
  Breakpoint, 12
  Debug, 12
  Find, 12
  Issue, 12
  Log, 12
  Project, 12
  Symbol, 12
  Test, 12
nesting
  containers, 54
  managed object contexts, 373
`.nib` extension, 21
nib file
  creating `UINib` object for, 168
  using, 30
nibs
  explained, 86
  function, 86–87
  loading, 86
  loading objects from, 193
notifications
  Dynamic Type, 180–181
  listening for, 96–100
  private methods, 414–422
`NSCalendar` object, using, 155
`NSCoder`, 282
`NSCoding`, 282, 305–307
`NSFetchedResultsControllerDelegate`, 412–414
`NSFileManager():`, 269
NSHipster, 188
`NSKeyedArchiver`, 282–283
`NSKeyedUnarchiver`, 282–283, 305
`NSUserDefaults`, 270–272

O

Object library, 14
Objective-C, 5, 75
objects, loading from nibs, 193. *See also* data objects
observer
  removing, 275–276
  removing observer, 297
opaque views, 194–195
opening apps, best practices for, 102
opening files, 12
`OpenWithCompletionHandler:`, 288
Option-clicking, 34
outlets
  Add Entry button, 299–300
  adding, 40–42
  connecting, 139–140
  creating for Entry Detail View, 149–150
  graph view, 203
overriding `loadFromContents:ofType:error:`, 292–294

P

PaintCode, 257
passing data for line graph. *See also* line graph
  adding guidelines, 214–217
  calculating x-coordinate, 211–212
  calculating y-values, 212–213
  checking for entries, 211, 217
  copying history array, 207
  drawing graph, 209
  Dynamic Type font, 215
  `GraphView.m`, 207
  method for drawing dots, 213
  overriding `HBT_updateFonts`, 218
  private `drawGraph` method, 210–211
  rectangle for text, 215
  `textRect`, 216
  weight history document, 206
  weight values, 217
  `weightEntries` property, 208–209
passing data to scenes, 36
paths, exploring in sandbox, 267–268
persistent stores
  custom incremental, 384
  encryption options, 385
  faults, 383
  in-memory, 383
  reducing memory overhead, 383–384
  speed, 382–383
PNG files, using, 220
predicates
  key paths, 380–381, 384
  resource, 429
  using in managed object contexts, 379–381
preference page, 280
preferences
  `dealloc` method, 275–276
  explicit, 270
  fixing warnings, 273
  Health Beat in Settings app, 280
  history view, 274
  implicit, 270
  implicit property, 272
  managing, 270–271
  `NSUserDefaults`, 270–272
  passing selector, 275
  removing observer, 275–276
  resource, 311
  saving, 270
  saving defaults, 271–272
  `setDefaultUnits()`, 272
  Settings application, 271
  settings bundle, 278
  system settings support, 278–282
  `updateUI` method, 277
  updating fonts, 277
prefixes
  using with class names, 8–9
  using with shared code, 9
private methods
  `fetchAllEntries`, 419
  `refreshAll`, 422
  `removeNotifications`, 417–418
  `resetMocs`, 420–421
  `setupNotifications`, 414
  `setupResultsController`, 418–419
  `updateMOCFromNote:CompletionHandler:`, 421
`processQuery:` method, 347–348
programming
  concurrency, 290–291, 311
  file system, 311
  keychain services, 311
  preferences and settings, 311
  state preservation and restoration, 311
programming guides
  Core Data, 429
  iOS app, 49
  key-value coding, 107, 188
  predicates, 429
  Quartz 2D, 257
  resources, 107
  view controllers, 107, 188
Project tab in Navigator, 12
projects. *See also* Health Beat project; iOS Utility Application
  adding to workspace, 9
  walking through, 37

properties, declaring for WeightHistory Document, 71
push segue, selecting, 87

Q

Quick Help inspector, 13

R

rectangle, drawing for text, 216
refactoring code, 287
relationships
  explained, 25
  versus segues, 26
remove FromSuperview:, 116
removeConflictedVersions, 356–358
required capabilities
  deployment target, 476–477
  keys, 476
resolveConflicts method, 352–353
resources
  accessibility, 483
  app distribution, 484
  App Store Review Guidelines, 484
  Asset Catalog Help, 483
  Auto Layout, 188
  concurrency, 311
  core animation programming, 257
  Core Data, 429
  custom transitions, 257
  event handling, 257, 469
  file coordination, 360
  file system, 311
  iCloud, 360, 429
  internationalization, 483
  iOS App Programming Guide, 49
  iOS Human Interface Guidelines, 483
  iOS versions and devices, 483
  iTunes Connect Developer Guide, 484
  keychain services, 311
  key-value coding, 107
  motion events, 469
  NSHipster, 188
  PaintCode, 257
  predicate programming, 429
  preferences and settings, 311
  state preservation and restoration, 311
  storyboard techniques, 107
  UIKit Dynamics, 469
  view controllers, 107, 188
  warnings, 107
  WWDC Videos, 188
  Xcode, 49
responder chain, 164
Retina displays
  drawing for, 196–197
  support for, 195
Return key, connecting to done: method, 47
Root.plist, 279, 281
RootTabBarController.h, modifying, 89
rotations, defaults, 111
rows, deleting from screen, 157
running iOS devices, 62

S

sample project. *See* projects
sandbox. *See also* Document Storage API
  accessing, 261
  accessing directories, 265–270
  backing up data, 263–264
  expanding, 337–339
  exploring, 269
  extending, 265
  keychain data, 265
  manipulating file system, 267
  NSBundle():, 266
  NSFileManager():, 266, 269
  NSHomeDirectory():, 265
  NSTemporaryDirectory():, 265
  NSURLs, 265
  paths, 266–270
  Security framework, 265
  sharing with iTunes, 264
  URLsForDirectory():, 266
sandbox directories
  Application Bundle, 262
  Application Support, 263

Caches, 263
Documents, 262
Documents/Inbox, 262
Library, 262
Preferences, 263
Temporary (tmp), 262–263
Save button
adding to bottom of screen, 160
advice against use of, 304–305
saving
approaches toward, 304–305
files, 318
Health Beat project, 56
and loading data objects, 282
`searchForCloudDocument` method, 345–346
searching
Ubiquity Container, 346
workspace, 110
Security framework, 265
segue identifiers, checking, 35
segue unwinding, 176–177
segues
adding to storyboard, 85–88
explained, 25
modal, 221
passing data across, 95–96
versus relationships, 26
selector, passing, 275
`sendSubviewToBack:`, 116
Settings application, 271, 280
settings bundle, adding, 278
settings resource, 311
`Settings.bundle` file, 278
`setupDocument` method, 299
`setupMotionEffects`, 435
`setWeightHistoryDocument:` method, 90
shared code, using prefixes with, 9
sharing
files with applications, 479–480
with iTunes, 264
`showAlternate` segue, 29
simulator, using, 17
size, responding to change in, 116–117
Size inspector, 14, 28
sort descriptors, 409–410
Source Control preferences, 16
SQLite stores, 384, 389
start-up sequence. *See* document start-up sequence
state and UI controls, 175
state preservation and restoration, 311
static table, configuring, 82. *See also* table view controllers
storage. *See* iCloud APIs
stores. *See* persistent stores
Storyboard editor
Document Outline region, 22
Interface Builder, 22
storyboard for Health Beat. *See also* Health Beat project
adding weight entries, 82
application workflow, 88
configuring static table, 82
connecting controllers, 81
custom view controllers, 84–85
graph view, 81
history view controller, 80
navigation bar button, 82
navigation items, 85–88
navigation techniques, 83
push segue, 87
scene layout, 83
segues, 85–88
tab bar items, 85–88
table view controller, 81
table view controller subclasses, 84
tabs, 80
`UIViewController` subclasses, 84
use of background view, 200
storyboards
contents, 21
deleting controllers from, 58
initial scenes, 24
opening, 22
setting, 22
using, 30
subpixel drawing, 198

subscriber method, implementing, 302–303
subscriber notification method, 297
`supportedInterfacedOrientations` method, 111–112
Supporting Files subgroup, 18
Symbol tab in Navigator, 12
"sync storms," 316
system settings support, 278

T

tab bar controller, adding, 84
tab bar items, adding to storyboard, 85–88
tabbed application, creating, 56
`TabInteractiveTransition`
  animation methods, 247
  `interactive` property, 247
table view controllers. *See also* custom table view cells; `HistoryTableViewController`; static table
  configuring contents, 92
  high performance, 94
  subclasses, 84
template
  app delegate, 57
  running for Health Beat, 57
Test tab in Navigator, 12
testing
  architecture for Health Beat, 100–102
  custom animation, 248
  iOS devices, 62
Text Editing preferences, 16
text fields
  adding, 43–45
  updating, 161–164
`text` property, using, 45
`textRect`, 216
`tintColor:` method, 186
to- controller, 228
`// TODO`: comments, 110
toolbar in Xcode
  active scheme, 10–11
  Assistant Editor, 10–11
  hiding and showing, 12
  Run button, 10–11
  setting destination, 10–11
  Standard Editor, 10–11
  Status window, 10–11
  Stop button, 10–11
  toggles, 11–12
  Version Editor, 10–11
touch events. *See also* gesture recognizers
  handling, 249
  responding to, 164
transitions. *See* custom transitions; interactive transitions

U

Ubiquity Container, 317–318, 346
UI controls and state, 175
UI layout for Health Beat. *See also* Auto Layout
  adding views, 116
  autoresizing asks, 117–118
  change in size, 116–117
  coordinate system, 114
  interface elements, 112
  main screen and window, 114
  removing views, 116
  reordering views, 116
  `size`, 114–115
  view geometry, 113
  view hierarchy, 113
`UIActionSheet`, avoiding use of, 299
`UIAlertView`, avoiding use of, 299
`UIAppearance`, 186
`UIApplicationMain()` function, 20, 31
`UIBezierPath`, 200
`UiCollectionViewController`, 53
`UIDocument` subclass. *See also* Document Outline; document state changes
  adding to Health Beat, 284–286
  concurrent programming, 290–291
  `createNewFileATURL:` method, 289
  creating document, 286–290
  `documentReady` method, 287
  error messages, 292
  explained, 282, 284

loadFileAtURL:, 288
loading document, 286–290
localFileExists method, 285–286
OpenWithCompletionHandler:, 288
overriding contentsForType:Error:, 289–290
overriding handleError:userInteractionPermitted:, 293
overriding loadFromContents:ofType:error:, 292–294
URL, 285–286
UIDynamicBehaviors
UIAttachmentBehavior, 439
UICollisionBehavior, 439
UIDynamicItemBehavior, 439–440
UIGravityBehavior, 440
UIPushBehavior, 440
UISnapBehavior, 440
UIInterpolatingMotionEffect, 434
UIKit Dynamics
end points for animations, 450
performance notes, 440
resources, 469
UIDynamicAnimator, 439–440
UIDynamicBehaviors, 438–439
UIDynamicItem, 438
UILongPressGestureRecognizer, 250
UIManagedDocument, 385–386
UIMotionEffect, 433
UINavigationController, 53
UINib object, creating for nib file, 168
UIPanGestureRecognizer, 250
UIPinchGestureRecognizer, 250
UIRotationGestureRecognizer, 250
UIScreenEdgePanGestureRecognizer, 250
UISwipeGestureRecognizer, 250
UITabBarController, 53
UITableViewController, 53–54
UITapGestureRecognizer, 250
UIView, animation methods for, 222–223
undo action, adding, 307–310
Units button
adding to Health Beat, 172–174
control states, 174
updateAddButton method, 300–301
updateDateText method, 163
updateUI method, 163, 277
updateWeightText method, 162–163
upgrading applications, 264–265
user preferences
dealloc method, 275–276
explicit, 270
fixing warnings, 273
Health Beat in Settings app, 280
history view, 274
implicit, 270
implicit property, 272
managing, 270–271
NSUserDefaults, 270–272
passing selector, 275
removing observer, 275–276
resource, 311
saving, 270
saving defaults, 271–272
setDefaultUnits(), 272
Settings application, 271
settings bundle, 278
system settings support, 278–282
updateUI method, 277
updating fonts, 277
Utilities area
Attributes inspector, 14
Code Snippet library, 14
Connections inspector, 14
File inspector, 13
File Template library, 14
Identity inspector, 14
inspectors, 13–14
libraries, 13–14
locating, 11
Media library, 14
Object library, 14
Quick Help inspector, 13
Size inspector, 14

V

Version editor, 11
versioning. *See* file versioning
versions, adding in Core Data, 369
view controllers
from- and to-, 228
collection, 54
container view, 226
customizing, 84–85
explained, 52
flipside, 26
performing checks, 95–96
presenting, 55, 226, 228
selecting, 25
tab bar controller, 84
using in scenes, 23
view geometry
`bounds` property, 114–115
`center` property, 114–115
Core Graphics data types, 113
`frame` property, 114–115
view hierarchies, 52, 113
view presentation. *See also* custom transitions
ambiguous layout, 232
animator object, 226–230
avoiding keyboard, 233–238
cleaning up, 231–233
delegate methods, 230–231
fixing rotation, 233
length of animations, 238
making modal, 240
modifying appearance, 232–233
modifying text fields, 232
`viewDidAppear:` method, 153
`viewDidLoad:`, 220
views
adding, 116
`addSubview:`, 116
aligning, 456
`bringSubviewToFront:`, 116
caching content, 195
drawing, 198
`exchangeSubviewAtIndex:withSubviewAtIndex:`, 116
`insertSubview:*`, 116
redrawing, 198–199
`remove FromSuperview:`, 116
removing, 116
reordering, 116
`sendSubviewToBack:`, 116
width and height from `bounds`, 199
`viewWillAppear:` method, 153
visual formatting syntax, 126–129

W

warnings. *See also* error messages
fixing, 273
refactoring code, 287
renaming selections, 287
setting for Health Beat, 58–60
website resources
accessibility, 483
app distribution, 484
App Store Review Guidelines, 484
Asset Catalog Help, 483
Auto Layout, 188
concurrency, 311
core animation programming, 257
Core Data, 429
custom transitions, 257
event handling, 257, 469
file coordination, 360
file system, 311
iCloud, 360, 429
internationalization, 483
iOS App Programming Guide, 49
iOS Human Interface Guidelines, 483
iOS versions and devices, 483
iTunes Connect Developer Guide, 484
keychain services, 311
key-value coding, 107
motion events, 469
NSHipster, 188
PaintCode, 257

predicate programming, 429
preferences and settings, 311
state preservation and restoration, 311
storyboard techniques, 107
UIKit Dynamics, 469
view controllers, 107, 188
warnings, 107
WWDC Videos, 188
Xcode, 49
weight entries, adding, 82. *See also* passing data for line graph
Weight Entry scene, transitioning to, 106
weight history document
keyed archives, 283
`UIDocument` subclass, 282, 284
Weight label, revising, 137
weight loss. *See* Health Beat project
weight strings, formatting, 140–145
weight values, creating strings for, 217
`WeightEntry` class
for conversion to Core Data, 396–400
documentation comments, 66
implementing, 65–69
implementing conversion methods, 68–69
initialization methods, 68
localized string, 69
`WeightHistoryDocument`
accessor methods, 75
`addEntry:` method, 76
block enumeration method, 77–78
`count` public property, 71–72
custom `init` method, 73
declaring methods, 71
declaring properties, 71
`deleteEntryAtIndexPath:` method, 77
getter for `count`, 74
index to index path, 76–77
`insertionPointForDate:` method, 78
overriding description method, 79
string contents for notification names, 70
`WeightHistoryDocument` for Core Data
accessor methods, 404–405
fetch request, 409–410
header file, 401–402
implementation file, 402–404
`init` and `dealloc` methods, 404
`NSFetchedResultsControllerDelegate`, 412–414
private methods, 414–422
public methods, 405–411
sort descriptors, 409–410
`WeightHistoryDocument.m`, 72–73
weights, displaying, 145–146
`WeightUnit`, implementing, 64
window property, declaring, 31–33
workspace
adding projects to, 9
creating, 7
searching, 110
WWDC Videos, 188

X

Xcode
Debug area, 11, 14–15
Editor, 11, 14–15
Navigator, 12
preferences, 15–16
storyboards, 21
toolbar, 10–12
using, 6
Utilities area, 11, 13–14
Xcode 5.0, downloading, 62
Xcode preferences
Accounts, 16
Behaviors, 16
Double Click Navigation, 16
Downloads, 16
Fonts & Colors, 16
General, 15
Key Bindings, 16
Locations, 16
Navigation, 16
Source Control, 16
Text Editing, 16
Xcode User Guide, 49
`.xib` extension, 21