

VISUAL **QUICKSTART** GUIDE

Get up and running in no time!



*With downloadable
code samples!*

iPhone Application Development for **iOS 4**

DUNCAN CAMPBELL

● LEARN THE QUICK AND EASY WAY!

VISUAL QUICKSTART GUIDE

iPhone Application Development

FOR IOS 4

DUNCAN CAMPBELL



Peachpit Press

Visual QuickStart Guide

iPhone Application Development for iOS 4

Duncan Campbell

Peachpit Press
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at www.peachpit.com.

To report errors, please send a note to errata@peachpit.com.

Peachpit Press is a division of Pearson Education.

Copyright © 2011 by Duncan Campbell

Editor: Whitney Walker and Cliff Colby
Production Coordinator: Danielle Foster
Copyeditor/proofreader: Kim Wimpsett
Technical Editor: James Sugrue
Compositor: Danielle Foster

Indexer: Valerie Perry
Cover Design: RHDG/Riezebos
Holzbaur, Peachpit Press
Logo Design: MINE™ www.minesf.com
Interior Design: Peachpit Press

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 author 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

Visual QuickStart Guide is a registered of Peachpit Press, a division of Pearson Education. Any other product names used in this book may be trademarks of their respective owners.

Apple, Cocoa, Cocoa Touch, Dashcode, iPhone, iPod touch, Safari, and Xcode are trademarks of Apple Inc. registered in the U.S. and other countries.

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: 978-0-321-71968-3

ISBN-10: 0-321-71968-9

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

Dedication

For my son, Hamish.

Acknowledgments

Thanks to *Whitney Walker, Clifford Colby, Kim Wimpsett, Danielle Foster, Valerie Perry*, and everyone else at Peachpit Press who worked so hard to make this book happen.

Thanks to *James Sugrue* for his technical-editing expertise.

A big thank-you to my good friend *Kane Nickolichuck* who all those years ago pestered me relentlessly into buying my first Macintosh computer.

Cuddles and pets to my dog, *Kip*, for again keeping me company during the cold (yes, even in Australia!) winter evenings I spent working on this book.

Finally, the biggest thanks go to my wife, *Sarah*, for single-handedly looking after our newborn son while I spent the evenings locked away in my office each night.

Contents at a Glance

	Introduction	xv
Chapter 1	Objective-C and Cocoa	1
Chapter 2	The iPhone Developer's Toolbox	41
Chapter 3	Common Tasks	83
Chapter 4	iPhone User Interface Elements	111
Chapter 5	Tabs and Tables	181
Chapter 6	Files and Networking	233
Chapter 7	Touches, Shakes, and Orientation	279
Chapter 8	Location and Mapping	311
Chapter 9	Multimedia	349
Chapter 10	Contacts, Calendars, E-mail, and SMS	405
Chapter 11	Multitasking	455
	Index	469

This page intentionally left blank

Table of Contents

Introduction	xv
Chapter 1 Objective-C and Cocoa	1
Frameworks	2
Classes	3
Methods	5
Creating objects	7
Properties	8
Memory Management	10
Autorelease pools	11
Commonly Used Classes	14
Strings	14
Dates and times	20
Arrays	24
Dictionaries	27
Notifications	30
Timers	32
Design Patterns	35
Model View Controller	35
Delegate	36
Target-Action	37
Categories	37
Singletons	39

Chapter 2	The iPhone Developer's Toolbox	41
	About the Xcode IDE	42
	About the Groups & Files pane	44
	Targets	46
	About the toolbar	48
	About the details pane	49
	About the editor pane	50
	Gutter and focus ribbon	52
	Find-and-replace operations	53
	Bookmarks	53
	Jump-to-definition and help	53
	Code completion	54
	About the navigation bar	55
	Creating new files	57
	Building and running your application	58
	Cleaning	59
	About the iPhone Simulator	61
	About Interface Builder	64
	About the document window	65
	About the Library window	67
	About the inspector window	67
	About the Documentation	78
	The Xcode Organizer	79
	Projects & Sources	80
	Devices	80
	iPhone Development	81

Chapter 3	Common Tasks	.83
	Application Startup and Configuration	.84
	Using the application delegate	.84
	Understanding application settings	.87
	Working with user preferences	.87
	Application preferences	.90
	Adding controls	.92
	Localization	.94
	Accessibility	.98
	Making your applications accessible	.99
	Accessibility attributes	.101
	Interapp Communication	.103
	Sharing information between applications	.105
	Using the pasteboard	.109
Chapter 4	iPhone User Interface Elements	.111
	Views	.112
	Frames	.112
	Bounds	.113
	Animation	.115
	Autosizing	.117
	Custom drawing	.118
	Transforms	.123
	Image Views	.126
	Animating images	.127
	Scrolling	.129

Zoom	130
Paging	131
Labels	136
Progress and Activity Indicators	139
Indicating progress	139
Showing activity	140
Alerts and Actions	142
Alerting users	142
Confirming an action	144
Picker Views	146
Toolbars	152
Toolbar items	153
Text	156
To use keyboards:	157
Restricting content	159
Text views	160
Data detectors	161
Hiding the keyboard	161
Scrolling the interface	162
Web Views	164
Running JavaScript	167
Loading local content and handling hyperlinks	168
Controls	170
Buttons	170
Switches	172
Sliders	175
Segmented controls	177

Chapter 5	Tabs and Tables	181
	View Controllers.	182
	Presenting views	183
	Responding to changes in orientation	184
	Displaying modal views	189
	Handling low-memory conditions.	193
	Tab Views	194
	Adding graphics and titles to tabs	196
	Table Views	200
	Grouping rows into sections and styles	204
	Editing and searching table views	210
	Drilling down in table views	217
	Creating custom cells.	223
 Chapter 6	Files and Networking	233
	Files	234
	The file system	236
	Common directories	237
	Working with files.	239
	Previewing documents.	244
	Networking.	248
	Retrieving content from web pages.	248
	Parsing XML	254
	Sending data to Web pages	262
	Responding to HTTP Authentication	266
	Creating peer-to-peer applications.	271

Chapter 7	Touches, Shakes, and Orientation	279
	Touch	280
	Adding tapping support	285
	Adding long-touch support	288
	Multi-Touch Gestures	292
	The iPhone Accelerometer.	298
	Detecting shakes	298
	Determining orientation	299
	Redrawing the interface when the orientation changes	303
	Responding to the accelerometer	307
Chapter 8	Location and Mapping.	311
	About Core Location	312
	Handling location updates.	314
	Testing outside the simulator	315
	Increasing the accuracy	317
	Adding a timeout	318
	Accessing the compass	323
	About Map Kit	325
	Map Overlays	329
	Adding annotations.	333
	Adding reverse geocoding	338
	Putting It All Together.	341
Chapter 9	Multimedia	349
	Playing Audio	350
	Providing more control.	352

	Responding to audio events.	356
	Playing audio in the background	358
	Controlling audio from the background	361
	Recording Audio.	366
	Using the iPhone's Camera.	371
	Taking photos and video.	375
	Playing Video	381
	To gain more control over movie playback	386
	Using the iPod Library	392
	Accessing media items.	392
	Accessing media collections	394
	Using the media picker.	396
	Playing media	398
Chapter 10	Contacts, Calendars, E-mail, and SMS	405
	Working with the Address Book	406
	Group records.	410
	Person records	411
	Adding a User Interface	418
	Picking people	418
	Editing people.	421
	The iPhone Calendar	428
	Events	430
	Viewing event details.	434
	Editing events	438
	E-mail	443
	SMS	450

Chapter 11	Multitasking	455
	What Is Multitasking?	456
	Entering and exiting background mode	457
	Multitasking services	459
	Responding to Local Notifications	466
	Index	469

Introduction

Welcome to the updated version of this Visual QuickStart Guide for iPhone application development.

A lot has happened since the last version of this book was published: In only one short year, not only have we seen the introduction of the revolutionary iPad, but we've also seen the all-new iPhone 4, with its gorgeous high-resolution display and powerful new hardware capabilities.

The tools for iPhone development have also had a major upgrade. iOS 4 brings with it many new application programming interfaces (APIs) that give developers even more access to the iPhone's underlying hardware, as well as adds exciting new capabilities, such as multitasking and high-definition (HD) video recording and editing.

At the time of this writing, more than 250,000 applications are available for download from the iTunes App Store, with more being added every minute—it's an exciting time to be an iPhone developer!

This book is geared mainly toward new iPhone developers, but you should have some prior knowledge of a C-based language and be familiar with object-oriented (OO) concepts. It would take a book many times this size to cover all of the iPhone software development kit (SDK), so I focus on some of the more common and interesting subjects I think you should know about when developing your own iPhone applications.

How to Use This Book

I find that I always learn better by example, so I have created stand-alone applications when demonstrating the concepts in the book. The aim is to give you enough information to get you started coding (and building something useful) and then point you to the relevant place in the documentation for more information.

You should be able to jump straight into a chapter and start coding without reading the prior chapters, but if you are a beginner, I recommend you read the first few chapters, which discuss the tools and language used for iPhone development.

This book is a Visual QuickStart Guide, so it's filled with images to walk you through what you'll see on your computer screen as you build your iPhone applications. However, the interfaces for most of the examples are created directly in code, rather than by using Interface Builder. You might think this is unusual, since Apple has provided you with a powerful tool that makes laying out your application's user interface quick and easy, but it's important that you first learn what's happening under the hood. This will make it much easier for you to figure out where to look when things aren't working the way they should.

The source code for all the examples in this book—more than 65 projects—is available as a free download from my Web site:

<http://objective-d.com/iphonebook/>

I strongly encourage you to check them out.

1

Objective-C and Cocoa

Objective-C is the language most commonly used for iOS development. It is a superset of ANSI-C, with a Smalltalk-style syntax. If you have programmed in any modern language (such as C++, Java, or even PHP), you should be able to pick up Objective-C relatively quickly.

Cocoa is the collective name given to the frameworks provided by Apple for both OS X and iOS development. For the purpose of this book, Cocoa will be used to mean the iOS-specific APIs.

In this chapter, you will get a brief overview of how Objective-C code is structured and how you build your own classes. You'll then learn how memory is managed before learning about some of the more commonly used Cocoa classes. Finally, you'll learn about some of the design patterns used throughout the Cocoa frameworks.

In This Chapter

Frameworks	2
Classes	3
Memory Management	10
Commonly Used Classes	14
Design Patterns	35

Frameworks

iOS provides a set of *frameworks* for development. A framework, such as UIKit, Core Location, Map Kit, Address Book, and Media Player, is simply a collection of classes designed to help you work with a particular technology.

Adding a framework to your projects enables you to work with the classes contained within that framework. Apple groups these frameworks into four main areas of functionality (Table 1.1).

To add a framework to your project:

1. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
2. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and then add the framework from the available list **A**.
3. In the header file of your class, import the framework.

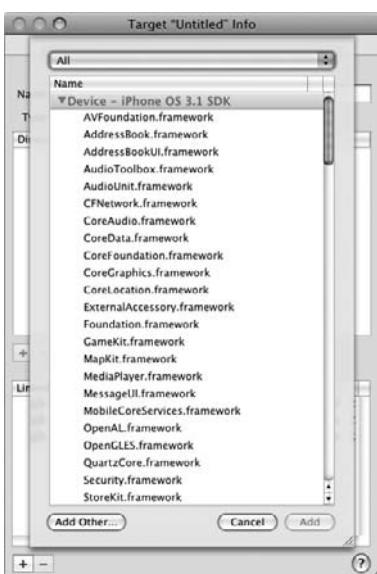
Code Listing 1.1 shows an example of adding a reference to the **CoreAudio** framework to a class.

Code Listing 1.1 Referencing a framework in your code.

```
//  
// UntitledViewController.h  
// Untitled  
//  
  
#import <UIKit/UIKit.h>  
#import <CoreAudio/CoreAudioTypes.h>  
  
@interface UntitledViewController : UIViewController  
{  
}  
@end
```

TABLE 1.1 iOS Frameworks

Framework Group	Description
Cocoa Touch	Frameworks for handling all the touch and event-driven programming as well as access to systemwide interface components such as the Address Book browser, mapping, messaging, and most of the user interface components.
Media	The frameworks used to play and record both audio and video as well as provide support for animation and 2D and 3D graphics.
Core Services	Frameworks for accessing many of the iPhone's lower-level features such as files, networking, location services, in-app purchase support, and configuration information such as network availability.
Core OS	Frameworks providing access to the memory, file system, low-level networking, and hardware of the iPhone.



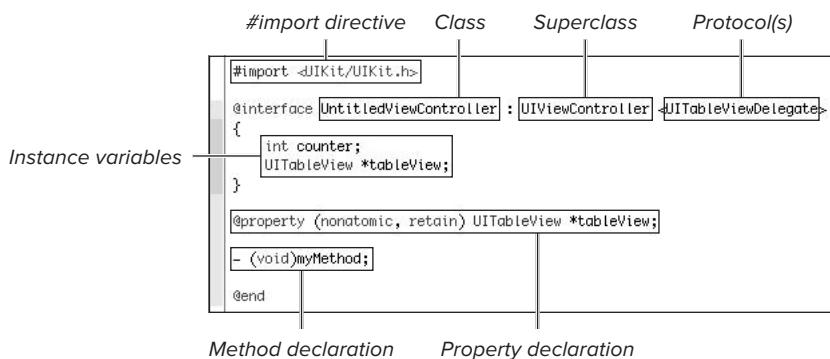
A Adding a framework to your project.

Classes

As with regular C, Objective-C separates classes into two files: the header file and the implementation file. The *header* (.h) file is the public interface to your class; it contains declarations for the properties, instance variables, and methods available.

The header file contains the following parts: A.

- The **#import** directive, much like the C **#include** statement, allows you to include header files in your source code. However, **#import** makes sure the same file is never included more than once.
- The **@interface** line declares your class name and its *superclass*, that is, the parent class from which your class inherits. Any protocols that the class implements are appended to the end within brackets (< and >).
- Next, within the braces ({ and }), you define any instance variables used by your class.
- Finally, you define the methods and property declarations of your class and then close the implementation file with the **@end** directive.



A The header file.

The *implementation* (.m) file is where you implement the code for the methods defined in the header file. You can also implement private methods here that won't be visible to anyone using your class.

The implementation file contains the following parts ❸:

- Again, the **#import** directive is used, this time to import the interface declaration.
- The **@implementation** line begins the area where you write the code for your class.
- You next use the **@synthesize** directive to generate the setter and getter methods for the properties of your class. Notice how they can be on the same line, separated by commas.
- Finally, you write your code, implementing each of the methods defined in the interface file before again closing the implementation with the **@end** directive.

```
#import directive      #import "UntitledViewController.h"
@implementation section @implementation UntitledViewController
Synthesizing a property @synthesize tableView;
Method declaration
- (void)myMethod
{
    counter = 0;
}
- (void)viewDidLoad {
    [super viewDidLoad];
    tableView = [[UITableView alloc] init];
}
- (void)dealloc {
    [tableView release];
    [super dealloc];
}
@end directive        @end
```

The diagram shows the structure of an implementation file. It is divided into several sections: **#import directive**, **@implementation section**, **Synthesizing a property**, **Method declaration**, and **@end directive**. The **Method declaration** section is further subdivided into three methods: `- (void)myMethod`, `- (void)viewDidLoad`, and `- (void)dealloc`. The **@implementation** and **@end** directives are shown at the top and bottom of the main code block respectively.

❸ The implementation file.

Methods

Methods in Objective-C perform an *action* on an object and are surrounded by square brackets:

```
[myObject foo];
```

Here you are calling a method named **foo** on the object **myObject**. The process of calling a method is known as *messaging*—the message is the signature of the method including any parameters that are passed.

Objective-C is a verbose language with long and descriptive method and parameter names. The method and parameter names combine to form a *phrase* explaining the action of the method. A variation of camel case notation is used where the first word is usually lowercase, the first letter of each subsequent word is capitalized, and no spaces appear between words:

```
[myObject performSomeAction];
```

This would call the **performSomeAction** method of **myObject**.

When passing a value into a method, the parameter name will also often describe the data type if the type is important:

```
[myObject saveInteger:10];
```

This will call the **saveInteger** method on the **myObject** object, passing the value **10** to the first parameter.

With multiple parameters, each parameter is named and helps form the phrase describing the purpose of the method. For example, a C function to create a fraction and return the result might look like this:

```
fraction = MakeFraction(10,20);
```

Implemented as an Objective-C method, it might look like this:

```
fraction = [Fraction fractionWith  
→ Numerator:10 denominator:20];
```

Here you are calling the **fractionWithNumerator:denominator:** method on **Fraction**, passing two parameters, and storing the returned value in the **fraction** variable.

The syntax for calling and defining methods is very similar. For example, you could define the previous method as follows:

```
- (double)fractionWithNumerator:  
→ (int)num denominator:(int)denom;
```

Many classes provide what are known as *class methods*—instead of creating an object and then calling a method on it, you can call a method directly on the class itself.

By convention, class methods (other than **+new** and **+alloc**) usually return *autoreleased* objects (see the “Memory Management” section later in this chapter).

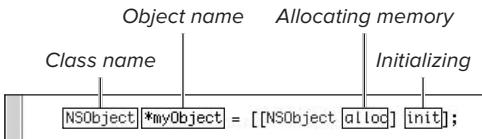
When defining class methods, you prefix the method type identifier with a plus (+) sign:

```
+ (MyClass *)classWithInteger:  
→ (int)iValue
```

Code Listing 1.2 shows an example of some commonly used class methods.

Code Listing 1.2 Some commonly used class methods.

```
- (void)viewDidLoad {  
  
    NSString *myString = [NSString stringWithFormat:@"foobar"];  
    NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithObject:nil];  
    NSMutableArray *arr = [NSMutableArray arrayWithObject:nil];  
  
    UIButton *myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];  
    UIImage *img = [UIImage imageNamed:@"apple.png"];  
    UIFont *font = [UIFont systemFontOfSize:14.0];  
}
```



C Creating an object.

The shorthand for writing methods is to remove the datatype and parameter name from the method signature, leaving a colon (:) to indicate a parameter. For example, the following method:

```
- (NSString *)appendString:  
→ (NSString *) string1 toString:  
→ (NSString *)string2
```

could be shortened to the following:

```
appendString:toString:
```

Creating objects

In general, to create an object in Objective-C, you do the following:

- Define the type of your object, and give it a name.
- Allocate memory with the `alloc` class method.
- Initialize the object with an `init` method.

For example, the following

```
NSObject *myObject;  
myObject = [NSObject alloc];  
[myObject init];
```

would normally be written as a single statement:

```
NSObject *myObject =  
→ [[NSObject alloc] init];
```

In the breakdown of this statement, notice that there are the same number of square brackets in both examples C. This *nesting* of method calls is common in Objective-C, and you will see examples of it throughout this book.

Many classes provide additional *initializer* methods allowing you to perform multiple steps in a single method call. For example, to create an **NSString** and assign it a value, you can use the following:

```
NSString *myString =  
→ [[NSString alloc] initWithString:  
→ @"some value"];
```

You can also use the class method:

```
NSString *myString = [NSString  
→ stringWithFormat:@"some value"];
```

(**NSString** contains many of these initializer methods, which are discussed in the “Commonly Used Classes” section later in this chapter.)

Properties

Properties provide a convenient way for you to get and set instance variables on objects without having to define or use accessor (commonly known as *getter* and *setter*) methods.

- For example, if you want to create a new **UIView** object, you write the following:

```
UIView *myView = [[UIView alloc]  
→ init];
```

- You can then set the **backgroundColor** property:

```
myView.backgroundColor = [UIColor  
→ redColor];
```

- You can also retrieve the value with the same property:

```
UIColor *bgColor =  
→ myView.backgroundColor;
```

Code Listing 1.3 Defining properties.

```
@interface UntitledViewController : UIViewController
{
    int counter;
    NSString *username;
    NSString *language;
    NSNumber *age;
}

@property int counter;
@property (copy, readwrite) NSString *username;
@property (readonly, assign) NSString *language;
@property (retain) NSNumber *age;
```

Code Listing 1.4 Synthesizing properties.

```
@implementation UntitledViewController

@synthesize counter;
@synthesize username,language;
@synthesize age;
```

As already mentioned, properties are defined using the **@property** keyword defined in the class header (.h) file (**Code Listing 1.3**).

Notice how you can define properties as being **readonly** and also how the set accessor will be implemented—as a direct assignment (which is the default), as a **retain**, or as a **copy** of the object being used for assignment.

In the class implementation (.m) file, using the **@synthesize** keyword will automatically generate the getter and setter methods (**Code Listing 1.4**).

TIP For more information on properties, refer to the “Declared Properties” section of the *Objective-C 2.0 Programming Language Guide* in the developer documentation.

Memory Management

In the “Creating objects” section earlier in this chapter, you created an object:

```
NSObject *myObject =  
→ [[NSObject alloc] init];
```

Objective-C uses a process known as *reference counting* for managing memory. When an object is created (in this example by calling the **alloc** method), it contains a reference count—also known as a *retain* count—of one. From then on, each time the object is referenced by anyone (by calling its **retain** method), the reference count *increases* by one. When you are finished with the object, you call its **release** method, which then *decreases* the reference count by one. When the reference count reaches zero, the object’s memory is freed from the system.

Objects that aren’t released after being retained will leak memory, so it’s important to make sure you always release your objects when you are finished with them. Conversely, you need to know when you should retain an object created by someone else; you don’t want an object to be released if you are still working with it, and you also don’t want to release an object you have not retained.

One useful habit when working with objects is to release them as early as possible. Consider the following code:

```
UILabel *myLabel = [[UILabel alloc]  
→ init];  
  
myLabel.text = @"some text";  
[myView addSubview:myLabel];  
[myLabel release];
```

You first create a label, which will set its retain count to one. After setting the text, you add the label to a view. This will increase the retain count to two (the view calls **retain** on the label when it is added as a subview). You no longer need the label (the view now owns it), so you then release it on the next line. You can now safely forget about managing memory for the label—you balanced your **retain**/**release** calls, and the view will release its subviews (and therefore the label) by itself.

This pattern of releasing an object as soon as you are done with it (rather than waiting until later in your code) is a good one to use and helps reduce the likelihood of memory leaks.

Autorelease pools

To make things a little easier to work with, Objective-C provides an *autorelease pool*.

Consider the following example:

```
- (NSString *)makeUserName
{
    NSString *name = [[NSString alloc]
                      → initWithString:@"new name"];
    return name;
}
```

Here you have created a new string and are returning it from a method. Unfortunately, someone using this method has no way of knowing that they are supposed to call **release** on the string being returned, and therefore you'd have a memory leak. You obviously can't call **release** inside the method because this would set the retain count to zero and you would have nothing to return.

The solution to this situation is to use an autorelease pool:

```
- (NSString *)makeUserName
{
    NSString *name = [[NSString alloc]
→ initWithString:@"new name"];
    return [name autorelease];
}
```

Objects created in the autorelease pool do not need to have the **release** method explicitly called but instead will release themselves at some point in the future—typically when the autorelease pool itself is released.

The disadvantage here, of course, is that while an object exists, it's using memory. If you create a lot of autoreleased objects, you will use up more memory, which may have a detrimental effect on your application's performance. Because of the limited memory resources on the iPhone, it's a good idea to manually manage memory yourself (using **retain/release**) whenever possible.

Objects created by calling a class method will generally return an autoreleased object. For example, the following:

```
UIButton *myButton = [UIButton
→ buttonWithType:UIButtonTypeRounded
→ Rect];
```

returns an autoreleased object that you don't need to (and should not) release.

Of course, you can still call **retain** and **release** on autoreleased objects if you like, which might be important if you want to hold on to an object. Just make sure you never call **release** without first calling **retain**. Doing so will cause an error and likely crash your application.

Remember this basic rule of thumb: Any time you call the `alloc`, `copy`, or `retain` methods on an object, you must at some point later call the `release` method.

If you are creating a lot of autoreleased objects (for example, within a loop), it can often help to create your own autorelease pool at the start of your loop and then free it manually at the end. This gives you the best of both worlds: You don't have to worry about leaking memory with manually created objects, and you can keep your memory usage under control more efficiently by manually releasing the objects in your own pool.

Code Listing 1.5 shows an example of creating and using your own autorelease pool.

TIP You may have noticed in the first example that you didn't have to create an autorelease pool. This is because all iOS applications have a global autorelease pool created within the `main.m` file (the entry point for all iOS applications).

TIP In most cases, the Cocoa Touch frameworks use a naming convention to help you decide when you need to release objects: If the method name starts with the word `alloc`, `new`, or `copy`, then you should call `release` when you are finished with the object.

TIP For more information on memory management, refer to the *Memory Management Programming Guide for Cocoa* in the developer documentation.

Code Listing 1.5 Using an autorelease pool.

```
- (void)myMethod
{
    NSString *myString = @"some value";
    for (int i=0; i<9999; ++i)
    {
        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        NSString *myString2 = [myString stringByAppendingString:
                               [[NSString stringWithFormat:@"%d", myString, i]]];
        [pool release];
    }
}
```

Commonly Used Classes

Of the hundreds of classes available in the Cocoa Touch frameworks, you will use a couple of them frequently, even in the simplest of applications.

Strings

Probably the most common class you will use will be **NSString**. An **NSString** is *immutable*, meaning that once you have created one, you cannot change it. If you need to alter the contents of an **NSString**, you should use the **NSMutableString** class to create a *mutable* string. However, it's much more common to just create a new **NSString** with the new contents.

- Of the many ways to create an **NSString**, the simplest is probably the following:

```
NSString *myString =  
    @"some string";
```

- To create a formatted string, you could use the following code:

```
NSString *myString = [NSString  
    stringWithFormat:@"object =  
    %@", someObject];
```

Some of the more common format specifiers are **%d** for **integer**, **%f** for **double**, and **%@** for **objects**. (For a complete list of available format specifiers, refer to the “String Format Specifiers” section of the *String Programming Guide for Cocoa* in the developer documentation.)

- If you have strings that contain only numbers, you can return numeric values by using the following:

```
NSString *myString = @"12345";  
double doubleString = [myString  
→ floatValue];  
  
int intString =  
→ [myString intValue];
```

Both of these methods attempt to create numeric values up to the first non-numeric character in the string, so, for example, a string of "**123abc**" would return **123** for the **intValue** method.

- You can get the length of a string:

```
int stringLength = [myString  
→ length];
```

- To compare two strings, you can use the following:

```
BOOL areEqual = [string1  
→ isEqualToString:string2]
```

This will return **TRUE** if all the characters in both strings are exactly equal.

- To perform a case-insensitive comparison, you can use this:

```
BOOL areEqual = ([string1  
→ caseInsensitiveCompare:string2]  
→ == NSOrderedSame);
```

- You can also convert the case of a string:

```
NSString *myString = "abcdef";  
NSString *upper = [myString  
→ uppercaseString];  
  
NSString *lower = [myString  
→ lowercaseString];
```

continues on next page

- You can easily trim a string of unwanted characters. For example, to remove all whitespace from a string, you could use this:

```
NSString *myString = @" one two  
→ three ";  
  
NSString *trimmed =  
→ [myString string  
→ ByTrimmingCharactersInSet:  
→ [NSCharacterSet whitespace  
→ CharacterSet]];
```

This will give you the string "one two
three".

You can create substrings from existing strings in several ways:

- For example, to create a new string with the contents "one" from this string:

```
NSString *numberString = @"one  
→ two three";
```

you can use the following code:

```
NSString *aString = [numberString  
→ substringToIndex:3];
```

- You can use this:

```
NSRange range = NSMakeRange(4,3);  
  
NSString *aString = [numberString  
→ substringWithRange:range];
```

to create a new string with the contents "two".

- Finally, you can use the following:

```
NSString *aString = [numberString  
→ substringFromIndex:8];
```

to create a new string with the contents "three".

- You can also create an array containing these three substrings as elements (using the space character as a delimiter) by using this:

```
NSArray *arr = [numberString  
→ componentsSeparatedByString:  
→ @" "];
```

This will give you the array
`{"one","two","three"}`.

- To replace substrings in your strings, you can use the following:

```
NSString *aString = [numberString  
→ stringByReplacingOccurrencesOf  
→ String:@"three" withString:  
→ @"four"];
```

This will give you the string `"one two four"`.

- You can search for a substring within a string:

```
NSRange foundRange =  
→ [numberString  
→ rangeOfString:@"two"];
```

This will return the range `{4,3}` (indicating a match was found at position 4 with a length of 3).

- You can determine whether a string contains a substring:

```
BOOL found = ([numberString  
→ rangeOfString:@"two"].location !=  
→ NSNotFound);
```

- You can combine strings:

```
NSString *string1 = @"one";  
NSString *string2 = [string1  
→ stringByAppendingString:  
→ @" two"];
```

This will give you the string `"one two"`.

NSString also contains numerous functions for dealing with files. You can read from and write to files, as well as get information such as the file path and extension.

- For example, to read the contents of a file into a string, use the following:

```
NSString *fileContents = [NSString  
→ stringWithContentsOfFile:  
→ @"myfile.txt"];
```

- You can get the file extension of a file:

```
NSString *fileName =  
→ @"myfile.txt";  
  
NSString *fileExtension =  
→ [fileName pathExtension];
```

You can also use an **NSString** to both read and write to a URL.

- For example, to read the contents of a URL into your string, you can use this:

```
NSURL *url = [NSURL URLWithString:  
→ @"http://google.com"];  
  
NSString *pageContents = [NSString  
→ stringWithContentsOfURL:url];
```

Code Listing 1.6 shows some commonly used string methods.

TIP For more information on **NSString**, refer to the **NSString Class Reference** in the developer documentation.

Code Listing 1.6 Some commonly used string methods.

```
- (void)myStringExample
{
    NSString *fileName = @"somefile.txt";
    NSString *fileExtension = [fileName pathExtension];

    NSString *myString = @" one two three ";
    NSString *trimmed = [myString stringByTrimmingCharactersInSet:
        [NSCharacterSet whitespaceCharacterSet]];

    NSString *string1 = @"some string";
    NSString *string2 = @"some other string";
    NSString *string3 = @"SoMe StrINg";
    NSString *string4 = @"some other string";

    NSLog(@"%@",[string2 isEqualToString:string4]);
    NSLog(@"%@",[string1 isEqualToString:string3]);
    NSLog(@"%@",([string1 caseInsensitiveCompare:string3] == NSOrderedSame));

    NSString *n = @"12345";
    double d = [n doubleValue];
    int i = [n intValue];
    NSLog(@"%@",d,i);

    NSURL *url = [NSURL URLWithString:@"http://google.com"];
    NSString *pageContents = [NSString stringWithContentsOfURL:url];

    NSString *numberString = @"one two three";
    NSRange foundRange = [numberString rangeOfString:@"two"];
    NSString *oneString = [numberString substringToIndex:3];
    NSString *twoString = [numberString substringWithRange:NSMakeRange(4,3)];
    NSString *threeString = [numberString substringFromIndex:8];

    NSString *repeatString = [numberString
        stringByReplacingOccurrencesOfString:@"three"
        withString:@"four"];
    BOOL found = ([numberString rangeOfString:@"two1"].location != NSNotFound);

    NSString *stringa = @"one";
    NSString *stringb = [stringa stringByAppendingString:@" two"];

    NSLog(@"%@", foundRange.location, foundRange.length);
    NSLog(@"%@",repeatString);
    NSLog(@"%@",[NSArray arrayWithObjects:oneString,twoString,threeString]);
    NSLog(@"%@",[numberString componentsSeparatedByString:@" "]);
}
```

Dates and times

You use the **NSDate** class to compare dates and calculate date and time intervals between dates.

- You can create an **NSDate** with the current date and time:

```
NSDate *myDate = [NSDate date];
```

- You can create an **NSDate** that represents 24 hours from now:

```
NSTimeInterval secondsPerDay =  
→ 24*60*60;
```

```
NSDate *tomorrow = [NSDate  
→ dateWithTimeIntervalSinceNow:  
→ secondsPerDay];
```

- You can also create a date from an existing date by using the following:

```
NSTimeInterval secondsPerDay =  
→ 24*60*60;
```

```
NSDate *now = [NSDate date];
```

```
NSDate *yesterday = [now  
→ addTimeInterval:-secondsPerDay];
```

This will create a date representing this time yesterday.

- You can compare whether two dates are exactly equal:

```
BOOL sameDate =  
→ [date1 isEqualToDate:date2];
```

- Or, to get which date occurs before or after another date, you can use the following:

```
NSDate *earlierDate =  
→ [date1 earlierDate:date2];
```

```
NSDate *laterDate =  
→ [date1 laterDate:date2];
```

- You can calculate how many seconds occurred between two dates:

```
NSTimeInterval  
→ secondsBetweenDates = [date2  
→ timeIntervalSinceDate: date1];
```

- Or you can calculate how many seconds occurred between now and a date in the future:

```
NSTimeInterval  
→ secondsUntilTomorrow =  
→ [tomorrow timeIntervalSinceNow];
```

By using the **NSCalendar** class, you can create **NSDate** objects more easily.

- For example, to create a date representing June 01, 2010, use the following:

```
NSDateComponents *comp =  
→ [[NSDateComponents alloc] init];  
[comp setMonth:06];  
[comp setDay:01];  
[comp setYear:2010];  
NSCalendar *myCal =  
→ [[NSCalendar alloc]  
→ initWithCalendarIdentifier:  
NSGregorianCalendar];  
NSDate *myDate =  
→ [myCal dateFromComponents:comp];
```

- Similarly, to get the day, month, and year components from an existing date, you could use this:

```
unsigned units =  
→ NSMonthCalendarUnit |  
→ NSDayCalendarUnit |  
→ NSYearCalendarUnit;  
NSDate *now =[NSDate date];  
NSCalendar *myCal =  
→ [[NSCalendar alloc]  
→ initWithCalendarIdentifier:  
NSGregorianCalendar];  
NSDateComponents *comp = [myCal  
→ components:units fromDate:now];  
NSInteger month = [comp month];  
NSInteger day = [comp day];  
NSInteger year = [comp year];
```

Calendars also make it a little easier when creating dates from existing dates since you don't have to convert everything to and from seconds.

- For example, to rewrite the previous example of creating a date representing tomorrow, use the following:

```
NSDate *now = [NSDate date];  
NSDateComponents *comp =  
    → [[NSDateComponents alloc] init];  
  
[comp setDay:01];  
  
NSCalendar *myCal =  
    → [[NSCalendar alloc]  
    → initWithCalendarIdentifier:  
    → NSGregorianCalendar];  
  
NSDate *tomorrow = [myCal  
    → dateByAddingComponents:comp  
    → toDate:now options:0];
```

NSDate in itself is not particularly friendly when you want to present human-readable dates and times to the user. For this, you would normally use an **NSDateFormatter**.

- To get a string representation of the current date using an **NSDateFormatter**, use the following:

```
NSDate *now = [NSDate date];  
NSDateFormatter *formatter =  
    → [[NSDateFormatter alloc] init];  
  
[formatter setDateStyle:  
    → NSDateFormatterMediumStyle];  
  
NSString *friendlyDate =  
    → [formatter stringFromDate:now];
```

- To get the current time, you can use the following:

```
NSDate *now = [NSDate date];
NSDateFormatter *formatter =
    [[NSDateFormatter alloc] init];
[formatter setTimeStyle:
    NSDateFormatterMediumStyle];
NSString *friendlyTime =
    [formatter stringFromDate:now];
```

Table 1.2 shows the five predefined formatter styles.

- Finally, you can also use the `dateFormat` property of a date formatter to manually set a style:

```
NSDate *now = [NSDate date];
NSDateFormatter *formatter =
    [[NSDateFormatter alloc] init];
[formatter setDateFormat:
    @"yyyy-mm-dd"];
NSString *friendlyDate =
    [formatter stringFromDate:now];
```

TIP See the “Date Formatters” section of the *Data Formatting Programming Guide for Cocoa* in the developer documentation for a complete list of available format strings.

TABLE 1.2 Predefined `NSDateFormatter` Styles

Style	Description
<code>NSDateFormatterNoStyle</code>	The default style if none is specified; no output produced
<code>NSDateFormatterShortStyle</code>	A short, numeric-only style: Date: “07/14/10”; Time: “12:32pm”
<code>NSDateFormatterMediumStyle</code>	An abbreviated style: Date: “Jun 14, 2010”; Time: “12:32pm”
<code>NSDateFormatterLongStyle</code>	A full-text style: Date: “Jun 14, 2010”; Time: “12:32:04pm”
<code>NSDateFormatterFullStyle</code>	The longest style: Date: “Tuesday, June 14, 2010 AD”; Time: “12:32:42pm GMT”

Arrays

Arrays represent an ordered collection of objects.

For creating arrays, you use the **NSArray** class. Similarly to an **NSString**, an **NSArray** is immutable; in other words, once you have created one, you cannot alter its contents. Use the **NSMutableArray** class to create a dynamic array whose contents can be edited after creation.

- To create an array, you can use the following:

```
NSString *string1 = @"one";
NSString *string2 = @"two";
NSString *string3 = @"three";
NSArray *myArray = [NSArray
    → arrayWithObjects:string1,
    → string2, string3, nil];
```

Notice that the list of objects being added to the array is terminated with **nil**.

- You can also create an array from an existing array:

```
NSArray *myArray2 = [NSArray
    → arrayWithArray:myArray1];
```

- You can create an array containing only part of an existing array:

```
NSRange range = NSMakeRange(0,2);
NSArray *subArray = [myArray
    → subarrayWithRange:range];
```

This will create an array containing the first two objects of **myArray**.

- To get the length of an array, use this:

```
int arrayLength = [myArray count];
```

- You can access an object at a particular position in the array:

```
NSString *myString = [myArray  
→ objectAtIndex:0];
```

Since arrays are zero-based, this will return the first object in the array.

- You can see whether an array contains an object:

```
NSString *string1 = @"one";  
NSString *string2 = @"two";  
  
NSArray *myArray =  
→ [[NSArray alloc]  
→ initWithObjects:string1,  
→ string2, nil];
```

```
BOOL isInArray = [myArray  
→ containsObject:string1];
```

- You can get the position of an object in the array:

```
int index = [myArray  
→ indexOfObject: string1];
```

- You can loop through the values of an array:

```
for (NSString *obj in myArray) {  
    NSLog(@"%@", obj);  
}
```

- Or, you can loop backward through the values of an array:

```
for (NSString *obj in [myArray  
→ reverseObjectEnumerator]) {  
    NSLog(@"%@", obj);  
}
```

- You can also sort an array of strings:

```
[myArray sortUsingSelector:  
→ @selector(localizedCase  
→ Insensitive Compare:)];
```

The `@selector` keyword here is the name of the method being used to perform the sort.

An **NSMutableArray** is similar to an **NSArray**, with the added advantage of being able to modify its contents after creation.

- You can add an object to the end of a mutable array:

```
NSString *string1 = @“one”;  
NSString *string2 = @“two”;  
NSMutableArray *myArray =  
    → [[NSMutableArray alloc]  
    → initWithObjects:string1,  
    → string2, nil];  
  
NSString *string3 = @“three”;  
[myArray addObject:string3];
```

or you can add an object at the beginning:

```
[myArray insertObject:string3  
    → atIndex:0];
```

- You can replace an object at a particular position in the array:

```
[myArray replaceObjectAtIndex:0  
    → withObject:string2];
```

- You can remove an object from an array:

```
[myArray removeObject:string3];
```

- Or, you can remove an object at a particular position in the array:

```
[myArray removeObjectAtIndex:0];
```

This will remove the first object.

- To remove several objects from an array, you can use this:

```
NSRange range = NSMakeRange(0,2);  
  
[myArray removeObjectsInRange:  
    → range];
```

This will remove the first two objects.

- Finally, you can remove all the objects in the array:

```
[myArray removeAllObjects];
```

Dictionaries

An **NSDictionary** is used to store associated key-value pairs. Again, like **NSArray**, **NSDictionary** is immutable. Use the **NSMutableDictionary** class if you need to be able to alter the contents of the dictionary after creation.

Generally, each key-value pair (called an *entry*) consists of an **NSString** for the key and an **NSObject** for the value. Keys must be unique within a dictionary; values do not need to be.

NSDictionarys are used extensively throughout the Cocoa frameworks because they provide an efficient and simple way of storing information in an easily retrievable manner. For example, user defaults are stored as dictionaries, and notifications often have a **userInfo** dictionary containing additional information about the notification.

- To create a dictionary, you can use the following:

```
NSArray *arr1 = [NSArray  
→ arrayWith Objects:@"iPhone",  
→ @"iPod",nil];  
  
NSArray *arr2 = [NSArray  
→ arrayWith Objects:@"iMac",  
→ @"Mac Pro", @"Macbook",  
→ @"Macbook Pro",nil];  
  
NSDictionary *myDict =  
→ [[NSDictionary alloc]  
→ dictionaryWithObjectsAndKeys:  
→ arr1, @"mobile", arr2,  
→ @"computers", nil];
```

Notice that like an **NSArray**, you end the list of objects with **nil** when creating the dictionary.

continues on next page

- You can see how many elements are in an dictionary:

```
int dictSize = [myDictionary  
→ count];
```

- You can access an object in the dictionary:

```
NSArray *mobile = [myDict  
→ objectForKey:@"mobile"];
```

- Or, to retrieve the keys for an object, use the following:

```
NSArray *keys = [myDict  
→ allKeysForObject:arr1];
```

- To retrieve an array of all the values in the dictionary, use this:

```
NSArray *values = [myDict  
→ allValues];
```

- You can also enumerate through the contents just like an **NSArray**:

```
for (id key in myDict) {  
  
    NSLog(@"key: %@", value:  
          → key, [myDict objectForKey:  
          → key]);  
  
}
```

- If your dictionary contains only property list objects (**NSData**, **NSDate**, **NSNumber**, **NSString**, **NSArray**, or **NSDictionary**), you can save it to a file:

```
NSString *filePath = [[[NSBundle  
→ mainBundle] resourcePath]  
→ stringByAppendingPathComponent:  
→ @"dict.txt"];  
  
BOOL success = [myDict  
→ writeToFile: filePath  
→ atomically:YES];
```

- Conversely, you can populate a dictionary from a file:

```
NSDictionary *myDict2 =  
→ [NSDictionary dictionaryWithContentsOfFile:filePath];
```

By using an **NSMutableDictionary**, you can add and remove objects after creation.

- To add an object to a mutable dictionary, you can use this:

```
NSArray *arr1 = [NSArray  
→ arrayWithObjects:@"iPhone",  
→ @"iPod",nil];  
  
NSArray *arr2 = [NSArray  
→ arrayWithObjects:@"iMac",  
→ @"Mac Pro", @"Macbook",  
→ @"Macbook Pro",nil];  
  
NSMutableDictionary *myDict =  
→ [[NSMutableDictionary alloc]  
→ initWithObjectsAndKeys:arr1,  
→ @"mobile", arr2, @"computers",  
→ nil];  
  
NSString *string1 = @"AppleTV";  
  
[myDict setObject:string1  
→ forKey:@"media"];
```

- You can also alter an existing object in the dictionary:

```
NSString @string2 = @"airport  
→ express";  
  
[myDict setObject:string2  
→ forKey:@"media"];
```

- Or, you can remove an object from the dictionary:

```
[myDict removeObjectForKey:  
→ @"media"];
```

- To remove multiple objects, use the following:

```
NSArray *keyArray = [NSArray  
→ arrayWithObjects:@"mobile",  
→ @"computers",nil];  
  
[myDict removeObjectsForKeys:  
→ keyArray];
```

- Finally, to remove all objects from the dictionary, you can use the following:

```
[myDict removeAllObjects];
```

Notifications

Notifications provide a handy way for you to pass information between objects in your application without needing a direct reference between them. They are represented by the **NSNotification** object, which contains a name, an object (often the object posting the notification), and an optional dictionary.

Notifications are posted to a *notification center* whose job is to forward the notification to all *registered observers*—objects that have requested to be told about certain notifications.

- To register your object as an observer for a notification, you can write the following:

```
[[NSNotificationCenter
    → defaultCenter] addObserver:self
    → selector:@selector(doSomething:)
    → name:@"myNotification"
    → object:nil];
```

- Here you are observing a notification called “**myNotification**.”

When the notification is sent to your object (by the notification center), a message is sent to the **doSomething:** method. By passing **nil** to the final parameter, you are saying that you want to be told about any notification named **myNotification**, no matter which object sends it. You could have optionally specified which object you wanted to observe, which might be useful in the situation where multiple notifications have the same name.

- To post your own notification to send the dictionary discussed earlier, you can use this:

```
[[NSNotificationCenter
    → defaultCenter]
    → postNotification
    → Name:MY_NOTIFICATION
    → object:myDict];
```

- You can then access the dictionary being sent:

```
- (void)doSomething:(NSNotification * )aNote {  
    NSDictionary *myDict = [aNote  
                           objectForKey];
```

Code Listing 1.7 shows an example of both posting and receiving a notification. Note that in this example both the sending and receiving objects are in fact the same, which wouldn't normally be the case.

TIP Notice how the notification name has been defined as a variable (MY_NOTIFICATION). This is a good practice because the compiler will report a compile-time error if you mistype the notification name.

TIP Remember to stop observing notifications by using `removeObserver`: when you are finished with the notification. As in the previous example, this is often done in the `dealloc` method.

Code Listing 1.7 Registering for, posting, and receiving a notification.

```
#define MY_NOTIFICATION @"MY_NOTIFICATION"  
  
- (void)doSomething:(NSNotification *)aNote {  
    NSDictionary *myDict = [aNote objectForKey];  
    NSLog(@"%@", myDict);  
}  
  
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    NSString *myString = @"some value";  
    NSDictionary *myDict = [[NSDictionary alloc] initWithObjectsAndKeys:myString,@"firstObject",nil];  
  
    [[NSNotificationCenter defaultCenter] addObserver:self  
                                             selector:@selector(doSomething:)  
                                               name:MY_NOTIFICATION  
                                             object:nil];  
  
    [[NSNotificationCenter defaultCenter] postNotificationName:MY_NOTIFICATION object:myDict];  
}  
  
- (void)dealloc {  
    [[NSNotificationCenter defaultCenter] removeObserver:self];  
    [super dealloc];  
}
```

Timers

Another common task is to have some code run based on some type of timing function. For example, you might be implementing a clock and want the display to update every minute, or you may want to present a message to a user and have it disappear after a certain amount of time.

You can use the **NSTimer** class to add this type of functionality to your applications. Timers allow you to execute a piece of code after a given amount of time.

- The simplest way to create a timer is by using the class method:

```
NSTimer *myTimer = [NSTimer  
→ scheduledTimerWithTime  
→ Interval:10.0 target:self  
→ selector:@selector  
→ (myTimerAction:)  
→ userInfo:nil repeats:NO];
```

This will create a timer that calls the **myTimerAction:** method in ten seconds from now. Notice the **userInfo** parameter: This lets you pass any object you like to your timer method. Passing **NO** to the **repeats** parameter means your timer method will be called only once. If you pass **YES**, the timer would keep repeating every ten seconds.

- You then implement your `myTimerAction:` method:

```
- (void)myTimerAction:(NSTimer *)  
    → timer  
{  
    NSLog(@"timer fired!: %@", [timer  
        → userInfo]);  
}
```

The timer passes itself as a parameter to the method. Notice how you can get the `userInfo` object you added when creating the time (in this example, `nil`).

- To stop a timer, you call the following:

```
[myTimer invalidate];
```

- You can also create a timer that you don't actually want to run until later:

```
myTimer = [[NSTimer timerWith  
    → TimeInterval:10.0 target:self  
    → selector:@selector  
    → (myTimerAction:) userInfo:nil  
    → repeats:NO] retain];
```

- Then, when you are ready to start the timer, you use the following:

```
[[NSRunLoop mainRunLoop]  
    → addTimer:myTimer forMode:  
    → NSDefaultRunLoopMode];
```

continues on next page

- Another less common way of creating a timer is by using the following:

```
NSTimeInterval *secondsPerDay =  
→ 24*60*60;  
  
NSDate *tomorrow = [NSDate  
→ dateWithTimeIntervalSinceNow:  
→ secondsPerDay];  
  
myTimer = [[NSTimer alloc]  
→ initWithFireDate:tomorrow  
→ interval:10.0 target:self  
→ selector:@selector  
→ (myTimerAction:) userInfo:nil  
→ repeats:YES];
```

This creates a timer to run tomorrow that will repeat every ten seconds.

Again, you need to call **addTimer:forMode:** to actually start the timer; however, this time you have created a timer that first fires at an exact point in time.

Code Listing 1.8 shows an example of working with a timer.

Code Listing 1.8 Using timers.

```
- (void)myTimerAction:(NSTimer *)timer  
{  
    NSLog(@"timer fired!: %@",[timer userInfo]);  
}  
  
- (void)viewDidLoad {  
  
    NSTimeInterval secondsPerDay = 24*60*60;  
    NSDate *tomorrow = [NSDate dateWithTimeIntervalSinceNow:secondsPerDay];  
    myTimer = [[NSTimer alloc] initWithFireDate:tomorrow  
                                interval:10.0  
                                  target:self  
                                selector:@selector(myTimerAction:)  
                                userInfo:nil repeats:NO];  
    [[NSRunLoop mainRunLoop] addTimer:myTimer forMode:NSDefaultRunLoopMode];  
  
    myTimer2 = [NSTimer timerWithTimeInterval:1.0  
                                    target:self  
                                  selector:@selector(myTimerAction:)  
                                    userInfo:nil repeats:NO];  
}
```

Design Patterns

When writing applications, you may often find yourself building the same functionality or encountering similar design problems time and time again. *Design patterns* offer a solution to this situation by providing a set of general, reusable, tested solutions to common programming scenarios.

Design patterns are used extensively throughout the iOS frameworks. If you write an iOS application, you are using them whether or not you know it. The following are some of the more frequently used design patterns in iOS development.

Model View Controller

The Model View Controller (MVC) pattern separates an application's data structures (the *model*) from its user interface (the *view*), with a middle layer (the *controller*) providing the "glue" between the two. The controller takes input from the user (via the view), determines what action needs to be performed, and passes this to the model for processing. The controller can also act the other way: passing information from the model to the view to update the user interface.

Breaking your application into distinct components like this reduces dependencies and increases the reusability of the code. It's entirely possible that you could reuse the same model across multiple applications with different views and controllers.

MVC is used everywhere in the iOS frameworks. You will notice many classes with the word *View* or *Controller* in their names, and almost all the project templates create an MVC-based project.

Delegate

The Delegate pattern is useful as an alternative to subclassing, allowing an object to define a method but assign responsibility for *implementing* that method to a different object (referred to as the *delegate object* or, more commonly, the *delegate*).

Delegates need not implement all (or even any) of the delegate methods for the source object. In that case, the source object's default behavior for the method is often used.

Code Listing 1.9 shows an example of a delegate used for a **UITextField** object. By implementing the **textFieldShouldBeginEditing** delegate method (and in this example, returning **NO**), you can control the appearance of the keyboard.

As with MVC, delegates are frequently used throughout the iOS frameworks, and you will use them often in your application development.

Code Listing 1.9 Using a delegate.

```
- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField
{
    return NO;
}

- (void)viewDidLoad {
    CGRect rect = CGRectMake(10,10,100,44);

    UITextField *myTextField = [[UITextField alloc] initWithFrame:rect];
    myTextField.delegate = self;

    [self.view addSubview:myTextField];
    [myTextField release];
}
```

Target-Action

This pattern is used by most of the controls in the iPhone user interface. When creating a control, you assign the *target* object to send a message to, and you supply the method in that object to call when a particular *action* (for example, tapping a button) occurs. It is your responsibility as the developer to implement the methods for these actions.

Code Listing 1.10 shows an example of using the Target-Action pattern to handle a **UIButton**.

Categories

Like delegates, categories provide an alternative to subclassing, allowing you to add new methods to an existing class. The methods then become part of the class definition and are available to all instances (and subclasses) of that class.

Code Listing 1.10 Buttons use the target-action pattern.

```
- (void)buttonTap:(id)sender
{
    NSLog(@"Button tapped");
}

- (void)viewDidLoad {
    CGRect rect = CGRectMake(10,10,100,44);
    UIButton *myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [myButton setFrame:rect];
    [myButton addTarget:self
                  action:@selector(buttonTap:)
            forControlEvents:UIControlEventTouchUpInside];

    [self.view addSubview:myButton];
}
```

Code Listing 1.11 and **Code Listing 1.12** show an example category that adds a new method to the **UIImage** class. Notice how in both the interface and implementation definitions you use the class name, with the category name in parentheses.

Although you can name your category header and implementation files anything you like, a common convention is to use the original class name with **+xxxx** appended. The **xxx** describes the additional functionality provided by the category (or, as in the example here, is a generic term indicating additional functionality).

Code Listing 1.12 The category implementation.

```
#import "UIImage+Additions.h"

@implementation UIImage (Additions)

+(UIImage *)newImageFromResource:(NSString *)filename
{
    NSString *imageFile = [[NSString alloc]
                           initWithFormat:@"%@/%@",
                           [[NSBundle mainBundle] resourcePath],
                           filename];
    UIImage *image = [[UIImage alloc]
                      initWithContentsOfFile:imageFile];
    [imageFile release];
    return [image autorelease];
}

@end
```

Code Listing 1.11 The category header.

```
#import <UIKit/UIKit.h>

@interface UIImage (Additions)

+(UIImage *)newImageFromResource:(NSString *)filename;
@end
```

Singletons

You can think of a *singletont* as a global object. There will only ever be one, and it is available to all the classes in your application. Although this pattern is not frequently used, you will use several important singletons when developing iOS applications such as **UIApplication** and **UIDevice**.

Code Listing 1.13 shows an example of using the **UIDevice** singleton to check the battery level of an iPhone.

TIP For more information on the design patterns used in iOS development, refer to the “Cocoa Design Patterns” section in the **Cocoa Fundamentals Guide** in the developer documentation.

Code Listing 1.13 Using the **UIDevice** singleton.

```
- (void)viewDidLoad {
    float level = [[UIDevice currentDevice] batteryLevel];
    bool batteryState = [[UIDevice currentDevice] batteryState];
    if (batteryState && level < 0.1)
        NSLog(@"battery almost empty! (%f)",level);
}
```

This page intentionally left blank

2

The iPhone Developer's Toolbox

Before you can begin building your own applications for the iPhone, you'll need to register with Apple as a developer and download the iOS software development kit (SDK). Luckily, both of these steps are easy and completely free—simply visit <http://developer.apple.com>. You'll also find a wealth of information regarding iPhone development, including documentation, tutorials, videos, sample code, and more.

The iOS SDK includes the Xcode integrated development environment (IDE), iPhone Simulator, Interface Builder, documentation, and a number of other tools you'll need when developing iPhone applications.

In This Chapter

About the Xcode IDE	42
About the iPhone Simulator	61
About Interface Builder	64
About the Documentation	78
The Xcode Organizer	79

About the Xcode IDE

This chapter presents a whirlwind tour of some of the tools you'll use on a daily basis as an iPhone developer. You can find more information on the Apple Developer Connection Web site (<http://developer.apple.com>).

Xcode is part of a suite of development tools used for creating iPhone, iPod touch, iPad, and OS X applications. Among others, there are tools for user-interface design, source-code editing and management, integrated debugging, and performance analysis.

When you first launch Xcode, you are presented with a welcome screen **A**. This screen contains lots of useful information such as links to videos, sample code, RSS feeds, tips and tricks, and documentation. It's well worth your time to go through the tutorials and code and subscribe to the mailing lists and RSS feeds. If you have a question, it's very unlikely you are the first to ask, and the searchable mailing list archives are an excellent place to look for an answer.

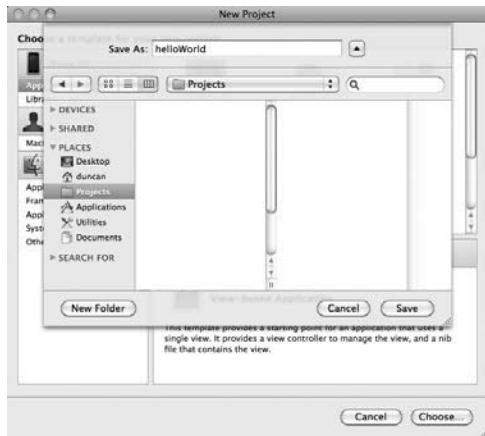
If you close this window and choose File > New Project, you'll see the Xcode New Project window where you can choose a template for your project **B**. As an iPhone developer, you are interested only in the iPhone OS section of the list.



A The Xcode welcome screen.



B Choosing a project template.



C Saving your project.

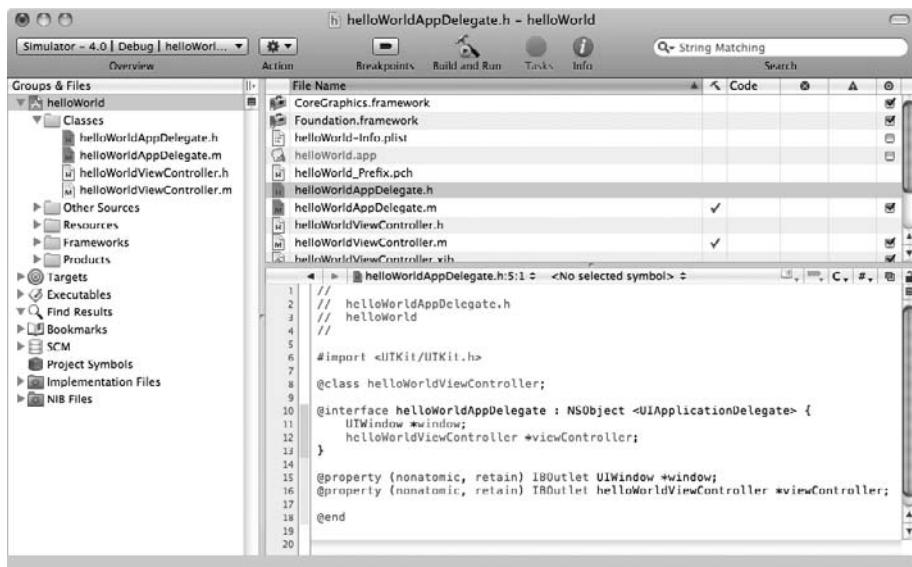
When you select a template type, you'll see a short summary of its use. Xcode automatically creates boilerplate interface (.xib) and code (.h/.m) files for each template. Again, you should spend some time going through each of these templates and familiarizing yourself with them. Most of the examples in this book use the View-based Application template, which is a good one for general use.

Now you'll create a new project and walk through the main parts of the Xcode interface.

To create a new project:

1. Start Xcode.
2. Choose File > New Project.
3. Select the View-based Application template, and save your project as helloWorld C.

Xcode now displays your project in a single window with a number of panes D.



D The main Xcode window.

About the Groups & Files pane

The Groups & Files pane shows the contents and configuration of your project, categorized into either static or smart groups **E**.

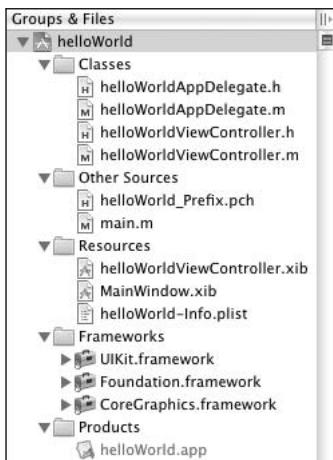
Static groups provide a convenient way for you to categorize files and folders. They are used only in the context of an Xcode project, and they do not necessarily reflect any physical folder structure in the file system.

You can create as many static groups as you like, and they can be nested within each other.

By default, Xcode creates the groups described in **Table 2.1**.

The `<app>-Info.plist` file contains configuration settings about your application. **Table 2.2** lists some of the settings automatically created with your application.

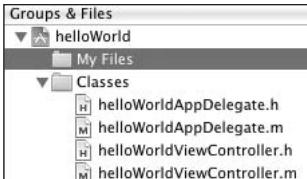
As a general rule, most developers create their own groups for their code and use the existing Resources folder for files such as graphics, but you can place your code and



E The Xcode Groups & Files pane.

TABLE 2.1 Static groups

Group	Description
Classes	Contains the header and implementation files for the classes that are automatically generated for the application type that you selected when you created your project.
Other Sources	Contains the prefix header file used by Xcode to reduce compile time and contains the main.m file that is the entry point to your application. (It's generally advised that you not touch either of these files unless you know what you are doing.)
Resources	Contains the XIB files containing your application's user interface and contains the <code><app>-Info.plist</code> file, which specifies your application's settings.
Frameworks	Contains the frameworks used by your application (these may differ based on the template type you chose when creating your project).
Products	Contains the compiled binary representing your application. This is the file that is installed on the simulator and uploaded to Apple for sale in the iTunes App Store. You may notice that if you've never compiled your application or you've never performed a clean (from the Build toolbar), this file will appear with red text. This is how Xcode displays missing files.



F Adding a new group.



G The default smart groups.

other project files in any of these groups, move them around, rename or delete them, or create your own. The groups are simply there to make your life as a developer easier when managing your application's code and assets.

To create a new group:

1. Right-click the topmost item in the Groups & Files pane (in this case `helloWorld`), and choose Add > New Group.
2. Name your group `My Files` **F**.

Smart groups are the second type of group **G**. Again, Xcode automatically creates a number of these groups to categorize various types of information (**Table 2.3**).

TABLE 2.2 <app>-Info.plist properties

Property	Description
<code>CFBundleDisplayName</code>	The name your app shows on the iPhone.
<code>CFBundleIconFile</code>	The PNG file containing your application icon.
<code>CFBundleIdentifier</code>	The identifier you set up on the iPhone Developer Program Portal section of the Developer Connection Web site to uniquely identify your application.
<code>CFBundleVersion</code>	The current version of your application. You will update this each time you submit your application to the App Store.
<code>LSRequiresiPhoneOS</code>	Set to <code>true</code> if your application can only run on iPhone OS
<code>NSMainNibFile</code>	The initial user interface XIB file, loaded by the call to <code>UIApplicationMain()</code> in the main.m file.

TABLE 2.3 Smart groups

Group	Description
Targets	You will have an entry here for each of your project targets (defined in the next section).
Executables	Contains all of the executables for your project.
Errors and Warnings	Each error or warning your application generates on compile will be listed here.
Find Results	This shows the history of any projectwide searches you have performed.
Bookmarks	This shows all bookmarks for your project.
SCM	If you are using source-code management, this group shows you the status of any files that may have changed under source control. Currently Xcode has support for the CVS, Perforce, and SVN source control systems.

Targets

A *target* is a set of instructions for building a product. These instructions might include such information as which source files to compile or resources to copy, as well as any special instructions for processing them. In most cases, you will be working only with the default target created with your project.

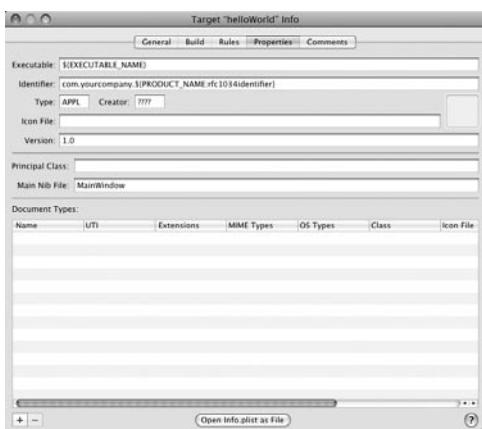
If you expand the `helloWorld` target **H**, you'll see folders representing the main steps Xcode will perform when building the sample application: copying files into the application bundle, compiling the source code, and linking against the frameworks in the project.

Right-click the `helloWorld` target, and choose Get Info (Command+I) to open the Target Info dialog box **I**. It has five tabs.

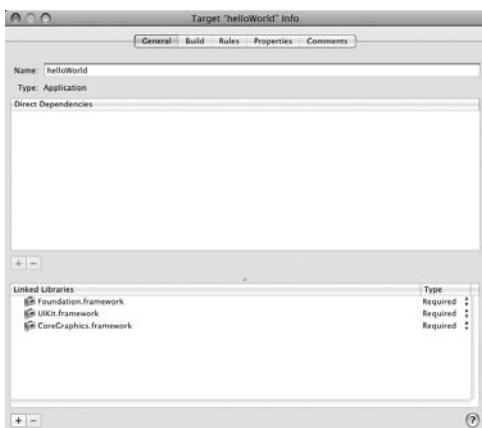
- **General**—Shows you the libraries and frameworks currently linked against your project, as well as any other dependencies (such as external projects) you may have. You can add other frameworks or linked libraries (such as SQLite) using the Add (+) button in the bottom-left corner **J**.
- **Build**—Shows you the settings Xcode will use when building your product **K**. They are grouped into configurations, providing a handy way to apply a range of settings without having to create multiple targets.



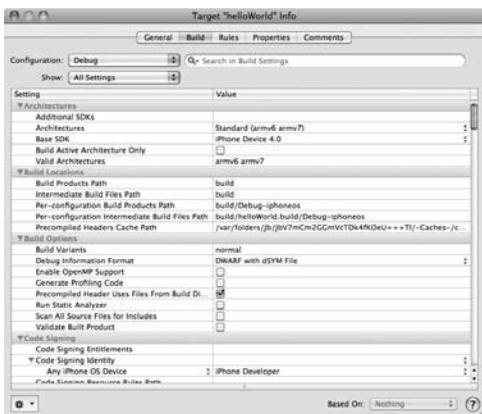
H The project target group, expanded.



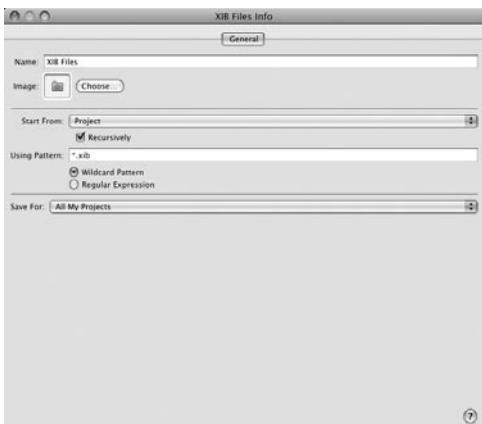
I The Target Info dialog box.



J The project's General tab.



K Build settings for project.



L Settings for the XIB Files smart group.

By default, Xcode creates Debug and Release configurations. The *Debug* configuration is what you will generally use in your day-to-day development, allowing you to interact with the application using the debugger, among other things. When you are ready to deploy your application to the iTunes App Store, you would build using the *Release* configuration, creating an optimized, production-ready version of your product.

- **Rules**—Specifies how particular file types are processed when your application is compiled. For example, you could pick a certain compiler version to compile your source files.
- **Properties**—Shows the contents of the <app>-Info.plist file in a slightly more human-readable form than simply viewing the file itself.
- **Comments**—Lets you add your own comments for the project. You can write anything you want here; comments will be stored with the project.

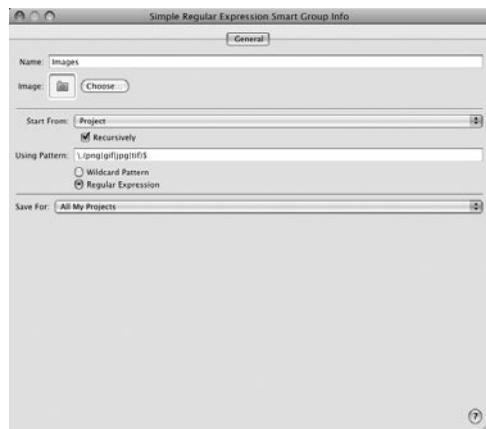
You can find a lot more information on targets and the build system in the *Xcode Build System Guide*.

You may also notice the last two smart groups in the Groups & Files pane, Implementation Files and XIB Files, have the same icon. These are special groups similar to Smart Folders in OS X. If you Control+click the XIB Files folder and choose Get Info (Command+I), you'll be doing a simple match for all files that have the extension .xib L.

To create a smart group:

1. Choose Project > New Smart Group > Simple Regular Expression Smart Group.
2. Change the name to Images and the Using Pattern value to `\.(png|gif|jpg|tif)$`.

You should now see this group at the bottom of your Groups & Files pane **M**. Any image files you add to your project will automatically be shown here.



M The completed smart group settings.



N The Overview menu showing the current build settings for the application.

About the toolbar

When you first launch Xcode, the default toolbar features the following options:

- **Overview**—Shows you the current build settings for your application. This tells you whether you are currently building for the simulator or an actual iPhone, along with which build configuration you are using **N**.
- **Action**—Provides a shortcut to a number of often-used actions.
- **Build and Go**—Builds your application and installs and launches it on either the iPhone Simulator or the iPhone (depending on what you've chosen in the Active SDK dialog box, which is accessible via the Overview toolbar menu).
- **Tasks**—Allows you to cancel a build in progress.
- **Info**—Shows additional information for any item selected in either the Groups & Files pane or the details pane.
- **Search**—Allows you to filter the list of items currently shown in the details pane.

When you first launch Xcode, the default toolbar is adequate **O**. However, there are a couple more items you can add to make it a little more useful. You'll use them later in this chapter.

To update the toolbar:

1. Control+click the toolbar, and choose Customize Toolbar.
2. Drag the Build toolbar item that has the disclosure triangle, and drop it onto the toolbar.
3. In the same way, drag the Debugger and Editor toolbar items, and click Done to close the Customize Toolbar window. Your toolbar is now updated **P**.

About the details pane

The details pane shows a list of files, based on what is currently selected in the Groups & Files pane. For example, if you select multiple group, you'll see all files for the groups **Q**.



O The default Xcode toolbar.



P The updated toolbar.

A screenshot of the Xcode interface showing the Groups & Files pane on the left and the Details pane on the right. The Groups & Files pane displays a project structure with groups like "helloWorld", "My Files", "Classes", and "Other Sources". The Details pane lists files with checkboxes next to them, showing which files are selected. The listed files are: helloWorld_Prefix.pch, helloWorldAppDelegate.h, helloWorldAppDelegate.m, helloWorldViewController.h, helloWorldViewController.m, and main.m. All files have checkboxes checked in the first column of the Details pane.

Q Selecting multiple groups in the Groups & Files pane will show all files in the details pane.

Column headers in the details pane show information such as filename, file size, build status, errors, and warnings. When the helloWorld project has a build error, Xcode highlights the error in the editor pane **R**.

Selecting a file in the details pane opens the file in the editor pane below. Double-clicking or pressing Enter when a file is selected causes it to open in its own editor window.

Depending on the file type, it may display as an image or as a property list **S**. Read-only files such as frameworks do not show in the editor pane at all.

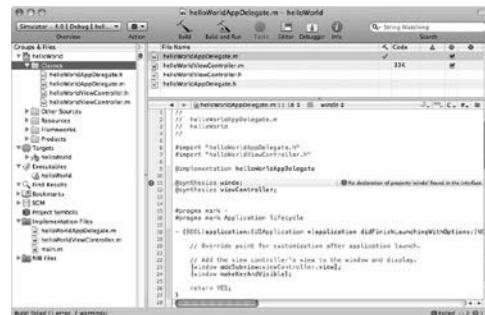
To see more details about a file, such as its physical location, click the Info toolbar button (or press Command+I).

TIP When you select a property list file, it defaults to displaying as a table of key-value pairs. Although this is useful most of the time, you sometimes want to edit the raw source of the file. You can accomplish this by Control+clicking the property list file in the details pane and choosing Open As.

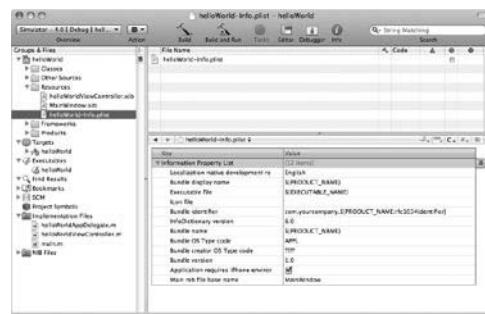
About the editor pane

Xcode contains all the features you would expect from a modern editor, such as syntax highlighting, code completion, smart indenting, code folding, multidocument support, unlimited undo and redo, and more.

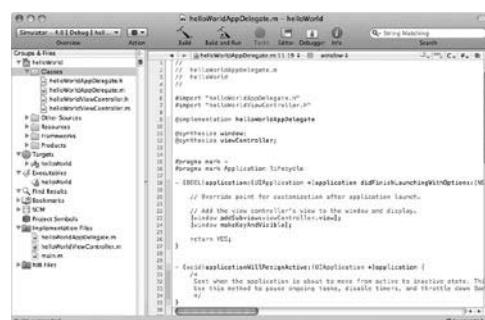
You can expand the amount of space you have for editing code by clicking the Editor toolbar icon you added earlier (or by pressing Shift+Command+E). This causes the details pane to collapse and provides the full window height for the editor **T**.



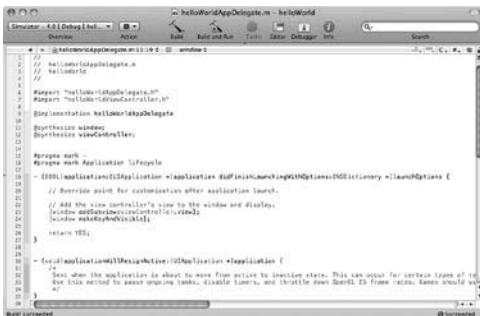
R Xcode shows detailed information about errors in code.



S Xcode displays property lists as a table of keys and values.



T Collapse the details pane to increase the area for your code.



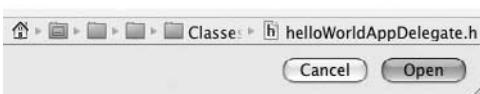
```
/* helloWorldAppDelegate.m - helloWorld */
// helloWorldAppDelegate.h
// helloWorldAppDelegate.m
// #include <Foundation/Foundation.h>
// #include "helloWorldAppDelegate.h"
// Implementation helloWorldAppDelegate
// By Thomas Wieden
// http://www.cocoadev.com/wiki/helloWorld

// Program main
// MyApplication's application lifecycle
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    // Add the view controller's view to the window and display.
    [window addSubview:[viewController view]];
    [window makeKeyAndVisible];
}
return YES;
}

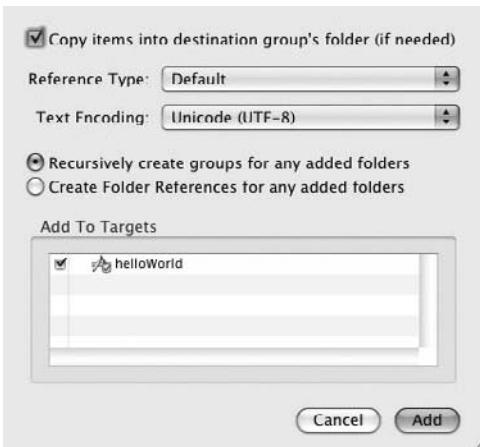
// Local application methods
- (void)applicationWillResignActive:(UIApplication *)application {
    // This is called when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.
    // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method to pause the game.
}

```

⑪ The editor pane with all other panes collapsed.



⑫ Quickly find and open files in your projects.



⑬ Adding external files to your projects.

Taking this further, if you also hold down the Option key (Option+Shift+Command+E), the Groups & Files pane collapses as well ⑭.

Finally, pressing Option+Command+O causes the current file to open in its own window.

To quickly open files in your project from the keyboard, you can use the Open Quickly dialog box (Shift+Command+D), which lets you start typing to find the filename. Positioning your pointer on a file, variable, or method before invoking this dialog box will default to showing the file containing that variable, method, or name ⑮.

When creating new files in Xcode, the default behavior is to put the file in whichever static group is currently selected. If no group is selected, the file is added at the top level of the list—you can of course move your files around within groups at any time after you've added them to the project.

When adding existing files to your projects, you will have the option to copy the items into your project's folder and choose what reference type to use ⑯. It's advisable to always select this option and to use the default reference type. If you were to move your project to a different location or machine at a later date, the file might not exist in the same location. In this situation, Xcode would indicate this by displaying the file in red.

Gutter and focus ribbon

The *gutter* on the left shows line numbers, breakpoints, errors, and warnings (you may have to enable line numbers in the Xcode preferences) . Clicking in the gutter will add a breakpoint at the current line. Clicking the breakpoint will change it to a light blue color, indicating it has been disabled. To remove the breakpoint, you can either drag the breakpoint off the ribbon or Control+click and choose “Remove breakpoint” (you can also press Command+\ to toggle breakpoints).

Next to the gutter is the *focus ribbon*

Moving your pointer over a shaded area in the ribbon causes the code for that scope to be highlighted and the rest of your code to be dimmed .

Clicking in the shaded scope area of the focus ribbon makes your code fold and unfold, replacing your code with a small graphic ellipsis symbol ⑦. Double-clicking the icon unfolds the code.

This is a great way for you to hide code you'll never read (such as comments at the header of a file) or that you may not change often.

TIP You can also control code folding from the keyboard: Pressing Control+Command+left arrow or Control+Command+right arrow causes the current scope to fold or unfold. Pressing Control+Command+up arrow or Control+Command+down arrow causes all methods in the current file to fold or unfold.

TIP If you like the scope highlighting effect, you can enable it all the time by choosing Focus Follows Selection from the View > Code-Folding menu. This is a great way to see your code in functional groups.

```
1 // > helloWorldAppDelegate.m:8.1 < No selected symbol > □ C ⌘ ⌂ ⌄
2 // helloWorldAppDelegate.h
3 // helloWorld
4 //
5
6 #import "helloWorldAppDelegate.h"
7 #import "helloWorldViewController.h"
8
9 @implementation helloWorldAppDelegate
10
11 @synthesize window;
12 @synthesize viewController;
13
14
15 - (void)applicationDidFinishLaunching:(UIApplication *)application {
16
17     // Override point for customization after app launch
18     [window addSubview:viewController.view];
19     [window makeKeyAndVisible];
20
21
22 }
```

X The Xcode editor pane with a breakpoint set.

```
8 // Import the Objective-C runtime header.
9 #import <objc/runtime.h>
10
11 @implementation HelloWorldAppDelegate
12
13 @synthesize window;
14 @synthesize viewController;
15
16 - (void)applicationDidFinishLaunching:(UIApplication *)application {
17
18     // Override point for customization after app launch
19     [window addSubview:viewController.view];
20     [window makeKeyAndVisible];
21 }
22
23
24 - (void)dealloc {
25     [viewController release];
26     [window release];
27     [super dealloc];
28 }
29
30 }
```

Y Highlighting scope makes code more readable.

```
4 <-- @autoreleasepoolController.m:1 --> <No selected symbols>
5
6 #import "HelloWorldViewController.h"
7
8 @implementation HelloWorldViewController
9
10
11
12 // MARK: -
13
14 // MARK: -
15
16 // MARK: -
17
18 // MARK: -
19
20 // MARK: -
21
22 // MARK: -
23
24 // MARK: -
25
26 // MARK: -
27
28 // MARK: -
29
30 // MARK: -
31
32 // MARK: -
33
34
35
36
37 // MARK: -
38
39
40
41 - (void).didReceiveMemoryWarning:(MemoryWarning *)memoryWarning {
42     // Releases the view if it didn't have a superview.
43     [super.didReceiveMemoryWarning(memoryWarning)];
44
45     // Release any cached data, images, etc that aren't in use.
46 }
47
```

Z Collapsed code is represented by an ellipsis in the editor window.



AA Searching for text within the current file.



BB Searching within the entire project.

Find-and-replace operations

Pressing Command+F opens the Single File Find dialog box AA, allowing you to perform find-and-replace operations within the current file. Pressing Command+G and Shift+Command+G navigates forward and backward through matches.

Pressing Shift+Command+F brings up the more powerful Project Find window BB, which allows you to search through all files within your project and even into frameworks and other projects. Remember that Project Find history is also available in the Find Results smart group in the Groups & Files pane.

Bookmarks

Often you'll want to return to a particular line within your source code. Pressing Command+D allows you to bookmark a location. You can then access those bookmarks from the Bookmarks pop-up menu as well as the Bookmarks smart group in the Groups & Files pane.

Jump-to-definition and help

If you Command+double-click any symbol, variable, or method within your code, Xcode opens the relevant file and highlights the definition of that symbol. If more than one definition exists, you'll be presented with a pop-up menu of all the files containing the symbol, allowing you to select which to open. This is extremely useful when browsing the Cocoa header files because you can quickly, for example, jump to the header definition of a class to see its properties and methods.

Similarly, if you Option+double-click, Xcode opens the documentation for that symbol. Again, this is a very quick and easy way of looking up how to use a particular function, class, or method.

TIP You can quickly comment or uncomment your code by highlighting a block and pressing Command+/.

Code completion

Objective-C often has very long method names, and the Cocoa frameworks are huge, making it very difficult to remember what to type. This is where code completion comes in: Begin typing a method name, and Xcode will suggest the method it thinks you want based on the context of your code **CC**.

The more you type, the more accurate Xcode's match will become. When the right choice appears, press Tab to have Xcode complete the code for you.

Optionally, if you press the Esc (Escape) key, you will be presented with a list of all matches for the current completion **DD**. You can then navigate the list with the up and down arrow keys and press Enter to choose a match.

If your completion has multiple parameters, Xcode will automatically highlight the first parameter. Once you enter a parameter, you can press Control+/ to move to the next one. Similarly, Shift+Control+/ moves you backward in the parameter list.

Code completion is enabled in the preferences menu, or you can manually invoke it by pressing the Esc key. You can even position the pointer over some existing, already completed code and then press Esc to see a list of other possible completions.

The button at the bottom right of the completion list allows you to toggle the sorting of the list between alphabetical and “best-guess” results **EE**.

```
15 - (void)applicationDidFinishLaunching:(UIApplication *)application {
16 // Override point for customization after app launch
17 [window addSubview:viewController.view];
18 [window makeKeyAndVisible];
19 [window convertRect:(CGRect)rect toWindow:(UIWindow *)window]
20 }
21 }
```

CC Code completion shows the parameters of your methods as you type.



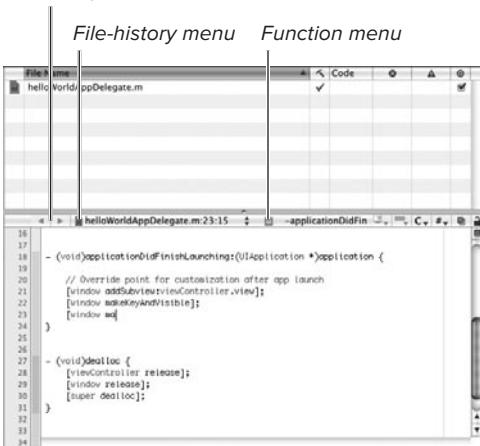
DD Xcode presents multiple completion matches as a list.

```
16 - (void)applicationDidFinishLaunching:(UIApplication *)application {
17 // Override point for customization after app launch
18 [window addSubview:viewController.view];
19 [window makeKeyAndVisible];
20 [window convertRect:(CGRect)rect toWindow:(UIWindow *)window]
21 }
22
23
24
25
26
27
28
29
30
31 }
```

A screenshot of the Xcode editor showing the same code as above, but with the completion list expanded. The list now contains two entries: 'makeKeyAndVisible' and 'makeKeyWindow'. Both entries are preceded by a small icon representing the type of completion: a blue square for 'makeKeyAndVisible' and a yellow square for 'makeKeyWindow'. The list is sorted alphabetically.

EE Toggle the way completions are presented.

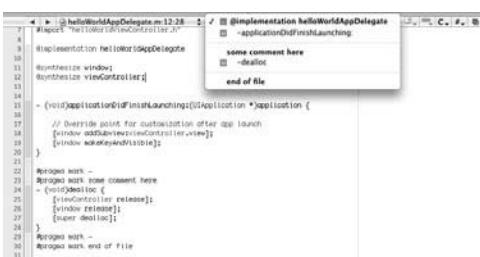
Navigation arrows



FF Arrows in the navigation bar and the file-history menu let you navigate through the editor pane's history.



GG You can enhance the function menu to show developer comments.



HH Adding breaks to the function menu makes it easier to read.

About the navigation bar

Above the content pane is the navigation bar **FF**.

The arrows on the left of the bar allow you to navigate forward and backward through your editor pane history. You can also get at the same information by clicking the file-history menu (Control+1 on the keyboard).

The function menu (Control+2 on the keyboard) shows a summary of all the class, function, method, and type declarations and definitions for the current file. Selecting one will jump to that point in your code.

There are also a number of special keywords you can put in your code to create marker labels in this menu. For example, putting the following before the **dealloc** method of a class

#pragma mark some comment here

would result in the function menu looking something like **GG**.

You can also break up the menu by adding separator labels, such as

#pragma mark -

which results in something like **HH**.

There are also a number of special comment types:

//???: a question

//!!!!: something important

//MARK: this is a mark

//TODO: a to-do item

//FIXME: something needs fixing

Now the code snippet contains all the special identifiers and the resulting menu **II**:

The right side of the navigation bar contains a number of useful pop-up menus and buttons **JJ**:

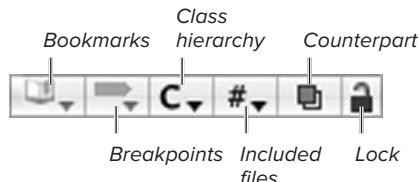
- **Bookmarks**—Gives you access to *bookmarks* for the current file (you can also press Control+4 on the keyboard). Recall that bookmarks for your entire project are visible in the Groups & Files pane.
- **Breakpoints**—Allows you to jump between any breakpoints set in the current file.
- **Class hierarchy**—Lists all the superclasses and subclasses of your current file.
- **Included files**—Lists all the files that the current file includes or is included in.
- **Counterpart**—Allows you to jump between the implementation and header files (Option+Command+up arrow on keyboard).
- **Lock**—Lets you set the read-only state of the current file.

You may have noticed the button at the top of the scroll bar on the right side of the editor pane. This is the handy split-view button, which lets you not only edit multiple parts of the same file at the same time but also open multiple files within an editor **KK**.



```
1 // @implementation helloWorldAppDelegate
2 // synthesized vars
3 // this is a mark
4 // -applicationDidFinishLaunching:
5 // FIXME: fix the bug on next line
6 // -dealloc
7 // TODO: need to remember to release everything here
8
9 // Implementation helloWorldAppDelegate
10 // Methods we'll synthesize our vars
11 // Synthesize window;
12 // Synthesize viewController;
13 // MARK: this is a mark
14 // - (void)applicationDidFinishLaunching:(UIApplication *)application {
15 //     // Overwrite point for customization after app launch
16 //     [window addSubview:viewController.view];
17 //     [window makeKeyAndVisible];
18 // }
19 //FIXME: fix the bug on next line
20 //[[[NSBundle mainBundle] release];
21 //[[[window release];
22 //[[[super release];
23 //
24 }
25
26 //MARK: this is a mark
27 // - (void)dealloc {
28 //     //FIXME: need to remember to release everything here
29 //     [[[viewController release];
30 //     [[window release];
31 //     [[super release];
32 // }
33 }
```

II The function menu showing multiple comment styles.



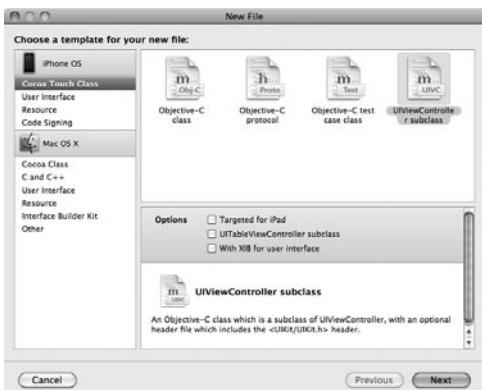
JJ Pop-up menus in the navigation bar.



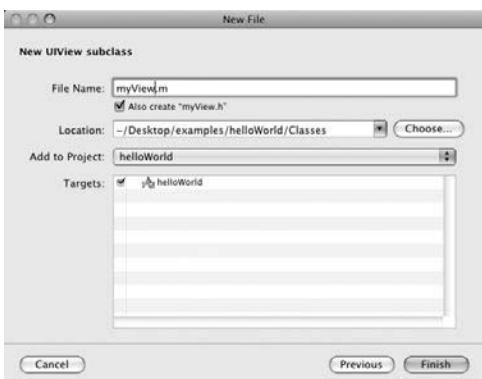
```
1 // @implementation helloWorldAppDelegate
2 // synthesized vars
3 // this is a mark
4 // -applicationDidFinishLaunching:
5 // FIXME: fix the bug on next line
6 // -dealloc
7 // TODO: need to remember to release everything here
8
9 // Implementation helloWorldAppDelegate
10 // Methods we'll synthesize our vars
11 // Synthesize window;
12 // Synthesize viewController;
13 // MARK: this is a mark
14 // - (void)applicationDidFinishLaunching:(UIApplication *)application {
15 //     // Overwrite point for customization after app launch
16 //     [window addSubview:viewController.view];
17 //     [window makeKeyAndVisible];
18 // }
19 //FIXME: fix the bug on next line
20 //[[[NSBundle mainBundle] release];
21 //[[[window release];
22 //[[[super release];
23 //
24 }
25
26 //MARK: this is a mark
27 // - (void)dealloc {
28 //     //FIXME: need to remember to release everything here
29 //     [[[viewController release];
30 //     [[window release];
31 //     [[super release];
32 // }
33 }
```

```
1 // @implementation helloWorldAppDelegate
2 // synthesized vars
3 // this is a mark
4 // -applicationDidFinishLaunching:
5 // FIXME: fix the bug on next line
6 // -dealloc
7 // TODO: need to remember to release everything here
8
9 // Implementation helloWorldAppDelegate
10 // Methods we'll synthesize our vars
11 // Synthesize window;
12 // Synthesize viewController;
13 // MARK: this is a mark
14 // - (void)applicationDidFinishLaunching:(UIApplication *)application {
15 //     // Overwrite point for customization after app launch
16 //     [window addSubview:viewController.view];
17 //     [window makeKeyAndVisible];
18 // }
19 //FIXME: fix the bug on next line
20 //[[[NSBundle mainBundle] release];
21 //[[[window release];
22 //[[[super release];
23 //
24 }
```

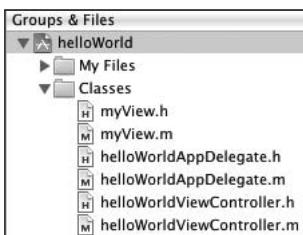
KK Splitting the editor allows you to work on the application delegate with both the header and implementation file open for editing.



LL Adding a new class file to the application.



MM Saving the new class.



NN The Classes folder showing the newly added classes.

TIP When creating a new class, the “Created by” and Copyright comments at the top of the files come from the current user’s Address Book card.

Creating new files

You create new files via the File > New File menu, which prompts Xcode to present a number of file templates for you to use during the application-creation process. In iPhone development you’ll normally use templates from the iPhone OS section only. Within this section are four subsections:

- **Cocoa Touch Class**—Is the most common area you’ll use; it provides templates for standard Objective-C iPhone classes.
- **Code Signing**—Allows you to create files related to signing your applications for ad hoc deployment.
- **Resource**—Provides templates for a number of resources, including Core Data models and application settings bundles.
- **User Interface**—Provides templates for creating the XIB files used in Interface Builder.

Now you’ll add a custom **UIViewController** subclass to your application.

To add a class to your application:

1. Select Cocoa Touch Class from the left side of the template window, and select “Objective-C class” as the file template. Notice how **UIView** is selected in the “Subclass of” drop-down menu LL.
2. Click Next, and save your file as **myView.m**. Make sure that “Also create ‘myView.h’” is selected, and click Finish MM.

You should see your new **myView.h** and **myView.m** files in the Classes group of the Groups & Files pane NN. When you open the **myView.m** file, you can see that Xcode has created stub placeholders for the most common methods.

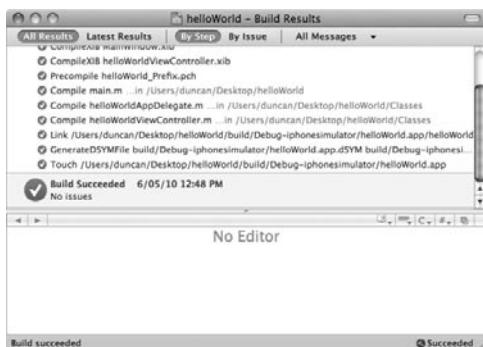
Building and running your application

To run your application, click Build and Go in the toolbar, or press Command+R. The Build Results window will open, displaying the status of your application target's build (you may have to enable this in the Building section of the Xcode preferences) **QQ**.

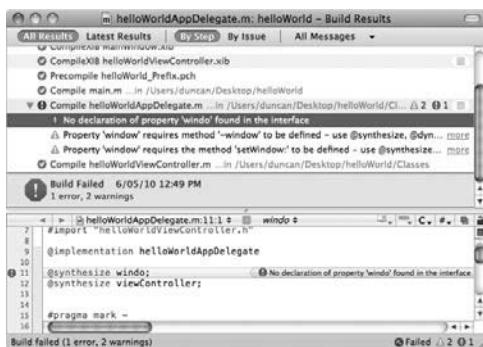
The Build Results window lists any errors and warnings in your application. Clicking one will open an Xcode editor pane directly in the Build Results window, allowing you to fix the error without having to leave the current window. Double-clicking opens a new editor window **PP**.

Some build errors may not be immediately obvious, such as a misreferenced file in your project. By right-clicking the build details and choosing Expand All Transcripts, you can show or hide the build details **QQ**.

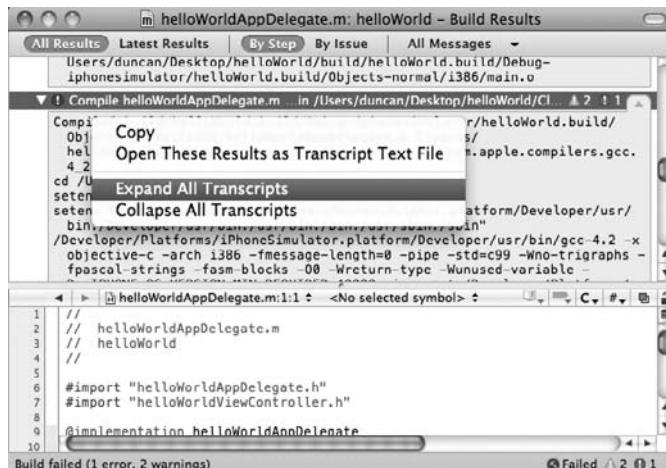
If there are no errors in your application, the iPhone Simulator will open, install, and launch your application.



QQ The Build Results window shows details of the project build.



PP The Build Results window showing an error in the code.



QQ The Build Transcript button lets you show or hide your build details.



RR The Clean Target dialog box.

Cleaning

Each time you build your application target, Xcode performs only the actions required to update any files that were changed since the last build.

Xcode is not perfect, however, and sometimes it doesn't notice that a file has changed (such as when you replace an image in your project with another of the same name). In this situation, you need to perform what's known as a *clean*.

To clean a target, choose Build > Clean, or press Shift+Command+K on the keyboard.

Cleaning a target removes any intermediate files created during the previous build process. The next time you build, every file will be processed RR.

For more details on the Xcode build system, see the “Building Products” section in the *Xcode Project Management Guide*.

Table 2.4 summarizes some of the common editor shortcuts.

Building for iPhone vs. iPhone Simulator

If you look at the Overview toolbar item in both Xcode and the Build Results window, you'll notice it has a section called Active SDK SS.

This is how you determine whether you are deploying your application to an actual iPhone or to the iPhone Simulator.

Although the simulator will work out of the box, in order to install your application onto an iPhone (or iPod), you must have a provisioning profile and a development certificate installed on both your Mac and your iPhone.

You can find details of what these are and how to create and install them at the iPhone Dev Center Web site (<http://developer.apple.com/iphone>).



SS The Overview toolbar shows you whether you are building for the simulator or iPhone.

TABLE 2.4 Common keyboard shortcuts

Shortcut	Description
Shift+Command+E	Collapse/expand details pane
Option+Shift+Command+E	Collapse/expand details pane and Groups & Files pane
Option+Command+O	Open current document in new window
Option+left arrow	Go to previous word (+Shift to select)
Option+right arrow	Go to next word (+Shift to select)
Command+left arrow	Go to beginning of line (+Shift to select)
Command+right arrow	Go to end of line (+Shift to select)
Command+up arrow	Go to beginning of document (+Shift to select)
Command+down arrow	Go to end of document (+Shift to select)
Command+L	Go to line
Command+Z	Undo
Shift+Command+Z	Redo
Command+/	Comment/uncomment selected block
Command+D	Create bookmark
Command+\	Add/remove breakpoint
Control+Command+\	Deactivate/reactivate all breakpoints (projectwide)
Command+F	Find in current file
Command+G	Go to next match
Shift+Command+G	Go to previous match
Shift+Command+F	Find in project
Control+1	Open file-history menu
Control+2	Open function menu
Control+3	Open Included files menu
Control+4	Open Bookmarks menu
Option+Command+up arrow	Switch between header and implementation file
Command+double-click	Jump to definition of current symbol
Shift+Command+K	Clean current target
Command+B	Build application
Command+R	Build and run application
Command+Y	Build and debug application

About the iPhone Simulator

You will usually use the iPhone Simulator for the majority of your iPhone development. The simulator lets you build, run, and test iPhone applications on your computer rather than on an actual iPhone.

Although you can simulate most of the iPhone environment, the simulator can't do a number of tasks well, and it has several important differences from the iPhone environment.

The simulator has *significantly* better performance than an actual iPhone. Everything from application launch time and general UI speed to available memory are much improved on the simulator. How much better it performs will depend on the computer you are developing with, but if speed is important to your finished application, you should be testing your code on an actual iPhone as early as possible in your development cycle to get an accurate measure of its actual performance.

The simulator provides a number of functions that mimic an iPhone:

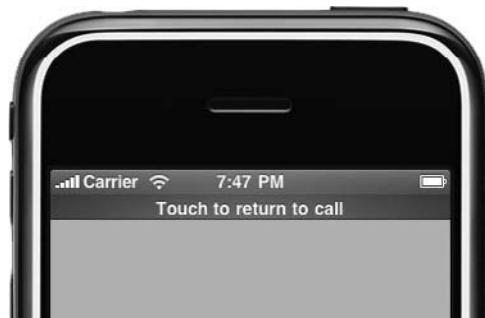
- **SDK version**—You can simulate your code running under multiple versions of iOS.
- **Rotation and shakes**—The simulator can be rotated left and right as well as shaken, triggering the appropriate delegate methods in your view controllers.
- **Home and Lock buttons**—Both buttons work as they do on the iPhone.

continues on next page

- **Memory warnings**—You can trigger low-memory warnings, which can be very useful in making sure your applications handle low-memory situations gracefully.
- **Toggle in-call status bar**—This status bar allows you to see how your application would look with the in-call status bar active **A**.
- **Touch**—Using the mouse, you can simulate either a single touch or, by holding the Option key, a two-finger touch. For more details on touch gestures in the simulator, see “Performing Gestures” in the *iPhone Development Guide* of the iPhone Reference Library.

The simulator can’t do the following:

- No camera means you cannot simulate any of the image-capture functionality of the iPhone.
- Although there are options to rotate the simulator, you cannot fully simulate the accelerometer.
- Core Location and Map Kit will always report your location as Apple’s headquarters in Cupertino.
- You cannot simulate more than two touches at once.
- The simulator has no iPod, iTunes, Calendar, Mail, SMS, Phone, or Maps applications.
- The Settings application has only limited functionality; for example, it doesn’t have airplane mode or data settings.
- It doesn’t have a proximity sensor.



A The simulator showing a call in progress.



B Drag an image onto the simulator to have it open in Safari.

If your application needs access to any of these functions, you will need to test your code on an actual iPhone rather than on the simulator.

Since there is no camera on the simulator, to add photos to the Photos application, you must follow these steps.

To add photos:

1. Open the simulator, and make sure you are at the iPhone desktop screen.
2. Drag any image from your computer's desktop onto the iPhone Simulator. Safari will open on the simulator with your photo displayed **B**.
3. Touch and hold the photo. A dialog box will pop up allowing you to save the image to the Photos application.

TIP Once your applications are installed on the simulator, you can touch and hold to change their order or delete them—just as you can on a real iPhone.

TIP Choosing Reset Content and Settings in the iPhone Simulator menu will remove all applications and reset the simulator to its initial configuration. However, it also *deletes* the default address book and photo album data. You can back up this data before you reset the simulator by copying the contents of `~/Library/Application Support/iPhone Simulator/User`.

About Interface Builder

Interface Builder (often abbreviated IB) is a visual tool used for creating your iPhone user interfaces.

IB lets you visually lay out the interface just as it would look on the iPhone, adding elements such as buttons, views, and labels and setting properties such as color and size.

You can also visually connect your interface to your application's code using a special system that encompasses outlets and actions.

An *outlet* is a connection from your code to your user interface. Imagine you have a label in your iPhone application that you want to display a message with. Using IB, you drag this label onto your interface. You then create an outlet between your code and your interface, which allows you to manipulate the properties of the label programmatically in your code.

An *action* goes the other way: from your interface to your application code. If you had a button on your interface that you wanted to have call a certain piece of code when clicked, you would create an action in your code and then hook up the button to this action in IB. When the button is clicked, the code is executed.

These concepts are common among many languages with visual design tools; however, the process of creating outlets and actions can be somewhat confusing at first. You'll examine this later in the chapter.



A The contents of the XIB file.

Your interface is saved as an XML-format XIB file. When you build your application, the XIB files are compiled into a deployable NIB file, which is then packaged up with your application bundle. At run time, your iPhone application will automatically unpackage and load these files to re-create your user interface.

Now you'll take a quick look at the various elements of the IB interface.

In Xcode, double-click the file named `helloWorldViewController.xib` (in the XIB Files group of the Groups & Files pane) to open your XIB in IB.

There are three main parts to the IB interface: the document window, the Library window, and the inspector window.

TIP NIB stands for “NeXT Interface Builder.” The file format was originally created for the NeXT operating system in the 1980s. The newer XIB format was introduced in Xcode 3.0. In this chapter, XIB and NIB both mean an Interface Builder file.

About the document window

The document window shows the contents of the XIB file A. In this example, you can see three objects.

- **File's Owner**—This is a proxy object representing an instance of a class (in this case the `helloWorldViewController` class). You will create your outlets and actions in objects like these.

continues on next page

- **First Responder**—Every XIB will have a First Responder object, providing an entry point for your interface into the responder chain, which consists of a list of objects connected together. Touch and motion events are sent to the first object in the chain (the **firstResponder**), which can choose to process it or pass it on up the chain to the next object via its **nextResponser** property.

All UI objects (including **UIViews** and **UIWindows**) are responder objects, as is your **UIApplication**. You can create custom actions on this object and have your visual elements call these actions. Objects in the responder chain can then handle these custom actions.

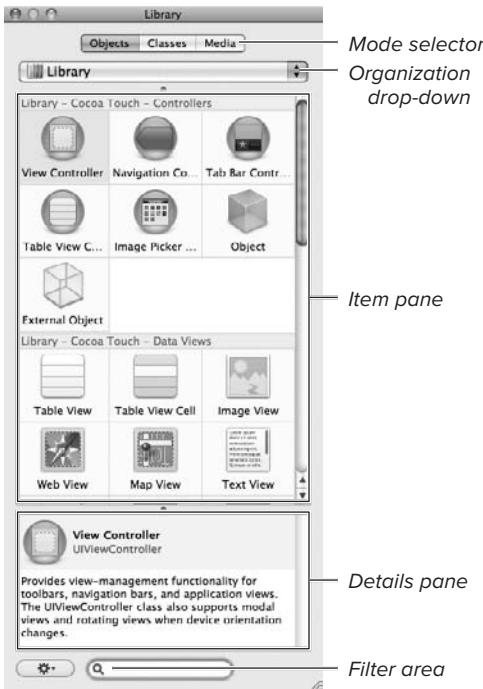
- **View**—This represents the interface of your iPhone application (in this example it is an instance of a **UIView** class).

Note: Here you have only a single object placeholder and a single view. It is, however, quite possible (and common) to have more than one of each within a single XIB file.

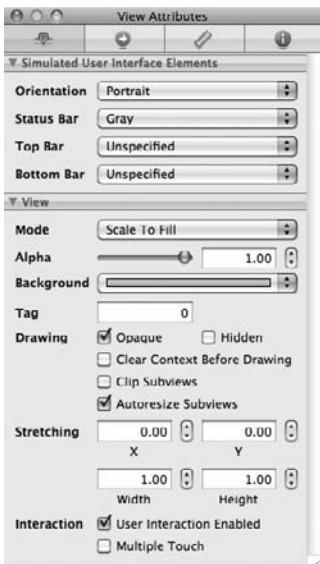
Often an interface can have many nested UI controls on it: labels on buttons within views within yet more views. This can make it very difficult to determine which object is which in the document window. Although you can name your objects (by clicking the label underneath them), the View Mode options at the top left make it much easier to visualize these hierarchies **B**.



B Changing the view mode of the XIB to visualize the object hierarchy.



C The Library window.



D The inspector window.

About the Library window

The Library window is divided into three main areas **C**.

- **Organization drop-down**—Divides your objects into functional groups
- **Item pane**—Lists the contents of the currently selected group (or groups), graphically depicting the functionality the object performs
- **Details pane**—Shows the name and a brief description of what the selected object is and what it does

You can filter the items shown by entering search text in the filter control at the bottom of the Library window.

Notice the *mode* selector tabs at the top, which allow you to switch between user interface objects, classes, and media resources. Resources will include the images and sound files within your project.

TIP Just as with the Groups & Files pane of Xcode, you can create your own groups and smart groups in the Organization pane of the Library window.

TIP The Action menu at the bottom of the Library window allows you to modify the way items are presented.

About the inspector window

The inspector window allows you to configure the properties and settings for each of the elements within the document window **D**. It's divided into four panes.

- **Attributes**—This is the main area for setting the visual properties of your UI elements, such as colors, fonts, and text. You can also set drawing properties and choose whether an element can respond to touches from the iPhone screen.

continues on next page

As you click each object in your document window, notice how the inspector window automatically updates to show the different properties and settings available for each object.

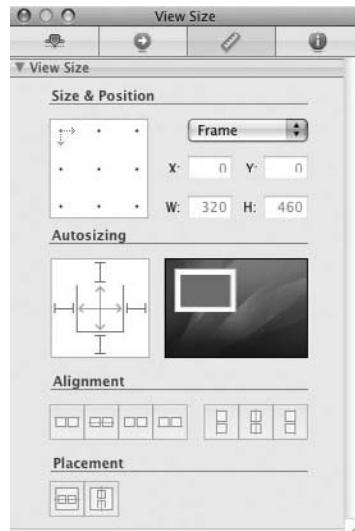
- **Connections**—This pane displays the outlets and actions for the currently selected object. If an outlet or action is connected, you will see the name of the object it is connected to **E**. You'll take a look at how to go about connecting outlets and actions later in this chapter. You can also see the connections for an object by Control+clicking it in the document window **F**.
- **Size**—This pane allows you to set the origin (the x and y coordinates) and size (width and height) of your visual elements. You can also adjust the autosizing behavior of your elements, which can be useful when your interface changes from portrait to landscape mode. Autosizing is intuitively shown by a small animation next to the control **G**.



E Outlets are visually represented in the Connections pane.



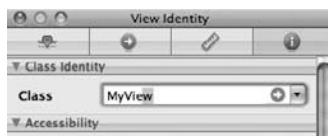
F Control+click to view an object's outlets and connections.



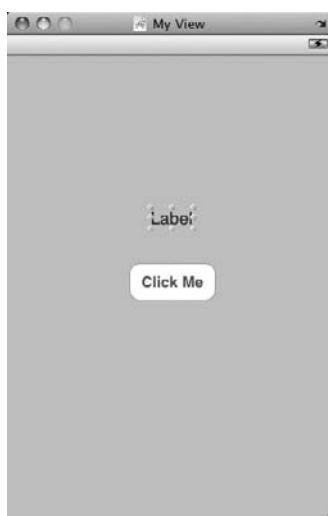
G The Size pane of the inspector window.



H The Identity pane of the inspector window.



I Changing a class in the Identity pane.



J Visually lay out your application by dragging elements onto the iPhone canvas.

■ **Identity**—The Identity pane shows you the class of your object, allowing you to override this with your own custom class should you choose to do so. You can visually add outlets and actions here and set the name used to display your objects in the document window (H).

The Identity pane is also the key to adding nonvisual instances of your own classes to your XIB files. After creating and saving your class in Xcode, simply drag an NSObject from the Library window and then rename its class to your own class name (I).

You'll now take a look at how to use all of this to create your own interface.

To create the interface:

1. Drag a UIButton from the Library window, drop it on your view, and set its text to Click Me.
2. Drag a UILabel from the Library window, and drop it on your view above the button (J).
3. Save your interface, go back to Xcode, and build and run your application.

You can click the button, but at the moment nothing happens.

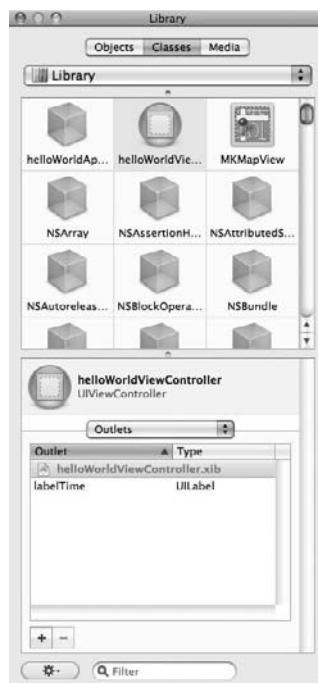
Now you will update your application to have it display the current time when you click the button.

To update the application:

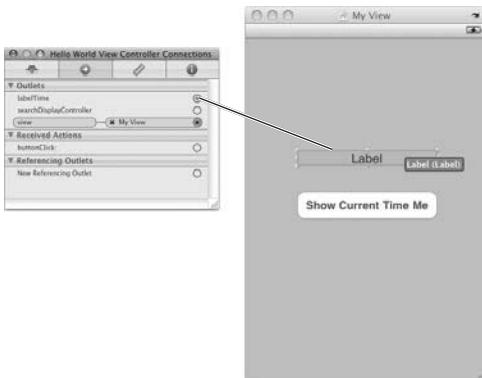
1. Back in IB, update the UIButton text to Show Current Time. You do this via the inspector window or just by double-clicking the button and typing. Notice how the button automatically resizes itself to fit the wider text. Resize your label to make it the same width as your button, and set the text alignment to center (the Alignment property is on the Attributes pane of the inspector window).
2. Select the Classes mode selector in the Library window, and select the helloWorldViewController object.
3. Select Actions from the drop-down. In the Class Actions area, click Add (+), and set the action to buttonClick **K**.
4. Select Outlets from the drop-down, and add an Outlet called labelTime. Change the type to UILabel **L**.
5. Making sure your File's Owner object is still selected in the document window, click the Connections pane of the inspector window.



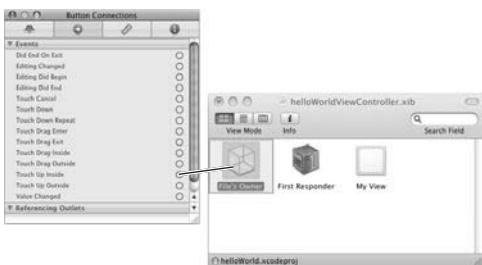
K The Library window showing your class's actions.



L The Library window showing your class's outlets.



M Hooking up the interface by dragging between elements.



N Connect actions by dragging between the actions window and the object you want to receive the action.



O Updating your code with the outlets and actions you've just created.

6. Connect the outlet by clicking the small circle on the right side of the `labelTime` outlet. Keep your mouse button pressed, and drag to the label on your view to make the connection **M**.

7. To connect the action, you need to do the reverse. Click your button in the document window, select the Touch Up Inside event, and drag to your File's Owner object. Select the `buttonClick:` action when it pops up to make the connection **N**.

8. Choose File > Write Class Files from the main menu. Then click Save **O**.

9. Click Replace in the next dialog box (or, if you like, you can click Merge to see what is happening).

10. Back in Xcode, open the file named `helloWorldViewController.h`. Notice how IB has generated code for the outlet and the action:

```
@interface
- helloWorldViewController : UIViewController {
    IBOutlet UILabel *labelTime;
}
- (IBAction)buttonClick:(id)sender;
```

11.

continues on next page

- 11.** In `helloWorldViewController.m`, edit this stub code so that it looks like this:

```
- (IBAction)buttonClick:  
- (id)sender {  
  
    NSDateFormatter *dateFormatter  
    → = [[NSDateFormatter alloc]  
    → init];  
  
    [dateFormatter setDateFormat:  
    → @"hh:mm:ss"];  
  
    labelTime.text = [dateFormatter  
    → stringFromDate:[NSDate date]];  
  
    [dateFormatter release];  
}
```

- 12.** Build and run your application. You should now be able to click the button, and the label will update to show the current time **P**.

In the previous example, you got IB to automatically generate the stub code for your outlet and action. Although this is convenient, you wouldn't normally do it with anything other than an empty class. It's much more common to create your outlets and actions yourself in Xcode manually and then use IB to hook them up to your user interface. Now you'll take a look at how you do this.

To manually create outlets and actions:

1. In Xcode, open the file named `helloWorldViewController.h`, and edit it to look like **Code Listing 2.1**. You will have created two more outlets and replaced your `buttonClick:` action with one called `controlChanged:`.



P The application running in the simulator.

Code Listing 2.1 The header file for the hello world application.

```
#import <UIKit/UIKit.h>  
#import <Foundation/Foundation.h>  
  
@interface helloWorldViewController : UIViewController  
{  
    IBOutlet UILabel *labelTime;  
    IBOutlet UISwitch *dateSwitch;  
    IBOutlet UISlider *fontSlider;  
}  
  
- (IBAction)controlChanged:(id)sender;  
|  
@end
```



Q Adding some more elements to the interface.

Code Listing 2.2 The updated code for the hello world application.

```
#import "helloWorldViewController.h"

@implementation helloWorldViewController

-(void)updateTimeLabel
{
    NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];

    if (dateSwitch.on)
        [dateFormatter setDateFormat:@"dd-MMM-yyyy, hh:mm:ss"];
    else
        [dateFormatter setDateFormat:@"hh:mm:ss"];

    labelTime.text = [dateFormatter stringFromDate:[NSDate date]];
    labelTime.font = [UIFont systemFontOfSize:fontSlider.value];
    [dateFormatter release];
}

-(IBAction)controlChanged:(id)sender {
    [self updateTimeLabel];
}

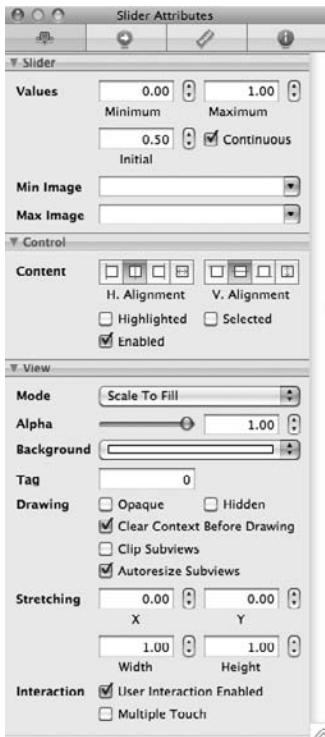
//Update time when view first loads
-(void)viewDidLoad
{
    [self updateTimeLabel];
}

@end
```

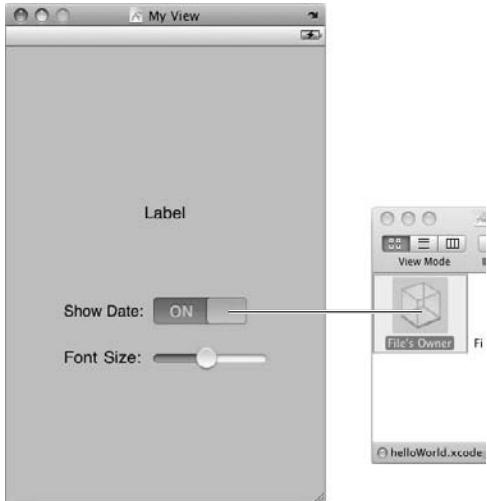
2. Switch over to `helloWorldViewController.m`, and edit it to look like **Code Listing 2.2**. You have modified your code, moving the label update into its own method and adding the option to display the date or the time. You've also added the ability to modify the font size of the label. Notice also that you call this method in the `viewDidLoad` method. This is so the label shows correctly when the application first starts.
3. In the Groups & Files pane, double-click your `helloWorldViewController.xib` file to open IB. Delete the button from your interface, and drag two labels—a `UISwitch` and a `UISlider`—onto your view. Arrange them so they look like Q. Notice how you've also increased the width of your time label to handle a larger font.

continues on next page

- Select your UISlider, and edit its Minimum Value setting to 10 and its Maximum Value setting to 25 **R**.
- Keeping your UISlider selected, press the Control key, and drag from the slider to the File's Owner object in the document window **S**. Then select controlChanged: in the pop-up menu. This is a shortcut method of connecting outlets and actions: Instead of using the Connections menu, you can simply Control+drag between objects to have the default action connected. Also connect the outlet from the File's Owner to the UISlider.



R Configuring the slider.



S Connecting the slider to the controller.



1 The updated application.

6. Repeat step 5, but this time for the UISwitch control, again selecting the controlChanged: option. Notice how you can have multiple controls connected to the same action. Again, also connect the outlet from the File's Owner object to the UISwitch control.

Back in Xcode, build and run your application. You should be able to switch between the date and time, as well as change the font size by moving the slider 1.

TIP Although IB automatically scans your code for IBAction and IBOutlet definitions in your header files, it can occasionally miss them, and you'll find you can't see your outlets and actions in the inspector window. You can solve this by manually dragging the .h file of your class from the Xcode Groups & Files pane onto the IB documents window.

Why are there two XIB files in the project?

In the previous example, you may have noticed that you are working only with `helloWorldView` Controller.xib, but your project actually contains two XIB files. So, what does `MainWindow.xib` do?

In the Xcode section, the `<app>-Info.plist` file has the property Main Nib File, which is the NIB that is loaded automatically when your iPhone application starts. `MainWindow.xib` is this file.

Opening `MainWindow.xib`, you can see several objects:

- **File's Owner**—An instance of **UIApplication** representing the entry point of your application, created by the **UIApplicationMain()** function call in `main.m`. You will only ever have a single **UIApplication** object in your iPhone applications.
- **helloWorld App Delegate**—An instance of your **helloWorldAppDelegate** class (the `helloWorldAppDelegate.h` and `helloWorldAppDelegate.m` files in your project). Since this object is the delegate of your **UIApplication**, it handles all of the startup behavior of your application. Notice it has **viewController** and **window** outlets.
- **helloWorld View Controller**—An instance of your **helloWorldViewController** class (the `helloWorldViewController.h` and `helloWorldViewController.m` files within your project). If you inspect this object in the inspector window, you will notice that it has its NIB Name property set to `helloWorldViewController` ①. This tells the object to initialize itself from the `helloWorldViewController.xib` file. This is how the XIB file you have been working with so far gets loaded.



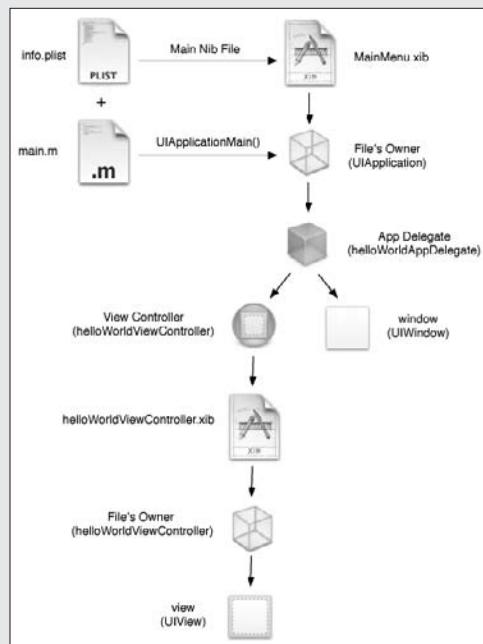
① The NIB Name property.

Why are there two XIB files in the project? (continued)

When an XIB is loaded, every object inside is created. By using this pattern of one XIB per view controller, you can lazily load your interface as you need it. This decreases your application's loading time and uses less memory. Having your views and view controller in individual XIB files also makes things easier to work with in IB—imagine how complex the outlets and actions would be if everything were in a single file!

- **Window**—Represents the top-level or root view in your application's interface. You will only ever have a single window in your application; views you create will be added as subviews to this object. Having this top-level view ensures that things like alert dialog boxes will always appear above your own views.

When your application is launched, the interface is created .



 An overview of how the application is constructed.

About the Documentation

As you saw earlier when looking at Xcode, iOS includes a comprehensive and powerful documentation viewer. You can launch the documentation viewer by using the Help menu, by pressing Option+Command+?, or by selecting a symbol in your code and Option+double-clicking.

Documentation is presented in an HTML-based, multiframe layout that allows you to move around related information by clicking hyperlinks in the text, just as you would in a Web browser A.

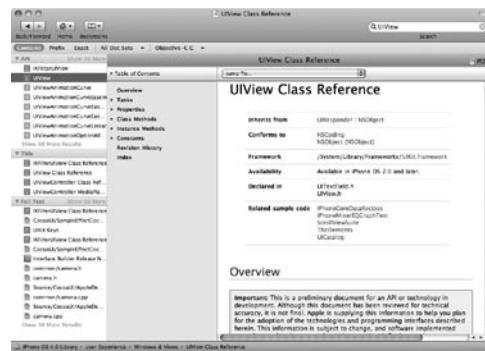
Above the main documentation pane, a history of your path through the pages is maintained, allowing you to navigate forward and backward or jump directly to a page via the pop-up menu.

In the left pane, you can subscribe to documentation feeds from Apple, which automatically update when new documentation becomes available. You can bookmark the page you are currently viewing by clicking the Bookmark button in the toolbar B.

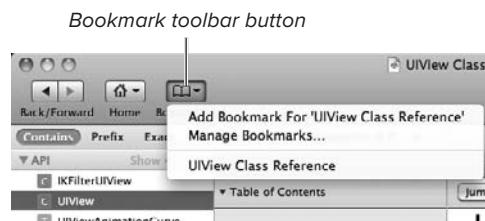
You can search the documentation by API (the fastest) or by document title; you can even search the full text of the documentation. The filter pane at the top allows you to limit searches to only the documentation sets and technologies in which you are interested.

TIP Pressing the Command key while clicking links in the documentation will make the page open in a new window.

TIP The latest documentation is also available online at <http://developer.apple.com/iphone>.



A The documentation browser



B The Bookmarks pane

The Xcode Organizer

The Xcode Organizer provides convenient access to a number of tasks you will need to perform during the iPhone development process. You can open the Organizer by selecting it from the Window menu in Xcode or by pressing Control+Command+O. The interface consists of two panes with a number of items on the left and their corresponding details on the right. Within the Organizer, there are several sections to explore.

Unique device identifier

The unique device identifier (also known as a UDID) is a way of identifying an iPhone or iPod. This identifier is used when adding devices in the iPhone Provisioning Portal and is also used when generating both Development and Ad Hoc provisioning profiles. A *provisioning profile* is necessary to run your application outside of the iPhone Simulator on an actual device (iPhone, iPod, or iPad). You would generally create three provisioning profiles: a Development profile associated to your own iPhone's UDID; an Ad Hoc profile used for beta testing, associated with up to 100 other UDIDs; and a Distribution profile used when submitting your application to the App Store on the iTunes Connect Web site. You can find details on creating and managing code-signing identities on the iPhone provisioning portal at <http://developer.apple.com/iphone>.

Clicking the Save As Default Image button on the Screenshots tab allows you to add a graphic to your project to be used as a load screen when your application first starts

A It is common for developers to use the opening or initial screen of their application as a load screen. This gives the impression of a speedy load time, even if the application has not yet fully loaded.



A The Screenshots tab allows you to manage and capture screenshots from a connected device.

Projects & Sources

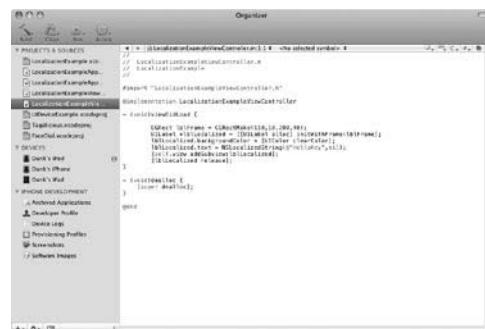
This area gives you access to frequently used directories and files. You can drag any item into this area, as well as perform other frequent operations such as editing, moving, and deleting files and directories **B**. You can even build and run projects directly from this area rather than opening them in Xcode.

Devices

The Devices section lists any iPod, iPhone, or iPad that can be used for development purposes. If a device is currently attached to your computer, an indicator graphic appears, showing whether it can be used for development (green) or whether there is something wrong (yellow or red). For example, you must have the correct SDK installed on your computer for the operating system currently running on the device **C**.

Selecting a device displays information including the device capacity, serial number, identifier, and which version of iOS your device is running **D**. If the device is connected, you will also be shown a list of all the provisioning profiles and applications installed on it. You can manually install and remove provisioning profiles by dragging them into the list or by using Add (+) and Delete (-). Similarly, you can add any applications you develop yourself to your device in this way rather than using iTunes.

The tabs at the top allow you to see logging information for your application both on the device and on your computer. You can also capture screenshots of any application running on your iPhone from here, which can be very handy when submitting to the iTunes Connect Web site. You can also drag images in this area onto your computer desktop.



B Editing a file directly in the Xcode Organizer.



C A graphic indicates the status of any connected devices.



D The device summary page.

Sharing

Validate Application...

Share Application...

Submit Application to iTunes Connect...

- E** The Sharing section helps with validating an deploying an iPhone application.



- F** Deployment options for sharing an application.

iPhone Development

This area allows you to manage a number of tasks specific to iPhone development:

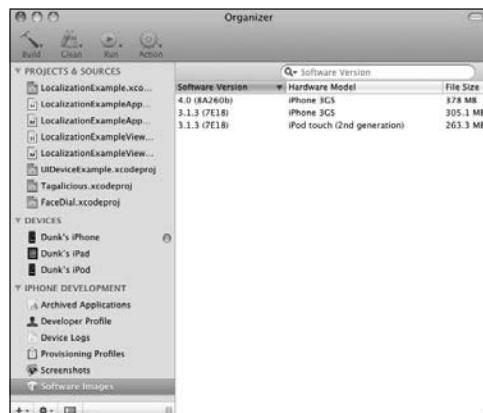
- **Archived Applications**—This lists all the archives created as a result of using the Build and Archive command from within Xcode. This command will create an archive containing your application bundle (.app) and the debug symbols file (.dSYM) used to decode any crash logs generated by an Ad Hoc or release build of your application. Archives are stored in the ~/Library/MobileDevice/Archived Application directory. Using the buttons underneath **E**, you have three options for working with archived applications.
- **Validate Application**—This runs a number of validation tests on your application, such as checking for the correct icon, checking that your .plist is correctly configured, and checking that the application has been correctly code-signed. You should always validate your application before submitting to the iTunes Connect Web site.
- **Share Application**—This provides a convenient way of creating Ad Hoc distributions of your application, often used when beta testing an application. You can choose to save to disk or e-mail the application directly from this screen **F**.

continues on next page

- Submit to iTunesConnect**—This allows you to sign your application with your Distribution identity and upload it directly to Apple for submission to the App Store. Note that you can submit only the application *binary* using this function. You need to have already set up the application on the iTunes Connect Web site, along with associated screenshots, description, keywords, and so on.
- Developer Profile**—This shows details of all the certificate identities and provisioning profiles installed on your computer. These are necessary for you to be able to install and run your applications on an iPhone, iPod, or iPad. The Import and Export profile buttons at the bottom of this screen provide a convenient way of migrating your identities and profiles between computers **G**.
- Device Logs**—This displays all the log files on any device paired with your computer. This can be handy for trying to figure out the cause of a crash in your applications. You can also see logs generated by the other applications on your iPhone.
- Provisioning Profiles**—This lists all the provisioning profiles available and which of your devices they are installed on.
- Screenshots**—This shows all the screenshots you have taken from any devices previously attached to your computer.
- Software Images**—This lists any operating software images you have on your computer **H**. Right-clicking an image lets you reveal the location of the image in the Finder, which can be handy should you need to restore a device to a different version of an operating system.



G The Developer Profile section shows you the identities and profiles installed on your computer.



H Showing the versions of the iPhone OS available on your computer.

3

Common Tasks

This chapter will show you some of the useful tips and tricks that are essential for building a great iPhone application. You'll start by looking at the application startup process, where to put your code, and how you set the various configuration options.

Next you'll see how easy it is to add user and application preferences to your applications. You'll also learn how to create an application preference page within the Settings application.

Then you'll learn about some of the techniques you can use to internationalize an iPhone application, making it available in different languages. You'll also find out how to make your applications accessible to people with visual impairments.

Finally, you'll learn how to launch other applications from your own, as well as some techniques for sharing data between applications.

In This Chapter

Application Startup and Configuration	84
Localization	94
Accessibility	98
Interapp Communication	103

Application Startup and Configuration

In this section, you'll learn what happens when you tap the icon to launch your iPhone application. You'll see how you can respond to different stages in the application startup and shutdown processes. You'll also look at some of the configuration options available for your applications and how to work with user and application preferences.

Using the application delegate

The application delegate, represented by the `UIApplicationDelegate` class, is the starting point for all iPhone applications. Since there is only a single application delegate, it can be a useful place to put methods and variables you want to be available globally.

You add a reference to the application delegate from within your classes by writing this:

```
id appDelegate = [[UIApplication  
→ sharedApplication] delegate];
```

(You also need to remember to import the header file of your application delegate.)

However, since the application delegate needs to be loaded before your application can begin, it's probably good practice to avoid putting code there unless absolutely necessary. Instead, you can adopt a "lazy loading" style of programming where you create objects only when needed. This has the added benefit of keeping the memory footprint of your applications to a minimum.

The application delegate serves as both the first place (when your application loads) and the last place (before your application exits) that you can execute some code. It provides a number of important methods:

- **applicationDidFinishLaunching:**—This is the first method called when your application has loaded. This is a good place for you to perform application-wide initialization of data such as loading any previous state or settings.
- **applicationDidFinishLaunching WithOptions:**—If implemented, **applicationDidFinishLaunching:** will be ignored, and this method will be used instead. The **launchOptions** parameter is a dictionary containing information used in two scenarios:
 - It will be used if your application was launched as the result of a “push” notification, in which case **launchOptions** will contain a dictionary of information relating to the notification. See the *Apple Push Notification Service Programming Guide* for more information on working with push notifications.
 - It will be used if your application was launched as a receiver of a URL resource. See the “Interapp Communication” section later in this chapter for more information.
- **applicationWillTerminate:**—This is the last chance to perform any action before your application exits. This is a good place for you to free memory and save application state or settings such as user preferences.

continues on next page

- **applicationDidReceiveMemoryWarning**

Warning:—This is called whenever the device is running low on memory due to improper memory management or simply because your application is trying to load too much data into memory. Normally you would attempt to free as much memory as possible in this situation by releasing any cached or unnecessary objects.

If this method is called, then the corresponding **didReceiveMemoryWarning** delegate method will also be called on every view controller class in your application. This gives you granular control over your application, allowing you to clean up memory on a per-view controller basis. Although optional, it's recommended that you try to implement this method, because your application may become unstable or crash if memory warnings are ignored.

- **applicationWillResignActive:**—

This is called just before your application becomes inactive. This can happen if another window pops up over your application (for example, an incoming phone call) or if the device goes into a locked state, either by going to sleep or by the user pressing the power button.

- **applicationDidBecomeActive:**—

This is the reverse of the previous method and will be called every time your application becomes active. This obviously happens when the application is first launched but also when you dismiss any windows that have appeared over your application (for example, an incoming text message) or when the device wakes from a locked state. You can prevent your application from going to sleep by writing this:

```
[[UIApplication sharedApplication]
→ setIdleTimerDisabled:YES];
```

TIP Some applications have splash screens that display while the application is loading. You can add a splash screen to your own application by adding a file called Default.png to the project. Rather than displaying a graphical splash screen (which can itself increase load time), many developers opt to show an image that replicates the initial screen of the application.

TIP After your application has launched, you can use the UIDevice class to determine information such as the make, model, operating system name, and version of the device on which it's running.

LocalizationExample-Info.plist	
Key	Value
Information Property List	{12 items}
Localization native development region	English
Bundle display name	\$PRODUCT_NAME
Executable file	\$(EXECUTABLE_NAME)
Icon file	
Bundle identifier	com.yourcompany.\$PRODUCT_NAME.rfc1034identifier
InfoDictionary version	6.0
Bundle name	\$PRODUCT_NAME
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone environment	✓
Main nib file base name	MainWindow

A The Info.plist file contains the configuration settings for your application.

Understanding application settings

As mentioned in the “About the Xcode IDE” section in Chapter 2, “The iPhone Developer’s Toolbox,” the <app>-Info.plist file contains many of the preferences and settings used when launching your application. A. Table 3.1 summarizes some of the more commonly used settings.

Working with user preferences

User preferences are generally stored using the **NSUserDefaults** class, also known as the *defaults system*. You would normally read from the defaults system when your application first launches to restore any previous settings for the user or application.

The **NSUserDefaults** object is quite smart. When your application launches for the first time, there obviously won’t be any user preferences (other than those you set programmatically). Once you set a value, the defaults system will automatically save it for you. The next time you launch the application and request the value, it will be returned to you. All the complexities of dealing with loading and saving files and values are hidden, and for performance everything is cached in memory. This cache is periodically refreshed automatically.

To begin using the defaults system, you first get a reference by writing this:

```
NSUserDefaults *prefs =
→ [NSUserDefaults
→ standardUserDefaults];
```

You can then read values by using something like this:

```
NSInteger age =
→ [prefs integerForKey: @"age"];
```

TABLE 3.1 Commonly Used <app>-Info.plist Settings

Key	Description
Application does not run in background	If you want your application to terminate rather than move into the background when quit, set this property to True. For information about multitasking, see Chapter 11.
Application supports iTunes file sharing	This property allows you to share files in your application's documents directory with your desktop computer via iTunes.
Application requires iPhone environment	If your application does not run on an iPod touch, set this to True.
Application uses Wi-Fi	If your application requires Wi-Fi to function, you should set this property to True. Doing so will prompt the user to enable Wi-Fi if it is not already enabled. For power-saving, the iPhone automatically closes any Wi-Fi connections in your application after 30 minutes. Setting this property will prevent this from happening and keep the connection active.
Bundle display name	This sets the name of your application, displayed below the icon on the iPhone screen. You are limited to approximately 10 to 12 characters for an application name before the iPhone abbreviates the name.
Bundle identifier	This sets the unique identifier for your application setup in the iPhone Developer Program Portal Web site.
Bundle version	This sets the version number of your application. This is used in the iTunes App Store and should be incremented each time you deploy a new version of your application.
Icon already includes gloss and bevel effects	By default, the iPhone applies a “gloss effect” to application icons. Setting this key to True will prevent this.
Icon file	This is the filename of your application's icon (added to your project).
Initial interface orientation	This determines whether your application starts in landscape or portrait mode.
Launch image	This property allows you to specify the name of the launch image file for your application. This can be handy when building universal applications as you can specify a different launch image for iPad, iPhone, and iPod touch devices.
Localizations	This is a comma-delimited list of localizations that your application supports. For example, if your application supports English and Japanese, you would use English,Japanese. You should then provide the appropriate localization strings for each language. See the “Localization” section later in this chapter for more details on how to localize an application.
Status bar is initially hidden	This allows you to launch your application without a status bar. Your application interface will automatically resize to fill the entire iPhone screen.
Required background modes	If your application supports multitasking and runs in the background, this key lets you specify which services it should be allowed to run. For more information on multitasking, see Chapter 11.
Status bar style	If you leave the status bar visible, this key allows you to select from one of three different display styles.
Supported interface orientations	This key allows you to specify the initial orientation of your application when it first launches.
Main nib file base name	This is the XIB file loaded when your application first starts. You would not normally need to modify this value.
URL types	This is an array of URL identifiers supported by your application. See the “Interapp Communication” section later in this chapter for details on how to use this value.

You write values by using this:

```
NSInteger age = 30;  
[prefs setInteger:age forKey:@"age"];
```

Code Listing 3.1 shows some examples of reading and writing to the defaults system.

The defaults system has methods for working with property list datatypes (**NSNumber**, **NSString**, **NSData**, **NSArray**, or **NSDictionary**). If you want to store more complex values, you need to *archive* the value and store it as an **NSData** instance. You can then *unarchive* when reading the value back to convert from an **NSData** instance to the original datatype.

Code Listing 3.1 Some examples of reading and writing to the defaults system.

```
NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];  
  
//read values  
BOOL homeOwner = [prefs boolForKey:@"homeOwner"];  
NSInteger age = [prefs integerForKey:@"age"];  
NSString *userName = [prefs stringForKey:@"userName"];  
UIColor *favoriteColor;  
  
//is value in user defaults?  
NSData *colorData = [prefs objectForKey:@"favoriteColor"];  
  
if (colorData != nil)  
    favoriteColor = [NSKeyedUnarchiver unarchiveObjectWithData:colorData];  
else  
{  
    UIColor *myColor = [UIColor blueColor];  
    NSData *colorData = [NSKeyedArchiver archivedDataWithRootObject:myColor];  
    [prefs setObject:colorData forKey:@"favoriteColor"];  
}  
  
//change some values  
homeOwner = TRUE;  
age = 30;  
userName = @"Bob";  
  
//save back to prefs  
[prefs setBool:homeOwner forKey:@"homeOwner"];  
[prefs setInteger:age forKey:@"age"];  
[prefs setObject:userName forKey:@"userName"];
```

For example, to archive and store a **UIColor** object in the defaults system, you would use this:

```
UIColor myColor =  
→ [UIColor blueColor];  
  
NSData *colorData = [NSKeyedArchiver  
→ archivedDataWithRootObject:  
→ myColor];  
  
[prefs setObject:colorData forKey:  
→ @"myColor"];
```

To then unarchive and restore the value, you write this:

```
NSData *colorData = [prefs  
→ objectForKey: @"myColor"];  
  
UIColor myColor = [NSKeyedUnarchiver  
→ unarchiveObjectWithData:colorData];
```

TIP While developing your applications, it is possible to force-quit them (by pressing Command+Q). Although this is not something that can happen when your application is in production, doing so may mean that the defaults system does not save correctly. You can get around this problem by calling the synchronize method, which will force the defaults system to be saved and written to disk.

Application preferences

Just like user preferences, application preferences are stored and retrieved by using the **NSUserDefaults** class and are managed in the same way in your code.

iOS provides a second mechanism for managing application preferences, allowing you to create a page in the global Settings application **B**. The user can then edit the application preferences by using this page.



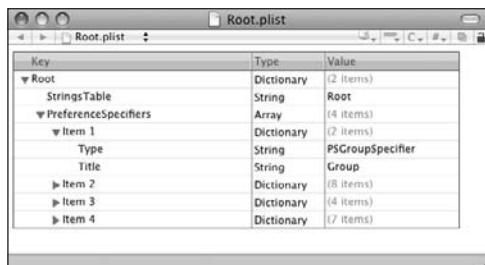
B The Settings application can be used to edit your application settings.



C Adding a settings bundle to your project.



D The settings page of your application.



E The PreferenceSpecifiers key contains the various items in the settings page.

To create a settings page for your application:

1. In Xcode, select File > New File.
2. Select the Resource section, and select Settings Bundle to create a new settings bundle C.
3. Save the file as Settings.bundle.
4. Build and run your application.
5. Switch to the Settings application. You will notice a new section with the same name as your application. Select this section to open the settings page showing four elements D.
6. In Xcode, expand the Settings.bundle file in the Groups & Files pane, and select the Root.plist file.
7. In the editing window on the right, expand the PreferenceSpecifiers section E.

The four items (items 1–4) represent the four interface elements in the settings page from step 2. If you expand each element, you will notice they all have a Type key, which determines the control type to display.

Adding controls

You can add seven types of controls to your settings page:

- **PSGroupSpecifier** creates a new group of controls, such as a settings page with several groups **F**.
- **PSTextFieldSpecifier** creates a text field where the user can enter a string of text **G**. This control has several properties:

isSecure stops the text you are entering from being shown as it's typed. This is useful for passwords and other sensitive information.

DefaultValue lets you specify a default value if no value has been set.

KeyboardType allows you to specify which keyboard is shown when the user taps the control.

AutoCapitalizationType controls the type of capitalization that occurs as the user types.

- **PSToggleSwitchSpecifier** creates a toggle (on/off) switch. The **DefaultValue** property determines whether the switch is on or off by default.
- **PSSliderSpecifier** creates a slider. You use the **MinimumValue** and **MaximumValue** properties to determine the range of the slider, and you use the **DefaultValue** property to determine the starting value.



F A settings page with three groups.



G Entering text into a settings page.

Root.plist		
Key	Type	Value
Root	Dictionary (2 items)	
StringsTable	String	Root
PreferenceSpecifiers	Array (5 items)	
> Item 1	Dictionary (2 items)	
> Item 2	Dictionary (6 items)	
Title	String	
Type	String	PSMultiValueSpecifier
DefaultValue	Number	0
Key	String	color_preference
> Titles	Array (3 items)	
Item 1	String	Red
Item 2	String	Green
Item 3	String	Blue
> Values	Array (5 items)	
Item 1	Number	0
Item 2	Number	1
Item 3	Number	2
> Item 3	Dictionary (6 items)	
> Item 4	Dictionary (4 items)	
> Item 5	Dictionary (7 items)	

Debugging terminated.

Succeeded

- ① Use the **PSMultiValueSpecifier** type to select from a set of multiple values.



- ① Choosing from a set of values.

- **PSTitleValueSpecifier** creates a simple read-only label. Use the **DefaultValue** property to set the text of the label.

- **PSMultiValueSpecifier** creates a second view consisting of a table of values that the user can select from ①. The **Values** property contains an array of possible values. The corresponding **Titles** property contains the display labels for each value and accepts an array of values to display ②. The **DefaultValue** property determines which value is initially selected.

- **PSChildPaneSpecifier** allows you to create another settings view. By setting the **File** property to the name of another setting's plist, you can nest settings within settings.

For all the control types (except the slider that has no title), the **Title** property enables you to set the title text that appears to the left of the control.

The **Name** property of each control is the key you use to reference the setting with **NSUserDefaults**. For example, to get the value of the **PSToggleSwitchSpecifier** toggle switch defined in item 3 of the **PreferenceSpecifiers** array, you would write this:

```
NSUserDefaults *prefs =
→ [NSUserDefaults
→ standardUserDefaults];
BOOL enabled = [prefs boolForKey:
→ @"enabled_preference"];
```

Localization

Localization refers to the representation of currency, dates, times, and numbers for a regional variant of a language called a *locale*. For example, currency and dates are displayed differently for the United Kingdom than they are for the United States. *Internationalization*, on the other hand, refers to representing the text (and possibly other user interface elements) of your application in the end user's language. In this section, the term *localization* is used to mean both of these concepts.

You can localize numbers and dates by using the localization support of the **NSNumberFormatter** and **NSDateFormatter** classes. For example, to display a localized currency value, you can use this:

```
NSNumber *currencyValue = [NSNumber  
→ numberWithFloat:1000.23];  
  
NSNumberFormatter *formatter =  
→ [[NSNumberFormatter alloc] init];  
  
[formatter setNumberStyle:  
→ NSNumberFormatterCurrencyStyle];  
  
NSLog(@"Formatted value for locale  
→ %@ is: %@",  
→ [[formatter locale]  
→ localeIdentifier],  
  
[formatter stringFromNumber:  
→ currencyValue]);
```

The user's locale is automatically detected based on their phone settings.

```
1 "Acre" = "Aire";
2 "Square Inch" = "Pouce carré";
3 "Square Foot" = "Pied carré";
4 "Square Centimeter" = "Centimètre carré";
5 "Square Yard" = "Yard carré";
6 "Square Millimeter" = "Millimètre carré";
7 "Square Meter" = "Mètre carré";
8 "Square Kilometer" = "Kilomètre carré";
9 "Square Mile" = "Mile carré";
10 "Hectare" = "Hécte";
11 "Nector" = "Nector";
12 "Energy" = "Energie";
13 "Kilogram-meters" = "Kilogramm-metres";
14 "Foot-Pounds" = "Pied-livre";
15 "Kilogram-calories" = "Kg-calories";
16 "Electron-Volt" = "Electron-Volt";
17 "Kilowatt-hours" = "Kilowatt-heures";
18 "BTU" = "BTU";
19 "Newton-Meters" = "Newton-mètres";
20 "Joules" = "Joules";
21 "Newton-meters" = "Newton-mètres";
```

A A French strings file.



B Creating the en.lproj and fr.lproj directories in your project directory.



C Adding a strings file to your application.

Words and phrases, however, are unique to each language and must be treated differently. You must create a *strings file* for each language that you want your application to support **A**. Strings files consist of a list of key-value pairs, where the “key” is the source representation of the word or phrase (which will be in the language used by the developer) and the “value” is the localized version of the word or phrase.

To create a localized application:

1. Create a new view-based application, saving it as LocalizationExample.
2. Open the directory where you have just saved your project in Finder.
3. Create two directories: one representing English called en.lproj and one for French called fr.lproj **B**.

These directories are where you will be storing the localized resources for your project.

4. In Xcode, select File > New File.
5. In the Mac OS X section on the left of the New File dialog box, select the Other section, and choose Strings File from the file types **C**.
6. Click Next, and save the file as Localizable.strings in the en.lproj directory created in step 3.
7. Repeat step 6, creating a second Localizable.strings file, but this time saving it in the fr.lproj directory.

You should now see a new item in the Groups & Files pane in Xcode called Localizable.strings.

continues on next page

8. Expand this item, and you will see that Xcode has intelligently grouped both of your strings files together **D**.

9. Select the English strings file (“en”), and enter the following:

```
"HelloKey" = "Hello World!";
```

This is a key-value pair, and the key (**HelloKey**) has a value of “Hello World!”. Notice you end the entry with a semicolon.

10. Repeat step 9, but this time add the entry to the French strings file (“fr”):

```
"HelloKey" = "Bonjour Monde!";
```

Again, you use the same key, only this time you have specified a different, localized value.

11. Switch to LocalizationExample.m, uncomment the **viewDidLoad** method, and write the following code:

```
CGRect lblFrame = CGRectMake
→ (110,10,200,40);

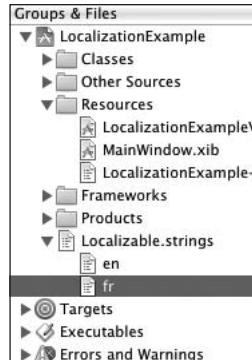
UILabel *lblLocalized = [[UILabel
→ alloc] initWithFrame:lblFrame];

lblLocalized.backgroundColor =
→ [UIColor clearColor];

lblLocalized.text = NSLocalizedString
→ String(@"HelloKey",nil);

[self.view addSubview:
→ lblLocalized];
```

Here you are just creating a label and adding it to your main view. Notice, however, that you set the text using the **NSLocalizedString** function. You pass this function a key for the localized text you want to display, and the function retrieves the localized text for you from the strings file.



D Xcode automatically groups together the strings files for a project.



E Selecting a different language.

12. Build and run your application.

You should see “Hello World!”

13. To see the French localization, switch to the Settings application, select General > International > Language, and choose French E. Switch back to your application, and you should see the text now in French.

You can localize your application’s settings and user interface in the same way by copying the <app>-Info.plist and XIB files to the localized en.lproj and fr.lproj directories.

TIP Notice that the second parameter of the NSLocalizedString function was not used. It’s there to allow you to provide a comment for each of your localized strings. You can then use the *genstrings* tool to parse your source code and automatically generate strings files for you. For information on how to use the *genstrings* tool, see the “Strings Files” section of the *Apple Internationalization Programming Topics* documentation.

TIP In the example here, you created localized strings using the language codes for English (“en”) and French (“fr”) to name the directories. For a complete list of available language codes, refer to the Unicode Consortium homepage at <http://www.unicode.org>.

Accessibility

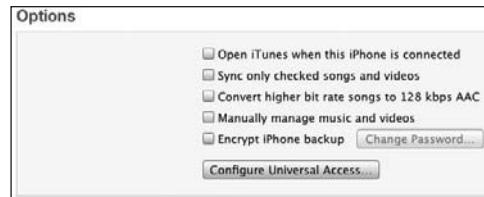
In the context of an iPhone application, *accessibility* means providing audible cues to describe the meaning and function of the user interface in your application. VoiceOver is an Apple technology that allows a user to interact with an iPhone without having to actually see the screen. The iPhone can speak to describe elements of the user interface, which buttons and controls are available, and which actions a user can take.

You can easily add VoiceOver support to your own applications by using the Accessibility API. All the standard UIKit controls and views are already accessible by default—you just need to provide the descriptive text for VoiceOver to speak.

To test your applications for Accessibility, you will need to enable VoiceOver support in iTunes.

To enable VoiceOver for your iPhone:

1. Launch iTunes, making sure your iPhone is attached to your computer.
2. Select your iPhone in the Devices section of the source list. Scroll to the bottom of the Summary tab, and click Configure Universal Access in the Options section **A**.
3. Make sure that VoiceOver is selected in the Universal Access pop-up window, and click OK **B**.



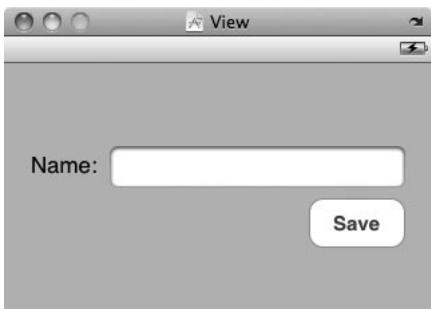
A Accessibility is enabled via iTunes.



B Enabling VoiceOver.



C iPhone accessibility settings.



D Creating the user interface for your accessible application.



E Select Device to install the application on your iPhone.

4. Press the Sync button to update your iPhone.

You can test that VoiceOver is working by selecting any interface element to hear a description. Note that your iPhone's controls function a little differently than normal when in VoiceOver mode. You must double-tap to select an item and use three fingers simultaneously to scroll. You can configure more of the Accessibility features on the iPhone by selecting Settings > General > Accessibility C.

Making your applications accessible

Now that you've enabled VoiceOver on your iPhone, you'll create a simple application with accessibility features enabled.

To create an accessible application:

1. Create a new view-based application, saving it as AccessibleExample.
2. Expand the Resources group, and double-click AccessibleExampleViewController.xib to open Interface Builder.
3. Drag a Label, Text Field, and Button onto your view.
4. Select the label, and change its text to Name.
5. Select the button, and change its title to Save D.
6. Save your interface, and go back to Xcode.

Since the iPhone Simulator does not support VoiceOver, you will need to select Device as the active SDK in Xcode E.

continues on next page

- With your iPhone connected to your computer, build and run the application to install it onto your iPhone.

You should be able to tap the label, text field, and button of the application to hear audio descriptions of them.

In this example, tapping the label made the iPhone speak its value, while the text field simply spoke the words “Text field, double-tap to edit.” Ideally, you would also want VoiceOver to describe the meaning of the text field. Similarly, when tapping the Save button, VoiceOver says, “Save, button.” Again, although descriptive, this may not be as much information as you want to convey to the user.

Next you’ll improve the values that VoiceOver uses when describing the functionality of your interface.

To improve the descriptions

VoiceOver uses in your applications:

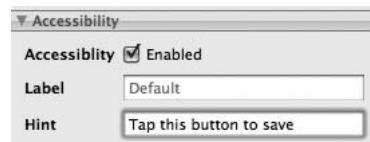
- In Interface Builder, select the text field, and make sure the Attributes inspector is showing.

You can show the Attributes inspector by pressing Command+1 or selecting it from the Tools menu.

- Select the Identity tab of the Attributes inspector, and expand the Accessibility section.
- In the Label text box, type “Enter your name here” as the value, and click the Save button on your interface **F**.
- On the Identity tab of the Attributes inspector, set the Hint text box to “Tap this button to save” **G**.



F Setting the Label accessibility attribute of the text field.



G Setting the Hint accessibility attribute of the Save button.

Traits	<input type="checkbox"/> Plays Sound	<input type="checkbox"/> Link
	<input type="checkbox"/> Image	<input type="checkbox"/> Static Text
<input checked="" type="checkbox"/> Button	<input type="checkbox"/> Search Field	
<input checked="" type="checkbox"/> User Interaction Enabled		
<input type="checkbox"/> Updates Frequently		
<input type="checkbox"/> Summary Element		
<input type="checkbox"/> Keyboard Key		
<input type="checkbox"/> Selected		

H The accessibility traits of the Save button in the previous example.

- Save your interface, and go back to Xcode. Build and run your application, again ensuring that iPhone Device is selected as the active SDK. Tap the text field and the Save button again to hear more meaningful descriptions spoken by VoiceOver.

Accessibility attributes

You've seen how simple it is to add basic VoiceOver support to your application and how to add more meaningful descriptions to the audio cues spoken. The Accessibility API defines five *attributes* to use when adding accessibility features to your applications:

- Label**—A short, simple word to describe the control, for example Name or Save.
- Hint**—A longer, more descriptive phrase to describe the meaning or action of a control, for example, "Tap to save."
- Traits**—A combination of one or more elements to describe the behavior of a control **H**.
- Value**—The value of the element or control. The label in the previous example had a value of Name. Generally, the Accessibility API will use this if the element's value can be changed or can't be easily described by the label.
- Frame**—The frame of the element onscreen. This is used by the accessibility API to report the control's location onscreen.

You can read more information on adding accessibility support to your applications, including how to do so programmatically (rather than using Interface Builder as you have in these examples), by referring to the *Accessibility Programming Guide for iPhone OS* in the developer documentation.

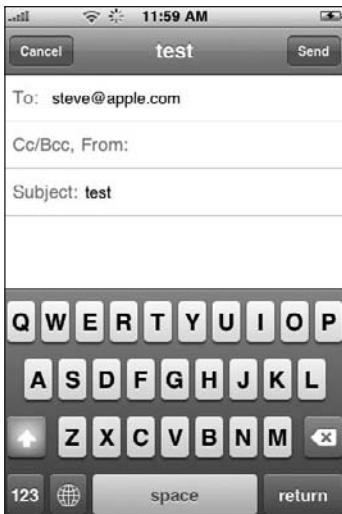
TIP In the previous examples, you had to install the application on your iPhone to test it. Although Interface Builder doesn't support VoiceOver, it does offer a tool called the Accessibility inspector to help when developing accessible applications. The Accessibility inspector appears as a floating window in your application's interface. Selecting any item on the interface shows its accessibility properties **I**. You can enable the Accessibility Inspector in the simulator by selecting Settings > General > Accessibility **I**.



I The Accessibility inspector can be used to test your application from the iPhone Simulator.



I Enabling the Accessibility inspector in the iPhone Simulator.



A Launching the Mail application.

Interapp Communication

Although the iPhone doesn't allow more than one application to run at a time, you can launch another application from your own, and you can share data between applications.

You can launch one application from another by using the `openURL:` method of the `UIApplication` class. For example, to open the Google homepage in the Safari application, you could write this:

```
NSURL *url = [NSURL URLWithString:  
→ @"http://google.com"];  
[[UIApplication sharedApplication]  
→ openURL:url];
```

The `http://` part here is called a *URL scheme* and identifies the application you want to launch.

URL schemes exist for several of the other native iPhone applications and can be used to launch them in a similar manner.

For example, to launch the Mail application A, you can use this:

```
NSURL *url = [NSURL URLWithString:  
→ @"mailto:steve@apple.com?subject=  
→ test"];  
[[UIApplication sharedApplication]  
→ openURL:url];
```

To launch the SMS application, you can write this:

```
NSURL *url = [NSURL URLWithString:  
→ @"sms:555-1234"];  
[[UIApplication sharedApplication]  
→ openURL:url];
```

To dial a phone number, you can use the following:

```
NSURL *url = [NSURL URLWithString:@"tel: //555-1234"];
[[UIApplication sharedApplication]
→ openURL:url];
```

To launch the Maps application and search for *pizza* **B**, you can use this:

```
NSURL *url = [NSURL URLWithString:
→ @"http://maps.google.com/maps?q=
→ pizza"];
[[UIApplication sharedApplication]
→ openURL:url];
```

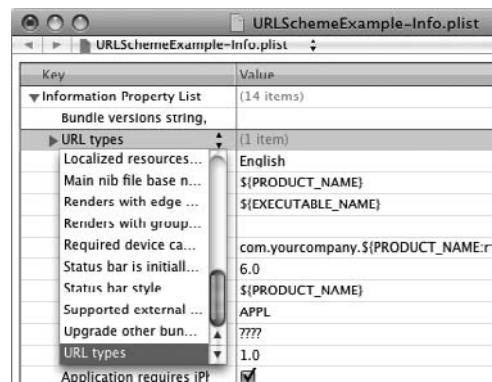
You can also use a URL scheme to launch your own application.

To create an application with a custom URL scheme:

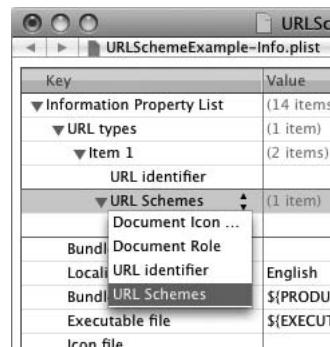
1. Create a new view-based application, saving it as `URLSchemeExample`.
2. In the Xcode Groups & Files pane, expand the Resource section, and select the `<app>-Info.plist` file.
3. Right-click the Information Property List key, and select Add Row. Select “URL types” from the list **C**.
4. Expand Item 1, right-click URL identifier, and again select Add Row. Select URL Schemes from the list **D**.



B Launching the Maps application and finding some pizza.



C Adding a URL type.



D Adding a URL scheme.

URLSchemaExample-Info.plist	
Key	Value
Information Property List	(14 items)
URL types	(1 item)
Item 1	(2 items)
URL identifier	
URL Schemes	(1 item)
Item 1	myapplication
Bundle versions string, Version	1.0
Localization native dev/English	English

E Setting the name of the URL scheme.

- Select Item 1, and set the value to myapplication **E**.

- Open **URLSchemaExampleView Controller.m**, uncomment the **viewDidLoad** method, and write the following:

```
[self.view setBackgroundColor:  
→ [UIColor redColor]];
```

- Build and run your application.

You should see a blank, red screen.

Your application doesn't do anything at the moment, but by running it (which installs the application on the iPhone or simulator), you have just registered the new URL scheme (**myapplication**) created in step 5.

- You can now launch this application from a different application by using the following code:

```
NSURL *url = [NSURL URLWithString:  
→ @"myapplication:"];  
[[UIApplication sharedApplication]  
→ openURL:url];
```

Sharing information between applications

When you launch an application using a URL scheme, the entire URL is passed to the application. This is how Safari knew to open the Google Web page in the earlier example.

To capture the URL, you need to implement the **application:didFinishLaunchingWithOptions:** method within the application delegate. The **launchOptions** parameter of this method is a dictionary whose contents will contain both the URL and an identifier for the source application that launched your application.

To respond to being launched via a URL scheme:

1. Open the URLschemeExampleApp Delegate.m file, and add the `application: didFinishLaunching`
→ `WithOptions:` delegate method.

```
if (launchOptions) {  
  
    NSString *sourceApp =  
    → [launch Options objectForKey:  
    → UIApplicationLaunchOptions  
    → Source ApplicationKey];  
  
    NSURL *url = [launchOptions  
    → objectForKey:UIApplication  
    → Launch OptionsURLKey];  
  
    NSString *msg = [NSString  
    → stringWithFormat:@"sourceApp:  
    → %@", url: %@", sourceApp, url];  
  
    UIAlertView *alert =  
    → [[UIAlertView alloc]  
    → initWithTitle:@"" message:msg  
    → delegate:self  
    → cancelButtonTitle:@"OK"  
    → otherButtonTitles:nil];  
  
    [alert show];  
    [alert release];  
}  
  
[self applicationDidFinish  
→ Launching: application];
```



F The application displaying the `launchOptions` values.

You first check to see whether the `launch Options` dictionary exists (it will be null if the application was launched normally), and then you extract two values from the dictionary:

UIApplicationLaunchOptionsSourceApplicationKey

`ApplicationKey` contains the bundle identifier for the source application that opened your application.

UIApplicationLaunchOptionsURLKey

`contains the complete URL (including the URL scheme itself).`

For testing, you then create an alert view and display these values. In a real-world application, you could parse the URL to extract any information passed to your application.

2. Build and run the application to install it onto the iPhone or simulator.

Code Listing 3.2 shows the updated code.

3. Again, switch to the other application, and change your code to this:

```
NSURL *url = [NSURL URLWithString:  
→ @"myapplication:message=  
→ helloworld"];  
  
[[UIApplication sharedApplication]  
→ openURL:url];
```

Your application will launch, and you should see an alert message similar to F.

Code Listing 3.2 The updated code to respond to being launched by a URL scheme.

```
#import "URLSchemeExampleAppDelegate.h"
#import "URLSchemeExampleViewController.h"

@implementation URLSchemeExampleAppDelegate

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    if (launchOptions) {
        NSString *sourceApp = [launchOptions objectForKey:UIApplicationLaunchOptionsSourceApplicationKey];
        NSURL *url = [launchOptions objectForKey:UIApplicationLaunchOptionsURLKey];

        NSString *msg = [NSString stringWithFormat:@"sourceApp: %@", sourceApp, url];
        UIAlertView *alert = [[UIAlertView alloc]
            initWithTitle:@""
            message:msg
            delegate:self
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];
        [alert show];
        [alert release];
    }

    [self applicationDidFinishLaunching:application];
    return YES;
}

- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}

@end
```

Using the pasteboard

Another useful way of passing information between applications is by using the *pasteboard*. Items copied *to* the pasteboard from one application can be copied *from* the pasteboard by another application.

You can get a reference to the pasteboard by writing this:

```
UIPasteboard *pasteboard =  
→ [UIPasteboard generalPasteboard];
```

You can then place a string on the pasteboard by using this:

```
pasteboard.string = @"Hello World";
```

After launching your app via a call to `openURL:`, you can then retrieve the string from the pasteboard by writing this:

```
UIPasteboard *pasteboard =  
→ [UIPasteboard generalPasteboard];  
  
NSString *myString =  
→ pasteboard.string;
```

The `UIPasteboard` class contains convenience properties like this for `NSString`, `UIImage`, `NSURL`, and `UIColor`.

You can add multiple items to the pasteboard by using the `setItems:` method. For example, to add two strings to the pasteboard, you can use this:

```
UIPasteboard *pasteboard =  
→ [UIPasteboard generalPasteboard];  
  
NSDictionary *item1 = [NSDictionary  
→ dictionaryWithObject:@"Hello World"  
→ forKey:@"public.utf8-plain-text"];  
  
NSDictionary *item2 = [NSDictionary  
→ dictionaryWithObject:@"Goodbye  
→ World" forKey:@"public.utf8-  
→ plain-text"];  
  
NSArray *items = [NSArray arrayWith  
→ Objects:item1,item2,nil];  
  
[pasteboard setItems:items];
```

Each item is first added to an array as a dictionary and then added to the pasteboard using the `setItems:` method. You can later retrieve this array from the pasteboard using the `items` property.

The key for each dictionary is a uniform type identifier (UTI). For a list of available UTIs, refer to *Uniform Type Identifiers Overview* in the developer documentation.

Code Listing 3.3 shows the code updated to retrieve a string from the pasteboard and display it in an alert view.

Code Listing 3.3 The code updated to use the pasteboard.

```
#import "URLSchemeExampleAppDelegate.h"
#import "URLSchemeExampleViewController.h"

@implementation URLSchemeExampleAppDelegate

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    if (launchOptions) {
        UIPasteboard *pasteboard = [UIPasteboard generalPasteboard];
        NSString *msg = pasteboard.string;

        UIAlertView *alert = [[UIAlertView alloc]
            initWithTitle:@""
            message:msg
            delegate:self
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];
        [alert show];
        [alert release];
    }

    [self applicationDidFinishLaunching:application];
    return YES;
}

- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}

@end
```

4

iPhone User Interface Elements

iOS offers a rich set of buttons, sliders, switches, and other user interface elements for you to use in creating your applications. These elements can be roughly divided into two main groups, views and controls.

Views provide the primary canvas and drawing functionality of your user interface. They also give your application the ability to handle touch events.

Controls extend upon this functionality and provide a way for users to interact with your application by defining what is known as the *target-action* mechanism: the ability for a control to send an *action* (method call) to a *target* (object) when an event (touch) occurs.

In this chapter, you'll look at the various views and controls available in iOS and examine how to use them.

All the examples use the View-based Application template, with the code running in the view controller.

In This Chapter

Views	112
Image Views	126
Scrolling	129
Labels	136
Progress and Activity Indicators	139
Alerts and Actions	142
Picker Views	146
Toolbars	152
Text	156
Web Views	164
Controls	170

Views

A **view** is the common name given to instances of **UIView**. You can think of a view as your application’s canvas; in other words, if you are adding UI elements to your iPhone’s interface, you are adding them to a view. All the UI elements discussed in this chapter are themselves subclasses of **UIView** and so inherit its properties and behavior.

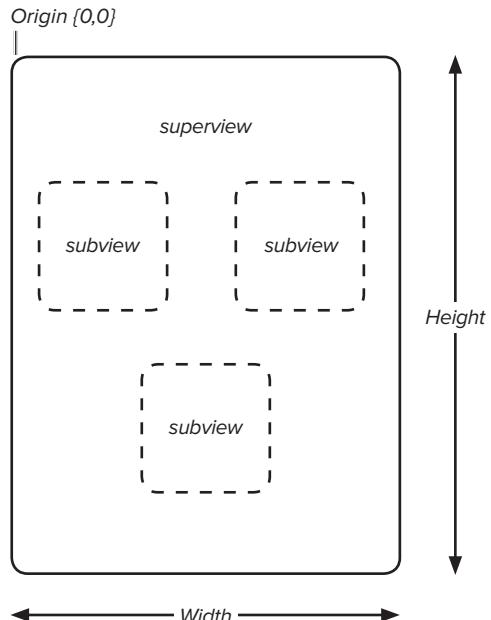
The root level of your iPhone application interface consists of a single **UIWindow** to which you would typically add one or more views to work with, instead of using **UIWindow** directly.

Since **UIView** is a subclass of **UIResponder**, it can receive touch events. For most views, you’ll receive only a single-touch event unless you set the **multipleTouchEnabled** property to **TRUE**. You can determine whether a view can receive touch events by modifying its **userInteractionEnabled** property. You can also force a view to be the only view to receive touch events by setting the **exclusiveTouch** property to **YES**. (For more information on working with touch events, see Chapter 7, “Touches, Shakes, and Orientation.”)

You can also nest views within each other in what’s known as the *view hierarchy*. Child views are known as *subviews*, and a view’s parent is its *superview*.

Frames

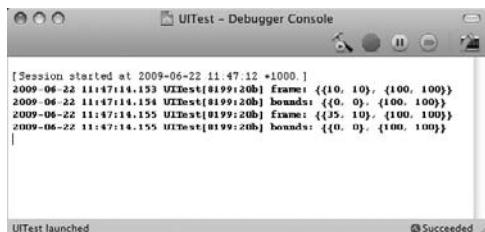
Views are represented by a rectangular region of the screen called a *frame*. The frame specifies the *origin* (*x*, *y*) and size (width, height) of the view, in relation to its parent superview. The origin of the coordinate system for all views is the upper-left corner of the screen **A**.



A Child views (subviews) are nested inside their parent view (superview). A view’s origin is at the top-left corner.



B Adding a subview to the view controller's main view.



C Console output after moving the view. Notice that although the frame changes, the bounds remain the same.

Code Listing 4.1 Creating a new view.

```
- (void)viewDidLoad {
    CGRect viewFrame = CGRectMake(10,10,100,100);
    UIView *myView = [[UIView alloc] initWithFrame:viewFrame];
    myView.backgroundColor = [UIColor blueColor];
    [self.view addSubview:myView];
    [myView release];
}
```

To add a view to your application:

1. Create a `CGRect` to represent the frame of the view, and pass it as the first parameter of the view's `initWithFrame:` method:

```
CGRect viewFrame = CGRectMake
→ (10,10,100,100);
UIView *myView = [[UIView alloc]
→ initWithFrame:viewFrame];
```

Here you are creating a view that is inset 10 pixels from the top left of its superview and that has a width and height of 100 pixels.

2. Since the view is transparent by default, set its background color before adding it to the view controller's existing view **B**:

```
myView.backgroundColor =
→ [UIColor blueColor];
[[self view] addSubview:myView];
```

Code Listing 4.1 shows the completed code.

TIP To improve performance, set your `UIView`'s `opaque` property to `YES` wherever possible.

Bounds

A view's *bounds* are similar to its frame, but the location and size are relative to the view's *own* coordinate system rather than those of its superview. In the previous example, the frame's origin is `{10,10}`, but the origin of its bounds is `{0,0}`. (The width and height for both the frame and the bounds are the same.)

The console output illustrates this **C**: After moving the view 25 pixels in the x direction (using the view's `center` property), the frame origin is now `{35,10}`, whereas the bounds origin remains at `{0,0}`.

Let's say you want to create a view so that it completely fills its superview. A common mistake is to use the frame of the superview.

If you tried to run this code in your application, you'd see a gap at the top of the subview **D**.

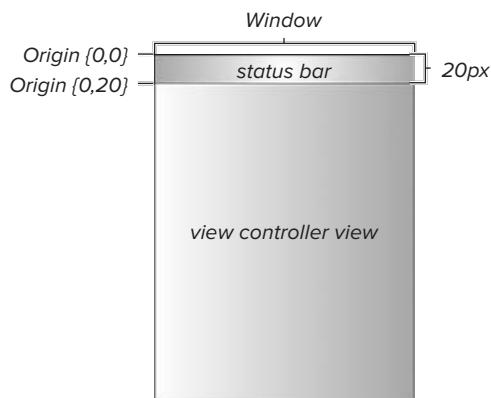
Recall that, in the project, the **UIWindow** is at the top level. The **UIWindow** has two subviews: the status bar and the main view 20 pixels below **E**. The origin of the frame of the main view is actually **{0,20}**. (Remember, a view's frame is in relation to its superview's coordinate system.)

The solution to this problem is to use the *bounds* of the superview (**Code Listing 4.2**), which causes the view to correctly fill its superview.

TIP You can use the `NSStringFromCGRect()` function to convert a `CGRect` into an `NSString`, making it useful for logging `CGRects` to the console via `NSLog()`. Other useful functions when dealing with `CGRects` are `NSStringFromCGPoint()` and `NSStringFromCGSize()`.



D Setting the frame incorrectly by using the frame of the superview. Notice the gap at the top.



E The enclosing **UIWindow** contains both the status bar and the view controller's view as subviews. Notice how the controller's view has an origin starting at **{0,20}** for its `frame`.

Code Listing 4.2 Initializing the view's frame with its superview's bounds.

```
- (void)viewDidLoad {
    UIView *myView = [[UIView alloc] initWithFrame:
        [self.view bounds]];
    myView.backgroundColor = [UIColor blueColor];
    [self.view addSubview:myView];
    [myView release];
}
```

Animation

Many properties of a view can be animated, including its **frame**, **bounds**, **backgroundColor**, **alpha** level, and more. You'll now look at some simple examples that illustrate additional view concepts.

To animate your view:

1. Retrieve the center of the view controller's main view:

```
CGPoint frameCenter =  
→ self.view.center;
```

2. Create a view, set its background color, and, just as you did earlier, add it to the main view:

```
float width = 50.0;  
float height = 50.0;  
  
CGRect viewFrame = CGRectMake  
→ (frameCenter.x-width,  
→ frameCenter.y-height,width*2,  
→ height*2);  
  
UIView *myView = [[UIView alloc]  
→ initWithFrame:viewFrame];  
  
myView.backgroundColor =  
→ [UIColor blueColor];  
[[self view] addSubview:myView];
```

Here you are positioning your view in the center of its superview and giving it a width and height of 50 pixels.

3. Set up an *animation block*:

```
[UIView beginAnimations:nil  
→ context: NULL];  
  
[UIView setAnimationDuration:1.0];
```

An animation block is a wrapper around a set of changes to animatable properties. In this example, the animation lasts for one second.

continues on next page

4. Resize the view:

```
viewFrame = CGRectMake(viewFrame,  
→ -width, -height);  
  
[myView setFrame:viewFrame];
```

The `CGRectInset()` function takes a source rectangle and then creates a smaller or larger rectangle with the same center point. In this example, a negative value for the width and height creates a larger rectangle.

5. Close the animation block:

```
[UIView commitAnimations];
```

This will cause all of the settings within the animation block to be applied.

6. Build and run the application.

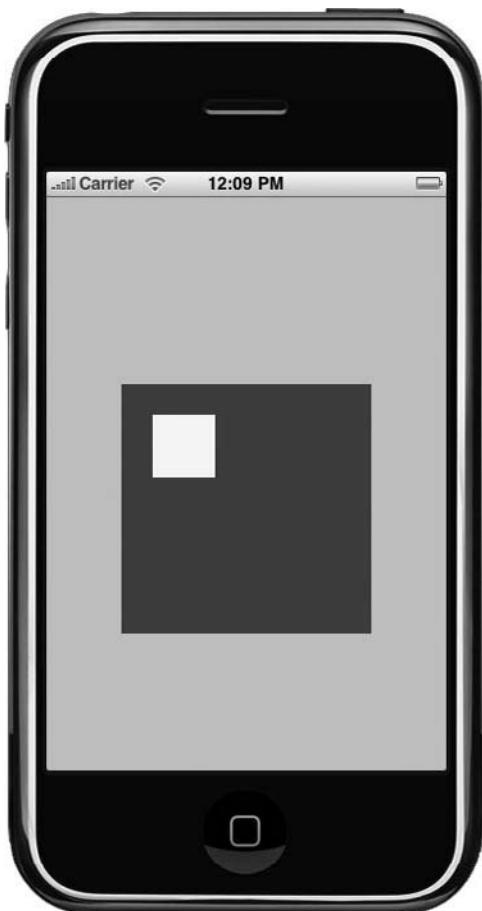
You should see the view grow in size over a period of one second. **Code Listing 4.3** shows the completed code.

TIP Try changing the `setAnimationDuration:` line to see how you can affect the speed of the animation.

TIP Try setting some other properties on the view within the animation block (such as `backgroundColor`) to see what effect they have.

Code Listing 4.3 Animating a view.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    CGPoint frameCenter = self.view.center;  
    float width = 50.0;  
    float height = 50.0;  
  
    CGRect viewFrame = CGRectMake(frameCenter.x-width,frameCenter.y-height, width*2, height*2);  
  
    UIView *myView = [[UIView alloc] initWithFrame:viewFrame];  
    myView.backgroundColor = [UIColor blueColor];  
  
    [[self view] addSubview:myView];  
  
    [UIView beginAnimations:nil context:NULL];  
  
    [UIView setAnimationDuration:1.0];  
    viewFrame = CGRectMakeInset(viewFrame, -width, -height);  
    [myView setFrame:viewFrame];  
  
    [UIView commitAnimations];  
  
    [myView release];  
}
```



F Animating multiple views without using an autoresizing mask. Notice how the new subview ends up in the top-left corner of its superview.

Autosizing

When a view changes size or position, you often want any subviews contained within the view to change size or position in proportion to their containing superview. You can accomplish this by using a view's *autoresizing mask*. Now let's add a second subview inside the view you created in the previous exercise.

To add a subview:

1. Create a `CGRect` for the subview's frame, again using the shortcut `CGRectInset()` function:

```
CGRect subViewFrame =  
    CGRectMakeInset(myView.bounds,  
    width/2.0, height/2.0);  
  
UIView *mySubview =  
    [[UIView alloc]  
    initWithFrame:subViewFrame];  
  
mySubview.backgroundColor =  
    [UIColor yellowColor];  
  
[myView addSubview:mySubview];
```

This time, the positive width and height values for the `CGRectInset` function make the new view smaller. To make them stand out, give it a different background color.

2. Build and run the application F. The new subview starts off in the center of its superview, but then it remains "pinned" to its initial location as the animation progresses and ends up in the top-left corner.

continues on next page

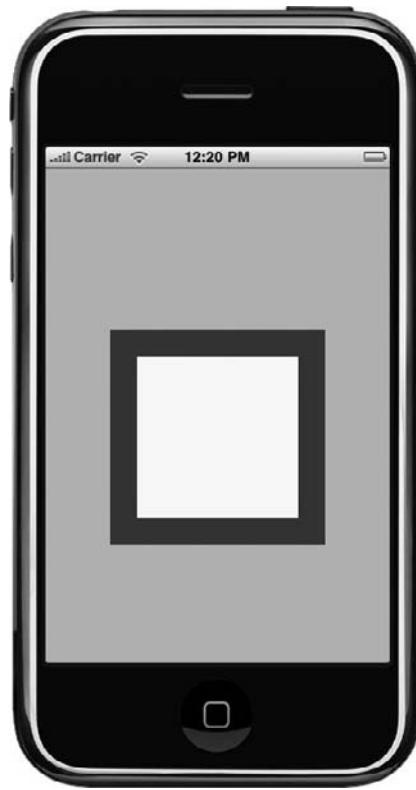
Code Listing 4.4 shows this code updated to use an autoresizing mask. Notice how you set all four margins of the subview using the bitwise OR operator (the `|` symbol) between the constant values (**Table 4.1**). Notice also that even though the animation is specified on the superview, the subview still animates automatically **G**.

3. You can visually set the `autoresizingMask` property in the size pane of the Inspector window in Interface Builder **H**.

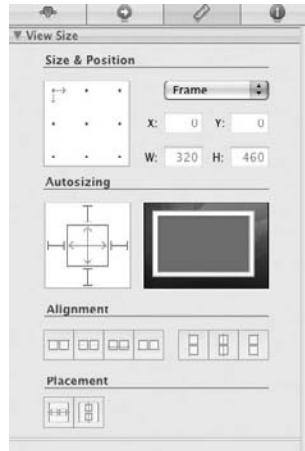
Custom drawing

By default, the visual representation of a `UIView` is fairly boring. You can manipulate the size, background color, and alpha levels of the view, but not much else.

Luckily, it's relatively simple to create your own `UIView` subclasses where you can implement custom drawing behavior. To see how this might be done, you'll now learn how to create a `UIView` subclass with rounded corners.



G Using the autoresizing mask property, the subview remains in the center of its superview during an animation.



H Setting the autoresizing mask in Interface Builder.

Code Listing 4.4 Using an autoresizing mask.

```
- (void)viewDidLoad {
    [super viewDidLoad];

    CGPoint frameCenter = self.view.center;
    float width = 50.0;
    float height = 50.0;

    CGRect viewFrame = CGRectMake(frameCenter.x-width,frameCenter.y-height, width*2, height*2);
    UIView *myView = [[UIView alloc] initWithFrame:viewFrame];
    myView.backgroundColor = [UIColor blueColor];

    //create subview
    CGRect subViewFrame = CGRectMakeInset(myView.bounds, width/2.0, height/2.0);
    UIView *mySubview = [[UIView alloc] initWithFrame:subViewFrame];
    mySubview.backgroundColor = [UIColor yellowColor];

    //set autoresizing mask
    mySubview.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin
        | UIViewAutoresizingFlexibleRightMargin
        | UIViewAutoresizingFlexibleTopMargin
        | UIViewAutoresizingFlexibleBottomMargin;

    [myView addSubview:mySubview];
    [[self view] addSubview:myView];

    //animate resize
    [UIView beginAnimations:nil context:NULL];

    [UIView setAnimationDuration:1.0];
    viewFrame = CGRectMakeInset(viewFrame, -width, -height);
    [myView setFrame:viewFrame];

    [UIView commitAnimations];

    [mySubview release];
    [myView release];
}
```

TABLE 4.1 Available autoresizingMask values

Value	Description
UIViewAutoresizingNone	The view does not resize.
UIViewAutoresizingFlexibleLeftMargin	The view resizes by expanding or shrinking in the direction of the left margin.
UIViewAutoresizingFlexibleWidth	The view resizes by expanding or shrinking its width.
UIViewAutoresizingFlexibleRightMargin	The view resizes by expanding or shrinking in the direction of the right margin.
UIViewAutoresizingFlexibleTopMargin	The view resizes by expanding or shrinking in the direction of the top margin.
UIViewAutoresizingFlexibleHeight	The view resizes by expanding or shrinking its height.
UIViewAutoresizingFlexibleBottomMargin	The view resizes by expanding or shrinking in the direction of the bottom margin.

To create a custom rounded-corner view:

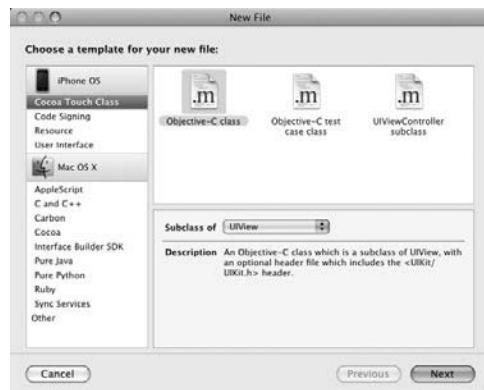
1. In Xcode, select File > New File. Create a new Objective-C class, making sure that “Subclass of” is set to `UIView` **I**. Save the file as `roundedCornerView`.
2. Open `roundedCornerView.m`, and modify your code to look like **Code Listing 4.5**.
3. Open `UITestViewController.m`, and replace all instances of `UIView` with `roundedCornerView`. Don’t forget to also import the header file for `roundedCornerView.h` at the top of the file. **Code Listing 4.6** shows the updated code.
4. Build and run your application to see the result with rounded corners for the views **I**.

As you can see, custom drawing happens in the `drawRect:` method of `roundedCornerView`. You set a couple of variables here—one to determine the width of the line you will be drawing and another to determine the color.

5. By setting the color to the superview’s background color, you are essentially “erasing” any time you draw in the subview.

```
float lineWidth = 10.0;  
UIColor *parentColor = [[self  
→ superview] backgroundColor];
```

continues on page 122



I Adding a custom class to draw the rounded corner view.



J In the updated application, the views now have rounded corners.

Code Listing 4.5 The `roundedCornerView` class.

```
@implementation roundedCornerView

- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame])
    {
        self.opaque = TRUE;
    }
    return self;
}

void CGContextStrokeCorners(CGContextRef ctx, CGRect rect) {
    int radius = 12;

    CGFloat xOrigin = rect.origin.x;
    CGFloat yOrigin = rect.origin.y;

    CGFloat xMiddle = CGRectGetMidX(rect);
    CGFloat yMiddle = CGRectGetMidY(rect);

    CGFloat width = rect.size.width;
    CGFloat height = rect.size.height;

    CGContextBeginPath(ctx);

    CGContextMoveToPoint(ctx, xOrigin, yMiddle);
    CGContextAddArcToPoint(ctx, xOrigin, yOrigin, xMiddle, yOrigin, radius);
    CGContextAddArcToPoint(ctx, width, yOrigin, width, yMiddle, radius);
    CGContextAddArcToPoint(ctx, width, height, xMiddle, height, radius);
    CGContextAddArcToPoint(ctx, xOrigin, height, xOrigin, yMiddle, radius);

    CGContextClosePath(ctx);
    CGContextStrokePath(ctx);
}

- (void)drawRect:(CGRect)rect {
    float lineWidth = 10.0;
    UIColor *parentColor = [[self superview] backgroundColor];

    CGContextRef ctx = UIGraphicsGetCurrentContext();
    CGContextSetStrokeColorWithColor(ctx, parentColor.CGColor);
    CGContextSetLineWidth(ctx, lineWidth);

    //draw corners
    CGContextStrokeCorners(ctx,rect);
}

- (void)dealloc {
    [super dealloc];
}

@end
```

6. Now you get a reference to the current graphics context and set the pen color and width.

A *graphics context* is a special type that represents the current drawing destination, in this case the custom view's contents.

```
CGContextRef ctx =  
    → UIGraphicsGetCurrentContext();  
  
CGContextSetStrokeColorWithColor  
    → (ctx, parentColor.CGColor);  
  
CGContextSetLineWidth(ctx,  
    → lineWidth);
```

7. Finally, call a custom function that draws a line around the outside of the view, rounding at each corner:

```
CGContextStrokeCorners(ctx,rect);
```

Code Listing 4.6 Replacing regular views with the custom class.

```
#import "UITestViewController.h"  
#import "roundedCornerView.h"  
  
@implementation UITestViewController  
  
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    CGPoint frameCenter = self.view.center;  
    float width = 50;  
    float height = 50;  
  
    CGRect viewFrame = CGRectMake(frameCenter.x-width, frameCenter.y-height, width*2, height*2);  
    roundedCornerView *myView = [[roundedCornerView alloc] initWithFrame:viewFrame];  
  
    myView.backgroundColor = [UIColor blueColor];  
    [[self view] addSubview:myView];  
  
    CGRect subViewFrame = CGRectMakeInset(myView.bounds, width/2, height/2);  
    roundedCornerView *mySubview = [[roundedCornerView alloc] initWithFrame:subViewFrame];  
    mySubview.autoresizingMask = UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;  
    mySubview.backgroundColor = [UIColor yellowColor];  
    [myView addSubview:mySubview];  
  
    [UIView beginAnimations:nil context:NULL];  
    [UIView setAnimationDuration:1.0];  
  
    viewFrame = CGRectMakeInset(viewFrame, -width, -height);  
    [myView setFrame:viewFrame];  
  
    [UIView commitAnimations];  
  
    [mySubview release];  
    [myView release];  
}
```

Transforms

You've already looked at resizing a view by increasing the width and height of its frame. Another way to perform the same task is by using a *transform*.

A transform maps the coordinates system of a view from one set of points to another. Transformations are applied to the *bounds* of a view. In addition to scaling, you can also rotate and move a view using transforms.

To resize your view using a scale transform:

- Add the following code to your application:

```
CGAffineTransform scale =  
→ CGAffineTransformMakeScale  
→ (2.0,2.0);  
  
myView.transform = scale;
```

This creates a scale transform, doubling both the width and the height of your view.

or

Transforms can also be used to move views by using a *translate* transform:

```
CGAffineTransform translate =  
→ CGAffineTransformMakeTranslation  
→ (50,50);  
  
myView.transform = translate;
```

This would cause a view to move by 50 pixels along both the x- and y-axes.

or

Finally, you can apply a *rotation* transform to rotate your views:

```
CGAffineTransform rotate =  
→ CGAffineTransformMakeRotation  
→ (radiansForDegrees(180));  
  
myView.transform = rotate;
```

Because rotations are specified in radians, you use a function to convert from degrees.

To apply both a rotation transform and a scale transform to your view:

1. Update the code to look like the following:

```
CGAffineTransform scale =  
→ CGAffineTransformMakeScale  
→ (2.0,2.0);  
  
CGAffineTransform rotate =  
→ CGAffineTransformMakeRotation  
→ (radiansForDegrees(180));  
  
CGAffineTransform myTransform =  
→ CGAffineTransformConcat  
→ (scale,rotate);  
  
myView.transform = myTransform;
```

Note how you can combine transformations using the `CGAffineTransform Concat()` function.

Code Listing 4.7 shows the completed code.

2. Build and run your application ⑩.

Your view should rotate and scale at the same time.

TIP You no longer need to set the `autoresizingMask` property of the subview because the transform is applied to the view and its subviews at the same time.

TIP You can return a view to its original state by setting its `transform` property to `CGAffineTransformIdentity`.



⑩ The view both rotating and scaling.

Code Listing 4.7 Rotating and scaling the view.

```
CGFloat radiansForDegrees(CGFloat degrees)
{
    return (M_PI * degrees / 180.0);
}

- (void)viewDidLoad {
    [super viewDidLoad];

    CGPoint frameCenter = self.view.center;
    float width  = 50;
    float height = 50;

    CGRect viewFrame = CGRectMake(frameCenter.x-width, frameCenter.y-height, width*2, height*2);
    roundedCornerView *myView = [[roundedCornerView alloc] initWithFrame:viewFrame];

    myView.backgroundColor = [UIColor blueColor];
    [[self view] addSubview:myView];

    CGRect subViewFrame = CGRectMakeInset(myView.bounds, width/2, height/2);
    roundedCornerView *mySubview = [[roundedCornerView alloc] initWithFrame:subViewFrame];
    mySubview.backgroundColor = [UIColor yellowColor];
    [myView addSubview:mySubview];

    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:1.0];

    CGAffineTransform scale = CGAffineTransformMakeScale(2.0,2.0);
    CGAffineTransform rotate = CGAffineTransformMakeRotation(radiansForDegrees(180));
    CGAffineTransform myTransform = CGAffineTransformConcat(scale,rotate);
    myView.transform = myTransform;

    [UIView commitAnimations];
    [mySubview release];
    [myView release];
}
```

Image Views

The `UIImageView` class extends `UIView` to provide support for displaying images. Its default initializer, `initWithImage:`, takes a `UIImage` as its only parameter (Code Listing 4.8):

```
UIImage *anImage = [UIImage  
→ imageNamed:@"myImage.png"];  
  
UIImageView *myImageView =  
→ [[UIImageView alloc]  
→ initWithFrame:anImage];
```

Note that `initWithImage:` automatically adjusts the frame of the new image view to match the width and height of the image assigned **A**.

If you resize the image view, you can see that the image automatically scales to fit **B**:

```
CGSize viewSize =  
→ myImageView.bounds.size;  
  
//shrink width 50%  
  
viewSize.width = viewSize.width  
→ * 0.5;  
  
//keep height the same  
  
viewSize.height = viewSize.height;  
  
  
CGRect newFrame = CGRectMake  
→ (0,0,viewSize.width,  
→ viewSize.height);  
  
[myImageView setFrame:newFrame];
```

Code Listing 4.8 Creating an image view.

```
- (void)viewDidLoad {  
  
    UIImage *anImage = [UIImage imageNamed:  
        @"myImage.png"];  
    UIImageView *myImageView = [[UIImageView alloc]  
        initWithFrame:anImage];  
  
    [self.view addSubview:myImageView];  
    [myImageView release];  
}
```



A The image displaying a graphic.



B Resizing the image view.



➊ Resizing the image view while maintaining its aspect ratio.

You can control scaling behavior by the **contentMode** property of **UIView**, which defaults to **UIViewContentModeScaleToFill**.

For example, to maintain the aspect ratio of the image, you would write this:

```
myImageView.contentMode =  
    UIViewContentModeScaleAspectFit;
```

In the resulting image, note that although the *image* itself is scaled, the *image view* still has the same bounds ➋. Any part of the bounds not rendered in the image will be transparent.

Animating images

UIImageView lets you animate over an array of images, which is handy for creating progress animations. **Code Listing 4.9** shows the code updated to animate over three images.

Code Listing 4.9 Animating over an array of images.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    NSArray *arrImages = [[NSArray alloc] initWithObjects:  
        [UIImage imageNamed:@"apple.png"],  
        [UIImage imageNamed:@"apple2.png"],  
        [UIImage imageNamed:@"apple3.png"],nil];  
  
    CGRect viewFrame = CGRectMake(0,0,200,200);  
    UIImageView *myImageView = [[UIImageView alloc] initWithFrame:viewFrame];  
  
    [myImageView setAnimationImages:arrImages];  
  
    [myImageView setAnimationRepeatCount:0];  
    [myImageView setAnimationDuration:0.5];  
  
    [self.view addSubview:myImageView];  
    [myImageView startAnimating];  
  
    [arrImages release];  
    [myImageView release];  
}
```

To animate over an image:

1. Create the image view, and set its frame:

```
CGRect viewFrame = CGRectMake  
→ (0,0,200,200);
```

```
UIImageView *myImageView =  
→ [[UIImageView alloc]  
→ initWithFrame:viewFrame];
```

2. Create and set the image array:

```
NSArray *arrImages =  
→ [[NSArray alloc] initWithObjects:  
  
[UIImage imageNamed:  
→ @"apple.png"],  
  
[UIImage imageNamed:  
→ @"apple2.png"],  
  
[UIImage imageNamed:  
→ @"apple3.png"],nil];  
  
[myImageView  
→ setAnimationImages:arrImages];  
  
[arrImages release];
```

3. You can control the speed of the animation (in seconds) and number of times the animation is repeated. The default is 0, making the animation loop indefinitely:

```
[myImageView  
→ setAnimationDuration:0.5];  
  
[myImageView setAnimation  
→ RepeatCount:0];
```

4. To begin the animation, add the following:

```
[myImageView startAnimating];
```

5. To stop the animation, you call `stopAnimating`.

TIP For simplicity, the previous examples use `imageNamed:` to create the images. Although convenient, this method creates autoreleased objects that can't be manually released in a low-memory situation. So, it's usually wiser to use something like the `initWithContentsOfFile:` method and manually allocate/release your images.

Scrolling

Often your views will be larger than the visible area, and you need a way to scroll. For this, you use the **UIScrollView** class.

A scroll view acts as a container for a larger subview, allowing you to pan around the subview by touching the screen. Vertical and horizontal scroll bars indicate the position in the subview.

Code Listing 4.10 shows an example of using a scroll view.

To create a scroll view:

1. Set the frame as usual:

```
CGRect scrollFrame = CGRectMake  
→ (20,90,280,280);  
  
UIScrollView *scrollView =  
→ [[UIScrollView alloc]  
→ initWithFrame:scrollFrame];
```

2. Create an image view, assigning it an image that is larger than the scroll view:

```
UIImage *bigImage = [UIImage  
→ imageNamed:@"appleLogo.jpg"];  
  
UIImageView *largeImageView =  
→ [[UIImageView alloc]  
→ initWithImage:bigImage];
```

continues on next page

Code Listing 4.10 Using a scroll view.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    CGRect scrollFrame = CGRectMake(20,90,280,280);  
    UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:scrollFrame];  
  
    UIImage *bigImage = [UIImage imageNamed:@"appleLogo.jpg"];  
    largeImageView = [[UIImageView alloc] initWithImage:bigImage];  
  
    [scrollView addSubview:largeImageView];  
    scrollView.contentSize = largeImageView.frame.size; //important!  
  
    [self.view addSubview:scrollView];  
    [scrollView release];  
}
```

3. Add the image view to the scroll view, and set the `contentSize` property of the scroll view:

```
[scrollView addSubview:  
    →largeImageView];  
  
scrollView.contentSize =  
    →largeImageView.frame.size;
```

This is an important step: If you don't tell the scroll view how large its subview is, it won't know how to scroll at all.

4. Finally, add the scroll view to the main view:

```
[self.view addSubview:scrollView];
```

You'll now see the scroll view with horizontal and vertical scroll bars indicating the current position in the image view

A. You can hide these scroll bars using the `showsHorizontalScrollIndicator` and `showsVerticalScrollIndicator` properties.

TIP If you play around with the previous code, you'll notice that if you scroll quickly to the edge of the subview, the scroll view actually moves a little too far before springing back. This behavior is controlled by the `bounce` property. You can restrict bouncing to the x- or y-axis using the `alwaysBounceHorizontal` and `alwaysBounceVertical` properties, or you can disable it entirely by setting `bounce` to NO.

Zoom

You can also zoom in and out of an image using a scroll view. The `minimumZoomScale` and `maximumZoomScale` properties control the scale by which you can zoom in and out. By default, both of these properties are set to the same value (`1.0`), which disables zooming. You must implement one of the `UIScrollViewDelegate` methods to return the view that is being zoomed.



A Using a scroll view to pan around a large image.



- ❸ The page control indicating the total number of pages and the current page as a series of dots at the bottom of the iPhone's screen.

To enable zooming:

1. Add the `UIScrollViewDelegate` protocol in the controller.h file:

```
@interface UITestViewController :  
    UIViewController  
    <UIScrollViewDelegate>
```

2. Update the scroll view code to allow you to zoom out by `1/2` and in by `2x`:

```
scrollView.minimumZoomScale = 0.5;  
scrollView.maximumZoomScale = 2.0;  
scrollView.delegate = self;
```

You've also set the delegate to be the controller (`self`).

3. Implement the `viewForZoomingInScrollView`: delegate method, and return the image view. **Code Listing 4.11** shows the updated code.

Paging

Scroll views support the *paging* of their content—the ability to add multiple subviews as “pages” and then scroll between them as you might turn the pages of a book. Adding a `UIPageControl` will provide a visual depiction of your current page ❸.

Code Listing 4.11 Adding zoom to the scroll view.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    CGRect scrollFrame = CGRectMake(20,90,280,280);  
    UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:scrollFrame];  
    scrollView.minimumZoomScale = 0.5;  
    scrollView.maximumZoomScale = 2.0;  
    scrollView.delegate = self;  
  
    UIImage *bigImage = [UIImage imageNamed:@"appleLogo.jpg"];  
    largeImageView = [[UIImageView alloc] initWithImage:bigImage];  
  
    [scrollView addSubview:largeImageView];  
    scrollView.contentSize = largeImageView.frame.size; //important!  
  
    [self.view addSubview:scrollView];  
    [scrollView release];  
}  
  
- (UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView  
{  
    return largeImageView;  
}
```

To create a page control:

1. Update the code to remove the image from the scroll view, and set some new properties:

```
float pageControlHeight = 18.0;  
int pageCount = 3;
```

```
CGRect scrollViewRect =  
→ [self.view bounds];  
  
scrollViewRect.size.height -=  
→ pageControlHeight;  
  
myScrollView =  
→ [[UIScrollView alloc]  
→ initWithFrame:scrollViewRect];  
  
myScrollView.pagingEnabled = YES;
```

The **pagingEnabled** property turns paging on for the scroll view.

2. Since you have three pages, set the **contentView** of the scroll view to be three times wider than its frame. You'll also turn off the scroll view indicators:

```
myScrollView.contentSize =  
→ CGSizeMake(scrollViewRect.size.  
→ width * pageCount,1);  
  
myScrollView.showsHorizontal  
→ ScrollIndicator = NO;  
  
myScrollView.showsVertical  
→ ScrollIndicator = NO;  
  
myScrollView.delegate = self;
```

3. Set up the page control by creating a frame below the scroll view, and add a target to the page control so that when it is tapped, it will call the **changePage:** method:

```

CGRect pageViewRect =
→ [self.view bounds];
pageViewRect.size.height =
→ pageControlHeight;
pageViewRect.origin.y =
→ scrollViewRect.size.height;

myPageControl = [[UIPageControl
→ alloc] initWithFrame:
→ pageViewRect];
myPageControl.backgroundColor =
→ [UIColor blackColor];
myPageControl.numberOfPages =
→ pageCount;
myPageControl.currentPage = 0;
[myPageControl addTarget:self
→ action:@selector(changePage:)
→ forControlEvents:UIControlEvent
→ ValueChanged];

```

4. Call the **createPages** method by adding three **UIViews** side by side to the scroll view to represent the three pages.

5. Set the **backgroundColor** property of the views.

In a real-world application, these would be more interesting! At this stage, your scroll view will actually work, but you need some more work to get the page control to reflect the current page.

6. Implement the **scrollViewDidScroll:** delegate method:

```

CGFloat pageWidth = sender.frame.
→ size.width;
int page = floor((sender.contentOffset
→ -Offset.x - pageWidth / 2) /
→ pageWidth) + 1;
myPageControl.currentPage = page;

```

continues on next page

This simply does some math to calculate your current page during the scroll and then updates the page control accordingly.

7. Finally, implement the **changePage:** method called when the page control is tapped:

```
int page = myPageControl.  
→ currentPage;  
  
CGRect frame = myScrollView.frame;  
frame.origin.x = frame.size.width  
→ * page;  
  
frame.origin.y = 0;  
  
[myScrollView scrollRectToVisible:  
→ frame animated:YES];
```

This scrolls the scroll view horizontally based on the page you have selected in the page control. **Code Listing 4.12** shows the completed code.

Code Listing 4.12 Implementing a page control.

```
UIScrollView *myScrollView;  
UIPageControl *myPageControl;  
  
@implementation UITestViewController  
  
- (void)loadScrollViewWithPage:(UIView *)page  
{  
    int pageCount = [[myScrollView subviews] count];  
  
    CGRect bounds = myScrollView.bounds;  
    bounds.origin.x = bounds.size.width * pageCount;  
    bounds.origin.y = 0;  
    page.frame = bounds;  
    [myScrollView addSubview:page];  
}  
  
- (void)createPages  
{  
    CGRect pageRect = myScrollView.frame;  
  
    //create pages  
    UIView *page1 = [[UIView alloc] initWithFrame:pageRect];  
    page1.backgroundColor = [UIColor blueColor];  
    UIView *page2 = [[UIView alloc] initWithFrame:pageRect];  
    page2.backgroundColor = [UIColor redColor];  
    UIView *page3 = [[UIView alloc] initWithFrame:pageRect];  
    page3.backgroundColor = [UIColor greenColor];
```

code continues on next page

Code Listing 4.12 continued

```
//add to scrollview
[self loadScrollViewWithPage:page1];
[self loadScrollViewWithPage:page2];
[self loadScrollViewWithPage:page3];

//cleanup
[page1 release];
[page2 release];
[page3 release];
}

-(void)viewDidLoad {
    [super viewDidLoad];

    float pageControlHeight = 18.0;
    int pageCount = 3;

    CGRect scrollViewRect = [self.view bounds];
    scrollViewRect.size.height -= pageControlHeight;

    //create scrollview
    myScrollView = [[UIScrollView alloc] initWithFrame:scrollViewRect];
    myScrollView.pagingEnabled = YES;
    myScrollView.contentSize = CGSizeMake(scrollViewRect.size.width * pageCount,1);
    myScrollView.showsHorizontalScrollIndicator = NO;
    myScrollView.showsVerticalScrollIndicator = NO;
    myScrollView.delegate = self;

    //create pageview
    CGRect pageViewRect = [self.view bounds];
    pageViewRect.size.height = pageControlHeight;
    pageViewRect.origin.y = scrollViewRect.size.height;

    myPageControl = [[UIPageControl alloc] initWithFrame:pageViewRect];
    myPageControl.backgroundColor = [UIColor blackColor];
    myPageControl.numberOfPages = pageCount;
    myPageControl.currentPage = 0;
    [myPageControl addTarget:self action:@selector(changePage:) forControlEvents:UIControlEventValueChanged];

    //create pages
    [self createPages];

    //add to main view
    [self.view addSubview:myScrollView];
    [self.view addSubview:myPageControl];

    //cleanup
    [myPageControl release];
    [myScrollView release];
}

-(void)scrollViewDidScroll:(UIScrollView *)sender
{
    CGFloat pageWidth = sender.frame.size.width;
    int page = floor((sender.contentOffset.x - pageWidth / 2) / pageWidth) + 1;
    myPageControl.currentPage = page;
}

-(void)changePage:(id)sender
{
    int page = myPageControl.currentPage;

    // update the scroll view to the appropriate page
    CGRect frame = myScrollView.frame;
    frame.origin.x = frame.size.width * page;
    frame.origin.y = 0;
    [myScrollView scrollRectToVisible:frame animated:YES];
}
```

Labels

Instances of the **UILabel** class display a read-only view that can contain one or more lines of text. For example, to create a simple label and set its **text**, **font**, **textColor**, and **backgroundColor** properties (Code Listing 4.13), use this:

```
myLabel.backgroundColor =
→ [UIColor clearColor];
myLabel.textColor =
→ [UIColor redColor];
myLabel.font = [UIFont
→ systemFontOfSize: 18.0];
myLabel.text = @"Hello World!";
```

By default, a label is rendered as black text on a white background. You can also set a font by name:

```
myLabel.font = [UIFont fontWithName:
→ @"Verdana" size:18.0];
```

Table 4.2 shows the available fonts you can use.

If you don't specify a font size, the label will automatically reduce the font to fit the text within the label's frame. You can control how small the font gets with the **minimumFontSize** property, and you can disable this behavior entirely with the **adjustsFontSizeToFitWidth** property.

To add a shadow to a label's text, you could write the following:

```
myLabel.shadowColor =
→ [UIColor darkGrayColor];
myLabel.shadowOffset =
→ CGSizeMake(1.0,1.0);
```

The **shadowOffset** controls set how far on the x- and y-axes from the label's text the shadow is drawn. The default is {0,-1}.

Code Listing 4.13 Creating a label.

```
- (void)viewDidLoad {
    CGRect labelFrame = CGRectMake(10,10,200,44);
    UILabel *myLabel = [[UILabel alloc] initWithFrame:
    labelFrame];
    myLabel.backgroundColor = [UIColor clearColor];
    myLabel.textColor = [UIColor redColor];
    myLabel.font = [UIFont systemFontOfSize:18.0];
    myLabel.text = @"Hello World!";
    [self.view addSubview:myLabel];
    [myLabel release];
}
```

TABLE 4.2 Fonts available on the iPhone

Family	Name
American Typewriter	AmericanTypewriter, AmericanTypewriter-Bold
AppleGothic	AppleGothic
Arial	ArialMT, Arial-BoldMT, Arial-BoldItalicMT, Arial-ItalicMT
Arial Hebrew	ArialHebrew, ArialHebrew-Bold
Arial Rounded MT Bold	ArialRoundedMTBold
Arial Unicode MS	ArialUnicodeMS
Courier	Courier, Courier-BoldOblique, Courier-Oblique, Courier-Bold
Courier New	CourierNewPS-BoldMT, CourierNewPS-ItalicMT, CourierNewPS-BoldItalicMT, CourierNewPSMT
DB LCD Temp	DBLCDTempBlack
Geeza Pro	GeezaPro-Bold, GeezaPro
Georgia	Georgia-Bold, Georgia, Georgia-BoldItalic, Georgia-Italic
Hiragino Kaku Gothic ProN	HiraKakuProN-W6, HiraKakuProN-W3
Heiti J	STHeitiJ-Medium, STHeitiJ-Light
Heiti K	STHeitiK-Medium, STHeitiK-Light
Heiti SC	STHeitiSC-Medium, STHeitiSC-Light
Heiti TC	STHeitiTC-Light, STHeitiTC-Medium
Helvetica	Helvetica-Oblique, Helvetica-BoldOblique, Helvetica, Helvetica-Bold
Helvetica Neue	HelveticaNeue, HelveticaNeue-Bold
Marker Felt	MarkerFelt-Thin
Times New Roman	TimesNewRomanPSMT, TimesNewRomanPS-BoldMT, TimesNewRomanPS-BoldItalicMT, TimesNewRomanPS-ItalicMT
Thonburi	Thonburi-Bold, Thonburi
Trebuchet MS	TrebuchetMS-Italic, TrebuchetMS, TrebuchetMS-BoldItalic, TrebuchetMS-Bold
Verdana	Verdana-Bold, Verdana-BoldItalic, Verdana, Verdana-Italic
Zapfino	Zapfino

The **textAlignment** property allows you to align the label text to the left (the default), center, or right.

The **lineBreakMode** property controls how a label wraps text that is too wide to fit within its frame. You can specify whether you want the text to be word or character wrapped, clipped, or truncated at the start, end, or middle of the text.

To display multiple lines of text in a label, use the **numberOfLines** property and the **\n** newline escape character:

```
myLabel.numberOfLines = 2;  
myLabel.text = @"Hello World\nSecond  
→ line";
```

The height of the label's **frame** property needs to be tall enough to accommodate the number of lines of text you specify, or the text will be wrapped using the value defined in the **lineBreakMode** property (Code Listing 4.14).

TIP Setting the **numberOfLines** property to 0 will make the label dynamically set the line count.

Code Listing 4.14 Setting various properties of a label.

```
- (void)viewDidLoad {  
    CGRect labelFrame = CGRectMake(10,10,200,44);  
  
    UILabel *myLabel = [[UILabel alloc] initWithFrame:  
                        labelFrame];  
  
    myLabel.backgroundColor = [UIColor clearColor];  
    myLabel.textColor = [UIColor redColor];  
    myLabel.font = [UIFont fontWithName:@"Verdana"  
                           size:18.0];  
    myLabel.numberOfLines = 2;  
    myLabel.text = @"Hello World!\nSecond line";  
  
    myLabel.shadowColor = [UIColor darkGrayColor];  
    myLabel.shadowOffset = CGSizeMake(1.0,1.0);  
  
    [self.view addSubview:myLabel];  
    [myLabel release];  
}
```



A A progress view at 33 percent completion.

Progress and Activity Indicators

When performing tasks that may take some time, you often need to provide some kind of visual feedback to your users. If you know how long the task will take to complete, you can use a progress indicator to show the user how much of the task has been performed and how much still has to run. If you are unable to determine the duration of the task, use a “busy” indicator (such as the beach ball or hourglass on OS X).

iOS provides classes for showing both progress and activity.

Indicating progress

When you want to show the progress of a task, use **UIProgressView**, a very simple class, consisting of only two properties.

You create a progress view and set its style using the **initWithProgressViewStyle:** method:

```
UIProgressView *myProgressView =  
→ [[UIProgressView alloc]  
→ initWithProgressViewStyle:  
→ UIProgressViewStyleDefault];
```

The indicator appears as a horizontal bar that fills from left to right to show completion A. This is controlled by the **progress** property, using a value between **0.0** (not started) and **1.0** (completed):

```
[myProgressView setProgress:0.33];
```

Although you set the frame of the progress view, the maximum height of a progress view is 9 pixels, so any larger value will be ignored.

Code Listing 4.15 shows an example of using **UIProgressView** with the progress updated in a timer to simulate a long-running task.

TIP The other progress bar style, **UIProgressViewStyleBar**, also uses a horizontal bar indicator but is more suitable for using in a toolbar (explained in the following section).

Showing activity

For tasks of an indeterminate duration, you can use the **UIActivityIndicatorView** class, represented by an animated “spinner” graphic **B**.

Code Listing 4.15 Updating the progress view.

```
NSTimer *timer;  
  
@implementation UITestViewController  
  
- (void)updateProgress:(NSTimer *)sender  
{  
    UIProgressView *progress = [sender userInfo];  
  
    //have we completed?  
    if (progress.progress == 1.0)  
        [timer invalidate];  
    else  
        progress.progress += 0.05;  
}  
  
- (void)viewDidLoad {}  
  
[super viewDidLoad];  
  
UIProgressView *myProgressView = [[UIProgressView alloc] initWithProgressViewStyle:UIProgressViewStyleDefault];  
  
CGRect progressFrame = CGRectMake(10,100,300,25);  
[myProgressView setFrame:progressFrame];  
  
[myProgressView setProgress:0.0];  
  
[self.view addSubview:myProgressView];  
  
[myProgressView release];  
  
//create timer  
timer = [[NSTimer scheduledTimerWithTimeInterval:0.1  
                                              target:self  
                                             selector:@selector(updateProgress:)  
                                               userInfo:myProgressView  
                                              repeats:YES] retain];  
}
```



B An activity indicator view.

Similar to a progress view, you can create an activity indicator using the `initWithActivityIndicatorStyle:` method:

```
UIActivityIndicatorView  
→ *myActivityView =  
→ [[UIActivityIndicatorView alloc]  
→ initWithActivityIndicatorStyle:  
→ UIActivityIndicatorViewStyleWhite];
```

The default size of an activity view is a 21-pixel square. If you use the `UIActivityIndicatorViewStyleWhiteLarge` style, this increases to a 36-pixel square.

Unlike the progress view, however, the frame property controls both the height and the width of the view. For activity views larger than 36 pixels, it's best to use the larger style so the image won't become pixelated.

The activity view will initially be invisible. Calling the `startAnimating` method shows the activity view and causes the spinner graphic to animate:

```
[myActivityView startAnimating];
```

Calling `stopAnimating` will stop the spinner animation, but you need to remember to set the `hidesWhenStopped` property if you want the activity view to hide ([Code Listing 4.16](#)).

Code Listing 4.16 Creating an activity indicator view.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    [self.view setBackgroundColor:[UIColor blackColor]];  
  
    UIActivityIndicatorView *myActivityView = [[UIActivityIndicatorView alloc]  
                                             initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhiteLarge];  
  
    CGRect activityFrame = CGRectMake(130,100,50,50);  
    [myActivityView setFrame:activityFrame];  
  
    [myActivityView startAnimating];  
  
    [self.view addSubview:myActivityView];  
  
    [myActivityView release];  
}
```

Alerts and Actions

Often in your applications you'll want to present a message to your users. Perhaps you want to alert them about an error or present them with options for a given action. As an iPhone developer, you handle these situations using *alert views* and *action sheets*.

Alerting users

To display an alert message, use the **UIAlertView** class. You define a **title**, **message**, and **delegate**, and then you configure buttons to be shown in the view.

To display an alert view:

1. First, create a simple alert view **A**:

```
UIAlertView *myAlert =  
→ [[UIAlertView alloc]  
→ initWithTitle:@"title"  
→ message:@"message"  
→ delegate:nil  
→ cancelButtonTitle:@"OK",  
→ otherButtonTitles:nil]  
→ [myAlert show];
```

2. Using the **otherButtonTitles** property, you can create the same alert view with up to four additional buttons **B**:

```
UIAlertView *myAlert =  
→ [[UIAlertView alloc]  
→ initWithTitle:@"title"  
→ message:@"message"  
→ delegate:nil  
→ cancelButtonTitle:@"OK"  
→ otherButtonTitles:@"button1",  
→ @"button2", @"button3",  
→ @"button4",nil];
```



A A bare-bones alert view.



B An alert view with several buttons added.

If you have only two buttons in your alert view, they will be displayed side by side. Otherwise, buttons are added from top to bottom, with the cancel button always being at the very bottom. If you don't need the message or title text, there is room for five buttons in addition to the cancel button.

3. You can add buttons after creating your alert view by using the **addButtonWithTitle:** method:

```
[myAlert addButtonWithTitle:  
→ @"new button"];
```

4. To determine which button is tapped, set the delegate, and implement the **alertView:clickedButtonAtIndex:** delegate method (Code Listing 4.17).

The **buttonIndex** parameter tells you which button was tapped, starting with the cancel button at index 0. Alert views close automatically when a button is tapped.

TIP Using the **dismissWithClickedButtonIndex:animated:** method, you can programmatically close an alert view without the user having to tap a button. This might be useful in a situation where you want to show an alert for a short time and then hide it automatically.

TIP Another way to alert the user is by making the iPhone vibrate by calling the function **AudioServicesPlayAlertSound(kSystemSoundID_Vibrate)**. You'll need to add the **AudioToolbox** framework to your project for this to work.

Code Listing 4.17 Display an alert view.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"title"  
                                                message:@"message"  
                                               delegate:self  
                                         cancelButtonTitle:@"OK"  
                                         otherButtonTitles:nil];  
  
    [myAlert addButtonWithTitle:@"new button"];  
    [myAlert addButtonWithTitle:@"another button"];  
  
    [myAlert show];  
    [myAlert release];  
}  
  
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {  
    NSLog(@"you clicked button: %i",buttonIndex);  
}
```

Confirming an action

When presenting the user with a number of options, you can use a **UIActionSheet**.

To create an action sheet:

1. An action sheet is created in a similar way to an alert view **C**:

```
UIActionSheet *mySheet =  
    [[UIActionSheet alloc]  
        initWithTitle:@"Do you really  
        want to delete?" delegate:nil  
        cancelButtonTitle:@"No"  
        destructiveButtonTitle:@"Yes"  
        otherButtonTitles:nil];  
  
[mySheet showInView:self.view];
```

2. Define titles for three types of button.

The *cancel* button is generally used to dismiss the action sheet.

The *destructive* button acts as the confirmation of the action and is usually shown in red to indicate its importance.

The *other* buttons are similar to the alert view and allow you to add more buttons.

Setting any of these parameters to **nil** prevents the button type from showing.

3. Set the delegate, and implement the **actionSheet:clickedButtonAtIndex:** method, which is called when a button is tapped.

You can compare the **buttonIndex** parameter to the action sheet's **cancelButtonIndex** and **destructiveButtonIndex** properties to determine which button was tapped.

Code Listing 4.18 shows the code updated with some of these options.



C An action sheet is “pinned” to the bottom of the screen, and it contains only a title.

Code Listing 4.18 Adding more options to the action sheet.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIActionSheet *mySheet = [[UIActionSheet alloc] initWithTitle:@"Email Deletion Options"
                                                       delegate:self
                                              cancelButtonTitle:@"Cancel"
                                              destructiveButtonTitle:@"Delete Everything"
                                              otherButtonTitles:@"All Read Email", @"Spam Only", nil];
    mySheet.actionSheetStyle = UIActionSheetStyleBlackOpaque;
    [mySheet showInView:self.view];
    [mySheet release];
}
- (void)actionSheet:(UIActionSheet *)actionSheet clickedButtonAtIndex:(NSInteger)buttonIndex {
    BOOL cancelClicked = actionSheet.cancelButtonIndex == buttonIndex;
    BOOL destructiveButtonClicked = actionSheet.destructiveButtonIndex == buttonIndex;
    NSLog(@"%@", @"button with index %i clicked (cancel:%i, destructive:%i)", buttonIndex, cancelClicked, destructiveButtonClicked);
}
```

Action sheets vs. alert views

Action sheets are functionally similar to alert views, with a number of important differences:

- Action sheets are attached to a view. The code used in the previous exercise attaches the action sheet to the controller's main view.
- You can optionally show the alert sheet from a tab bar or a toolbar using the **showFromTabBar:** and **showFromToolbar:** methods.
- Action sheets do not have a **message** property; they have a single **title** property.
- You can change how the action sheet looks by using the **actionSheetStyle** property. In addition to the default style, you can give your action sheet a black transparent or opaque style. Setting the style to **UIActionSheetStyleAutomatic** will give your action sheet the same appearance as the bottom bar if one exists.

Picker Views

The **UIPickerView** class allows users to “spin” a wheel-type control to select one or more values. Each picker view consists of one or more *components* consisting of one or more *rows*. Each component can be spun independently of the others. For example, the picker view can be used to select a date value with three components in the control, representing the month, day, and year **A**.

The number of components and rows in a picker view is determined by its *datasource*, an object that adopts the **UIPickerViewDataSource** protocol. The display and selection of the picker view content is handled by the *delegate*, which adopts the **UIPickerViewDelegate** protocol (the datasource and the delegate can be the same object).

To create a simple picker view:

1. Add the protocol declarations to your interface definition:

```
@interface UITestViewController :  
    → UIViewController  
    → <UIPickerViewDataSource,  
    → UIPickerViewDelegate>
```

2. Create a picker view, and add it to the main view (**Code Listing 4.19**):

```
CGRect pickerFrame =  
    → CGRectMake(0,120,0,0);  
  
UIPickerView *myPicker =  
    → [[UIPickerView alloc]  
    → initWithFrame:pickerFrame];  
myPicker.dataSource = self;  
myPicker.delegate = self;  
  
[self.view addSubview:myPicker];
```



A A picker view being used to select a date.

Picker views are always 320 pixels by 216 pixels in size and cannot be resized.

3. The **showsSelectionIndicator** property creates a translucent bar across the control to indicate the selected row.
4. At a minimum, you need to implement two data source methods.

numberOfComponentsInPickerView:

returns the number of segments or components in the picker view. In this example, you want a single component, so return the value **1**.

pickerView:numberOfRowsInComponent:

returns the number of rows for each component. Again, ignore the component parameter (since you have only a single component), and return the number of rows.

continues on next page

Code Listing 4.19 A bare-bones picker view implementation.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    CGRect pickerFrame = CGRectMake(0,120,0,0);

    UIPickerView *myPicker = [[UIPickerView alloc] initWithFrame:pickerFrame];
    myPicker.dataSource = self;
    myPicker.delegate = self;
    myPicker.showsSelectionIndicator = YES;

    [self.view addSubview:myPicker];
    [myPicker release];
}

-(NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}

-(NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:(NSInteger)component {
    return 10;
}

-(NSString *)pickerView:(UIPickerView *)thePickerView titleForRow:(NSInteger)row forComponent:(NSInteger)component {
    return [NSString stringWithFormat:@"Row %i",row];
}
```

5. Implement the delegate `pickerView:titleForRow:forComponent:` method, returning an `NSString` representation of the current row **B**:

```
return [NSString stringWithFormat:  
    @"Row %i",row];
```

The picker view can display much more interesting data than this simple example. Components can be of different widths and, rather than just simple text, can actually have entire views embedded within them **C**.

To enhance the picker view:

1. After calling the `initComp1` and `initComp2` methods to create some sample data, update the `numberOfComponentsInPickerView:` method to return two components (one for each of the sample arrays). Also, update the `pickerView:numberofRowsInComponent:` method to return the size of each array:

```
if (component == 0)  
    return [comp1 count];  
  
else  
    return [comp2 count];
```

The arrays here contain different numbers of elements; in other words, components do not need the same number of rows.

2. Define a new delegate method, `pickerView:widthForComponent:`, and set the widths of the components to different values:

```
if (component == 0)  
    return 100.0;  
  
else  
    return 200.0;
```



B The picker view shows the sample data.



C The updated picker view. Not only do the two components display different content, but they also have different widths and numbers of items.

3. Implement the **pickerView:viewForRow:forComponent:reusingView:** delegate, returning either an image view or a label. This method allows you to embed almost any view subclass in a picker view component.
4. Finally, in the **pickerView:didSelectRow:inComponent:** delegate, log the selected row and component to the console.

When you spin the picker view, this method isn't fired until the scrolling animation ends. **Code Listing 4.20** shows the updated code.

Code Listing 4.20 The updated picker view.

```

NSMutableArray *comp1;
NSMutableArray *comp2;

@implementation UITestViewController

-(void)initComp1
{
    comp1 = [[NSMutableArray alloc] init];
    UIImageView *imgView;

    imgView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"ca.png"]];
    [comp1 addObject:imgView];
    [imgView release];

    imgView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"gb.png"]];
    [comp1 addObject:imgView];
    [imgView release];

    imgView = [[UIImageView alloc] initWithImage:[UIImage imageNamed:@"us.png"]];
    [comp1 addObject:imgView];
    [imgView release];
}

-(void)initComp2
{
    comp2 = [[NSMutableArray alloc] init];
    UILabel *lbl;

    lbl = [[UILabel alloc] initWithFrame:CGRectMake(0,0,100,44)];
    lbl.backgroundColor = [UIColor clearColor];
    lbl.text = @"Red";
    [comp2 addObject:lbl];
    [lbl release];

    lbl = [[UILabel alloc] initWithFrame:CGRectMake(0,0,100,44)];
    lbl.backgroundColor = [UIColor clearColor];
    lbl.text = @"Blue";
    [comp2 addObject:lbl];
    [lbl release];
}

```

code continues on next page

Code Listing 4.20 *continued*

```
- (void)viewDidLoad {
    [super viewDidLoad];
    //create some sample data
    [self initComp1];
    [self initComp2];
    CGRect pickerFrame = CGRectMake(0,120,0,0);
    UIPickerView *myPicker = [[UIPickerView alloc] initWithFrame:pickerFrame];
    myPicker.dataSource = self;
    myPicker.delegate = self;
    myPicker.showsSelectionIndicator = YES;
    [self.view addSubview:myPicker];
    [myPicker release];
}
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 2;
}
- (NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:(NSInteger)component {
    if (component == 0)
        return [comp1 count];
    else
        return [comp2 count];
}
- (CGFloat)pickerView:(UIPickerView *)pickerView widthForComponent:(NSInteger)component
{
    if (component == 0)
        return 100.0;
    else
        return 200.0;
}
- (UIView *)pickerView:(UIPickerView *)pickerView
    viewForRow:(NSInteger)row
    forComponent:(NSInteger)component
    reusingView:(UIView *)view {
    if (component == 0)
        return [comp1 objectAtIndex:row];
    else
        return [comp2 objectAtIndex:row];
}
- (void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)row inComponent:(NSInteger)component {
    NSLog(@"row: %i, component:%i",row,component);
}
```

Picking dates and times

iOS also has a special version of a picker, **UIDatePicker**, geared toward picking dates as well as times. The **datePickerMode** property determines the style of the picker **D**.

Since the date picker is localized, it will automatically display dates and times in the format of the device locale. You can, however, override these settings to display dates and times for other locales.

You can set properties for start and end dates (for the date-style pickers) and for minute and countdown values (for the time-style pickers).

UIDatePicker is not actually a subclass of **UIPickerView**. It is a **UIControl** subclass that has a custom **UIPickerView** as a subview. This means that you use the target-action mechanism to manage the selection of values. As with other controls, you set the action:

```
[myPicker addTarget:self action:@selector(pickerChanged:) forControlEvents:UIControlEventValueChanged];
```

The date picker creates a **UIControlEventValueChanged** event when a date or time is selected (Code Listing 4.21).

Code Listing 4.21 Implementing a date picker.

```
- (void)pickerChanged:(id)sender {
    NSLog(@"%@", [sender date]);
}

- (void)viewDidLoad {
    CGRect pickerFrame = CGRectMake(0,120,0,0);
    UIDatePicker *myPicker = [[UIDatePicker alloc]
        initWithFrame:pickerFrame];
    [myPicker addTarget:self
        action:@selector(pickerChanged:)
        forControlEvents:UIControlEventValueChanged];
    [self.view addSubview:myPicker];
    [myPicker release];
}
```



D A date picker with the default style of **UIDatePickerModeDateAndTime** lets you pick both the date and the time.

Toolbars

You can create toolbars in iPhone applications using the **UIToolbar** class. A toolbar usually spans the entire width of the display and is aligned to either the top or the bottom of the screen **A**.

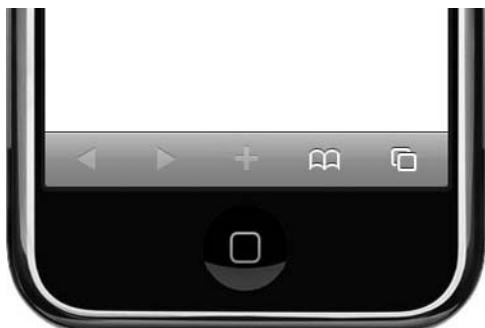
To create a toolbar:

- As with many other views, you can create a toolbar with the **initWithFrame:** method. Use the size of the main view to calculate the y position of the toolbar. This is important since you may not know the orientation of the iPhone and want the toolbar to sit at the bottom of the screen.

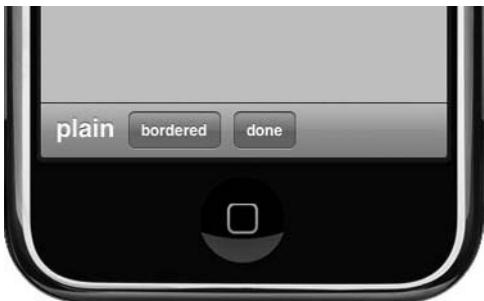
```
CGSize viewSize =  
→ self.view.frame.size;  
  
float toolbarHeight = 44.0;  
  
CGRect toolbarFrame = CGRectMake  
→ (0,viewSize.height-toolbarHeight,  
→ viewSize.width,toolbarHeight);  
  
UIToolbar *myToolbar =  
→ [[UIToolbar alloc] initWithFrame:  
→ toolbarFrame];
```

- Set the **autoresizingMask** property of the toolbar to ensure that it stays in the same position (in this case, aligned to the bottom of the screen) even if the user rotates their iPhone.

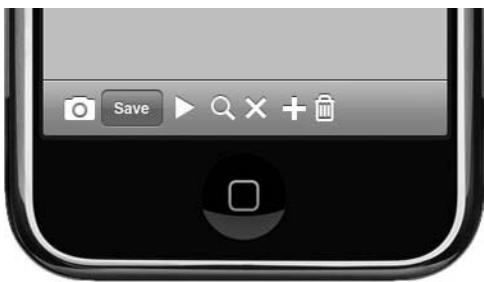
```
myToolbar.autoresizingMask =  
→ UIViewAutoresizingFlexible  
→ Width | UIViewAutoresizing  
→ FlexibleLeftMargin | UIView  
→ AutoresizingFlexibleRight  
→ Margin | UIViewAutoresizing  
→ FlexibleTopMargin;
```



A Most of the controls sit on the toolbar in Safari.



B A toolbar showing the three toolbar item styles for the `initWithTitle:style:target:action:` method.



C Some of the available system item styles.

3. You can change the color and translucency of the toolbar using the `tintColor` and `translucent` properties:

```
myToolbar.tintColor =  
→ [UIColor redColor];  
myToolbar.translucent = YES;
```

Toolbar items

Buttons you add to a toolbar are known as *toolbar items* and are created using the `UIBarButtonItem` class. Several types of buttons are available, and you can create them in several ways:

- The simplest way to create a button with some title text is by using the `initWithTitle:style:target:action:` method B. Use the `target` and `action` parameters to indicate which method to call when the button is pressed.
- Similarly, the `initWithImage:style:target:action:` method lets you create a button with an image instead of text. The button will automatically resize its width to that of the image.
- You can create a button from your own custom `UIView` subclass using the `initWithCustomView:` method. However, you must set the `target` and `action` properties manually.
- The final way is to use the `initWithBarButtonSystemItem:target:action:` method.

iOS offers a set of predefined buttons, known as *system items*, to ensure your application adheres to the iPhone interface guidelines. Use them whenever possible.

There are system items for play, pause, and stop buttons, as well as for search, trash, and camera C. For a complete list of system items available, refer to the `UIBarButtonItem` type in the developer documentation.

You will often use two particular system item types: `UIBarButtonSystemItemFlexibleSpace` and `UIBarButtonSystemItemFixedSpace`. Both are not visible and represent spaces on a toolbar. The flexible-space item lets you force a button to the other side of the toolbar, while the fixed-space item simply lets you add a space between buttons. You can set the `width` property of a fixed-space item to determine how wide you want the space to be.

Once you've created these buttons, add them to an `NSArray` and then use the `setItems:` method of the `UIToolbar` to add them to the toolbar itself. The optional `animated:` parameter allows you to have buttons fade in as they are added to the toolbar.

Code Listing 4.22 shows the updated code, with buttons of various types and a flexible-space item being used to push a button to the right side of the toolbar. If you try rotating the phone, you will notice that the toolbar and buttons correctly align themselves regardless of orientation **D**.



D The toolbar has correctly sized itself with the iPhone in landscape mode. The trash toolbar item is aligned to the right.

Code Listing 4.22 Creating several different types of toolbar items.

```
- (void)buttonClick:(id)sender {
    NSLog(@"you clicked button: %@",[sender title]);
}

- (void)viewDidLoad {
    [super viewDidLoad];

    CGSize viewSize = self.view.frame.size;
    float toolbarHeight = 44.0;
    CGRect toolbarFrame = CGRectMake(0,viewSize.height-toolbarHeight,viewSize.width,toolbarHeight);

    UIToolbar *myToolbar = [[UIToolbar alloc] initWithFrame:toolbarFrame];
    myToolbar.autoresizingMask = UIViewAutoresizingFlexibleWidth
        | UIViewAutoresizingFlexibleLeftMargin
        | UIViewAutoresizingFlexibleRightMargin
        | UIViewAutoresizingFlexibleTopMargin;

    UIBarButtonItem *button1 = [[UIBarButtonItem alloc] initWithTitle:@"button 1"
                                                               style:UIBarButtonItemStylePlain target:self
                                                               action:@selector(buttonClick:)];
    UIBarButtonItem *button2 = [[UIBarButtonItem alloc] initWithTitle:@"button 2"
                                                               style:UIBarButtonItemStyleBordered
                                                               target:self
                                                               action:@selector(buttonClick:)];
    UIBarButtonItem *button3 = [[UIBarButtonItem alloc] initWithImage:[UIImage imageNamed:@"apple_icon.png"]
                                                               style:UIBarButtonItemStyleBordered
                                                               target:self
                                                               action:@selector(buttonClick:)];
    UIBarButtonItem *flexButton = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                                                               target:nil
                                                               action:nil];
    UIBarButtonItem *trashButton = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemTrash
                                                               target:self
                                                               action:@selector(buttonClick:)];

    NSArray *buttons = [[NSArray alloc] initWithObjects:button1,button2,button3, flexButton,trashButton,nil];

    //cleanup
    [button1 release];
    [button2 release];
    [button3 release];
    [flexButton release];
    [trashButton release];

    [myToolbar setItems:buttons animated:NO];
    [buttons release];
    [self.view addSubview:myToolbar];
    [myToolbar release];
}
```

Text

For entering text into your applications, iOS provides two classes, **UITextField** and **UITextView**. Both allow the user to enter and edit text using an onscreen keyboard and support features such as cut/copy and paste, spell check, and more, but the two classes function differently.

To create a text field:

1. You can use the **UITextField** class to enter small amounts of text, such as user names, passwords, or search terms. This field is limited to a single line of text.
2. As with most other views, you use the **initWithFrame:** method to create them:

```
CGRect textRect = CGRectMake  
→ (10,10,300,20);  
  
UITextField *myTextField =  
→ [[UITextField alloc]  
→ initWithFrame:textRect];  
  
myTextField.backgroundColor =  
→ [UIColor whiteColor];
```

This also sets the background color of the text field; otherwise, it's transparent by default. Text fields also don't have a border by default.

3. Use the **borderStyle** property to choose from four different styles **A**.

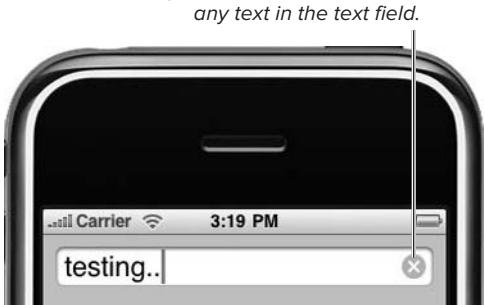
The **UITextBorderStyleRoundedRect** style has a white background and will ignore the **backgroundColor** property. If you set a custom **UIImage** as the **background**, the **borderStyle** property will be ignored.

4. You can set the text font, color, and alignment to apply to the entire text field.

Text fields do not support the styling of individual text elements.



A Border styles available for text fields.



- B Press the clear button to remove any text in the text field.



- C Two of the keyboards available by setting the `keyboardType` property of a text field.

5. You can set your text field to automatically resize the font to accommodate larger text:

```
myTextField.font =  
→ [UIFont systemFontOfSize:22.0];  
myTextField.adjustsFontSizeTo  
→ FitWidth = YES;  
myTextField.minimumFontSize = 2.0;
```

This example sets the initial font size as 22 and then tells the text field to automatically shrink the font to a minimum size of 2 if the text is wider than the text field's bounds.

6. Setting the `clearsOnBeginEditing` property to YES will clear any existing text when you first touch the control:

```
myTextField.clearsOnBeginEditing =  
→ YES;
```

```
myTextField.clearButtonMode =  
→ UITextFieldViewModeWhileEditing;
```

The `clearButtonMode` property adds a small button to the end of the text field, letting you clear the text at any time B. You can determine when this button is shown, such as only when editing the text.

To use keyboards:

1. Tap in the text field to open a keyboard from the bottom of the screen.
2. You can choose from a number of keyboard styles C, each designed for particular situations such as entering numbers or using a web browser.
3. Set the style with the `keyboardType` property.

By default, the keyboard will automatically suggest words as you type.

continues on next page

4. To disable this function, set the **autocorrectionType** property to **UITextAutocorrectTypeNo**.
5. Set the **autocapitalizationType** property to determine whether the keyboard capitalizes your typing by word, sentence, or even all characters.
6. You can change the text on the Return key via the **returnKeyType** property.
7. Use the **enablesReturnKeyAutomatically** property to determine whether the Return key is enabled even if you haven't entered any text into the text field.

In **Code Listing 4.23**, the **secureTextEntry** property is set to **YES**, which is useful for text fields that contain passwords or other sensitive information. As you enter text, you will see only the last letter typed.

8. You may have noticed that the keyboard doesn't disappear when you press the Return key. To hide the text field, you must implement the **textFieldShouldReturn:** delegate and tell the text field to resign its first responder status:

```
[textField resignFirstResponder];
return YES;
```

Code Listing 4.23 Creating a secure text field.

```
- (void)viewDidLoad {
    CGRect textRect = CGRectMake(10,10,300,31);
    UITextField *myField = [[UITextField alloc]
                           initWithFrame:textRect];
    myField.borderStyle = UITextBorderStyleRoundedRect;

    myField.font = [UIFont systemFontOfSize:22.0];
    myField.adjustsFontSizeToFitWidth = YES;
    myField.minimumFontSize = 2.0;

    myField.clearButtonMode = UITextFieldViewModeWhileEditing;
    myField.keyboardType = UIKeyboardTypeDefault;
    myField.autocorrectionType = UITextAutocorrectTypeNo;
    myField.autocapitalizationType = UITextAutocapitalizationTypeNone;
    myField.returnKeyType = UIReturnKeyDone;

    myField.secureTextEntry = YES;

    [self.view addSubview:myField];
    [myField release];
}
```

- 9.** Similarly, you can make the keyboard appear automatically when the view is loaded by setting the first responder status in the `viewDidLoad:` method:

```
[myTextField  
→ becomeFirstResponder];
```

- 10.** To prevent the keyboard from showing at all, which is useful if you are implementing your own custom keyboard, return `NO` from the `textFieldShouldBeginEditing:` delegate method.

TIP For a complete list of keyboard options, refer to the “`UITextInputTraits Protocol`” section of the developer documentation.

Restricting content

You can also use the delegate methods to control the text being entered into the text field. The `textField:shouldChangeCharactersInRange:replacementString:` delegate is called whenever the text is changed. You could, for example, use this method to restrict the number of characters entered. **Code Listing 4.24** shows a text field that allows a maximum of ten characters.

Code Listing 4.24 Limiting the contents of a text field to ten characters.

```
- (BOOL)textField:(UITextField *)textField  
shouldChangeCharactersInRange:(NSRange)range  
replacementString:(NSString *)string  
{  
    //limit text field to 10 chars  
    int MAX_CHARS = 10;  
    NSMutableString *newText = [NSMutableString  
                                stringWithString:textField.text];  
    [newText replaceCharactersInRange:range  
                           withString:string];  
    return ([newText length] <= MAX_CHARS);  
}
```

You should check the length of the replacement rather than just looking at the length of the text in the text field, since the text field's contents can be altered via copy and paste as well as by using the keyboard.

For the same reason, simply changing the keyboard type to numeric does not guarantee that a user will enter only numeric values (since a user could paste non-numeric values into the field). **Code Listing 4.25** shows the same delegate method, this time restricting the text field to allow numeric values only.

Text views

The **UITextView** class allows for multiline editable text. Although similar to text fields, text views feature a number of important differences.

Text views don't have any support for automatically reducing the font size like text fields have. Also, they don't have any support for clearing the text other than through programmatically setting the **text** property. There is also no support for secure text entry.

As with text fields, text views also apply the same text style to the entire text. Apple recommends using a **UIWebView** (see the “Web Views” section) if you require multiple styles in your text.

Code Listing 4.25 Restricting the contents of a text field to numeric values.

```
- (BOOL)textField:(UITextField *)textField  
shouldChangeCharactersInRange:(NSRange)range  
replacementString:(NSString *)string  
{  
    //limit text field to numeric values only  
    NSCharacterSet *numberSet = [NSCharacterSet  
        decimalDigitCharacterSet];  
    for (NSUInteger i=0; i<[string length]; i++)  
    {  
        unichar ch = [string characterAtIndex:i];  
        if (![numberSet characterIsMember:ch])  
            return NO;  
    }  
    return YES;  
}
```



D A text view with an active data detector.

Data detectors

Text views can analyze their contents and convert any links or phone numbers into tappable links by using a capability known as *data detectors*. Tapping the link will either launch the browser or call the phone number.

Two data detector types are available: **UIDataDetectorTypePhoneNumber** for phone numbers and **UIDataDetectorTypeLink** for Web **http:** links. To enable both, set the **dataDetectorTypes** property:

```
myTextView.dataDetectorTypes =  
→ UIDataDetectorTypeAll;
```

There's one caveat with data detectors: The default behavior of text views is to show the keyboard when tapped, so you can't tap the link of a data detector. For data detectors to work, you must set the **editable** property of the text view to **NO**. In D, the URL is underlined just as it would be in a web browser. Tapping it will launch the Safari application.

Hiding the keyboard

A text view's keyboard behaves the same as a text field, with one important difference: Since a text view supports multiline editing, pressing the Return button on the keyboard will insert a carriage return instead of calling a delegate method. Just as with the text field, resign the text view's first responder status to hide the keyboard when you have finished editing the text. This is often done as an action within another control.

Scrolling the interface

You may have noticed that since the iPhone's keyboards are very large, they take up a lot of the screen and can overlap other controls when shown. It would be handy if your interface moved up when the keyboard appeared and then moved back down once it disappeared.

You can make that happen by placing the controls inside a **UIScrollView**. When the keyboard appears, you simply scroll everything up, scrolling back down when the keyboard hides.

To scroll the interface in response to the keyboard:

1. Create and add a scroll view, making it the full size of the main view:

```
CGRect viewRect =  
    [self.view bounds];  
  
myScrollView = [[UIScrollView  
    alloc] initWithFrame:viewRect];  
  
myScrollView.contentSize =  
    viewRect.size;  
  
[self.view  
    addSubview:myScrollView];
```

2. Add the controls to the scroll view instead of the main view (since you want them to scroll).
3. Implement the **textViewDidBeginEditing** delegate method, which is called when the keyboard is shown.

Here you need to calculate both the bottom of the text view and the top of the keyboard and then tell the scroll view to scroll the difference. You must also look at the **orientation** property of the iPhone because the keyboard will have a different height in portrait mode than in landscape mode.

- 4.** Implement the `textViewDidEndEditing:` delegate so that when the keyboard is hidden, you scroll the text view to its original position. **Code Listing 4.26** shows the completed code.

Code Listing 4.26 Scrolling an interface in response to a keyboard.

```
UIScrollView *myScrollView;
UITextView *myTextView;

@implementation UITestViewController

-(void)buttonClick:(id)sender
{
    [myTextView resignFirstResponder];
}

-(void)viewDidLoad {
    CGRect viewRect = [self.view bounds];
    myScrollView = [[UIScrollView alloc] initWithFrame:viewRect];
    myScrollView.contentSize = viewRect.size;
    [self.view addSubview:myScrollView];

    CGRect buttonFrame = CGRectMake(10,10,60,32);
    UIButton *keyboardToggle = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [keyboardToggle setTitle:@"hide" forState:UIControlStateNormal];
    [keyboardToggle addTarget:self action:@selector(buttonClick:) forControlEvents:UIControlEventTouchUpInside];
    keyboardToggle.frame = buttonFrame;
    [myScrollView addSubview:keyboardToggle];

    CGRect textRect = CGRectMake(10,60,300,200);
    myTextView = [[UITextView alloc] initWithFrame:textRect];

    myTextView.font = [UIFont systemFontOfSize:22.0];
    myTextView.keyboardType = UIKeyboardTypeDefault;
    myTextView.returnKeyType = UIReturnKeyGo;
    myTextView.delegate = self;

    [myScrollView addSubview:myTextView];

    [myTextView release];
    [myScrollView release];
}

-(void)textViewDidBeginEditing:(UITextView *)textView {
    float keyboardHeight;
    if ([UIDevice currentDevice].orientation == UIDeviceOrientationPortrait | UIDeviceOrientationPortraitUpsideDown)
        keyboardHeight = 216.0;
    else
        keyboardHeight = 162.0;

    CGRect textViewRect = textView.frame;
    float textViewBottom = textViewRect.origin.y + textViewRect.size.height;
    CGRect viewRect = [myScrollView bounds];
    float keyboardTop = viewRect.size.height-keyboardHeight;
    float scrollOffset = fabs(textViewBottom - keyboardTop);
    [myScrollView setContentOffset:CGPointMake(0, scrollOffset) animated:YES];
}

-(void)textViewDidEndEditing:(UITextView *)textView {
    [myScrollView setContentOffset:CGPointMake(0, 0) animated:YES];
}
```

Web Views

Just as with iPhone's native Safari application, you can display web-based content in your own applications by using the **UIWebView** class. (In fact, Safari on the iPhone uses a **UIWebView** for display.)

Web views provide touch-based control for zooming in and out of pages, panning, and scrolling. Tapping links can load pages, and tapping in text controls will open a keyboard for data entry.

To display a web page in your application:

1. Just as with other views, you can add web views to your interface in the usual way:

```
CGRect webRect = CGRectMake  
→ (10,10,300,400);  
  
UIWebView *myWebView =  
→ [[UIWebView alloc]  
→ initWithFrame:webRect];  
  
myWebView.scalesPageToFit = YES;
```

The **scalesPageToFit** property ensures that larger pages are zoomed out or in enough to fit correctly in the current frame as well as letting you zoom in and out in response to pinch gestures.



A A web view displaying the Google homepage.

2. Use the **loadRequest**: method to load content into the web view, which takes an **NSURLRequest** object as its only parameter:

```
NSURL *url = [NSURL URLWithString:  
→ @"http://www.google.com"];  
  
NSURLRequest *request =  
→ [NSURLRequest  
requestWithURL:url];  
  
[myWebView loadRequest:request];  
[self.view addSubview:myWebView];
```

This would load the Google homepage A.

3. If your page is taking a long time or you want to cancel loading, use the **stopLoading** method. You can also check the **loading** property to make sure the page is actually in the process of loading.

What's the status?

Web views provide four optional delegate methods that will notify you about changes in the status of loading a web page:

webView:shouldStartLoadWithRequest:navigationType:: is sent before the web view begins to load the content and is a handy place to handle navigation within your web views (see the “Loading local content and handling hyperlinks” section).

webViewDidStartLoad: is sent when your web page starts loading and is a good place to show a progress indicator.

webViewDidFinishLoad: is sent when the web view finishes loading a page and is a good place for you to stop a progress indicator. This will not be sent if the page fails to load for any reason.

webView:didFailLoadWithError: is sent if an error occurs in loading the web page.

If you are building a web browser–type interface with Forward and Backward buttons, you can use the **canGoForward** and **canGoBackward** properties to determine whether your buttons should be enabled, and you can use the **goForward** and **goBackward** methods to navigate through the web view’s page history.

Although there is no direct access to the page history, you can easily maintain history via the delegate methods mentioned in the “What’s the status?” sidebar. **Code Listing 4.27** shows the code updated to include an activity indicator when the page is loading.

Code Listing 4.27 Implementing a web view.

```
UIActivityIndicatorView *activity;  
  
@implementation UITestViewController  
  
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    [self.view setBackgroundColor:[UIColor blackColor]];  
  
    CGRect webRect = CGRectMake(10,10,300,300);  
    UIWebView *myWebView = [[UIWebView alloc] initWithFrame:webRect];  
    myWebView.scalesPageToFit = YES;  
  
    myWebView.delegate = self;  
  
    NSURL *url = [NSURL URLWithString:@"http://www.google.com"];  
    NSURLRequest *request = [NSURLRequest requestWithURL:url];  
    [myWebView loadRequest:request];  
  
    [self.view addSubview:myWebView];  
  
    activity = [[UIActivityIndicatorView alloc] initWithActivityIndicatorStyle:UIActivityIndicatorViewStyleWhiteLarge];  
    [activity setCenter:CGPointMake(160,420)];  
    [self.view addSubview:activity];  
  
    [myWebView release];  
}  
  
- (void)webViewDidStartLoad:(UIWebView *)webView  
{  
    [activity startAnimating];  
}  
  
- (void)webViewDidFinishLoad:(UIWebView *)webView  
{  
    [activity stopAnimating];  
    [webView stringByEvaluatingJavaScriptFromString:@"alert('Finished Loading!');"];  
}  
  
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error  
{  
    [activity stopAnimating];  
    NSLog(@"Error: %@",error);  
}
```

Running JavaScript

You can execute JavaScript in a web view by using the **stringByEvaluatingJavaScriptFromString:** method. For example, if you wanted to open an alert dialog box in your web view, write the following:

```
[webView stringByEvaluating  
→ JavaScriptFromString:@"alert  
→ ('Hello World!');"];
```

This lets you manipulate your web page's style sheet or DOM or even call existing JavaScript functions defined within the page itself. For example, to change the background color of your web page, you could write the following:

```
[webView stringByEvaluating  
→ JavaScriptFromString:  
→ @"document.bgColor= '#000000\";"];
```

To call the JavaScript function **myFunction** defined in the web page, you could use this:

```
[webView stringByEvaluating  
→ JavaScriptFromString:  
→ @"myFunction();"];
```

For performance reasons, any JavaScript you call must execute fully within ten seconds and must be less than 10MB in size.

Loading local content and handling hyperlinks

You can also use web views to display *local* content, such as an .html file that ships in your application bundle. This makes web views handy for displaying content that mixes graphics with text or requires multiple text styles. (Remember that **UILabels** and **UITextView**s are restricted to only a single style per control.)

To load content from a local file:

1. Use the **loadHTMLString:baseURL:** method to load content contained in the resources folder of the application bundle:

```
myWebView.scalesPageToFit = NO;  
  
NSString *htmlPath =  
    → [[NSBundle mainBundle]  
    → pathForResource:@"myPage"  
    → ofType:@"html"];  
  
NSString *htmlContent =  
    → [NSString stringWithContents  
    → OfFile:htmlPath encoding:  
    → NSUTF8StringEncoding error:nil];  
  
[myWebView loadHTMLString:  
    → htmlContent baseURL:nil];
```

This time, **scalePageToFit** has been set to **NO** because you don't want the user to be able to zoom in and out of the web view as if it were a web page **B**.

2. To prevent a user from tapping any hyperlinks in the document, you can set **userInteractionEnabled** to **NO** for the web view. This also disables the ability to scroll content that may be longer than the control can fit on the screen at once.

or



B A web view displaying some local content, including a hyperlink to another page.

To disable links entirely, return **NO** from the `webView:shouldStartLoadWithRequest:navigationType:` delegate method.

3. To open a link in the native Safari application, you could write the following in the `webView:shouldStartLoadWithRequest:navigationType:` delegate method (Code Listing 4.28):

```
NSURL *pageURL = [request URL];
if ( ([[pageURL scheme]
      isEqualToString: @"http"]) &&
    (navigationType == UIWebView
     NavigationTypeLinkClicked ))
[[UIApplication sharedApplication]
  openURL:pageURL];
return NO;
```

Here you are trapping only **http:** links. Other link types, such as **https:**, would have no effect.

TIP When implementing this type of functionality, it's common to warn the user they are navigating away from your application and have them confirm the action. You can do this with the `UIAlertView` described in the exercise, "To display an alert view," earlier in this chapter.

Code Listing 4.28 Capturing clicks on a web view.

```
- (void)viewDidLoad {
[super viewDidLoad];

CGRect webRect = CGRectMake(10,10,300,400);
UIWebView *myWebView = [[UIWebView alloc] initWithFrame:webRect];
myWebView.delegate = self;

myWebView.scalesPageToFit = NO;
NSString *htmlPath = [[NSBundle mainBundle] pathForResource:@"myPage" ofType:@"html"];
NSString *htmlContent = [NSString stringWithContentsOfFile:htmlPath encoding:NSUTF8StringEncoding error:nil];
[myWebView loadHTMLString:htmlContent baseURL:nil];
[self.view addSubview:myWebView];
[myWebView release];
}

-(BOOL)webView:(UIWebView *)webView
shouldStartLoadWithRequest:(NSURLRequest *)request
navigationType:(UIWebViewNavigationType)navigationType {
NSURL *pageURL = [request URL];

if ( ([[pageURL scheme] isEqualToString: @"http"]) && (navigationType == UIWebViewNavigationTypeLinkClicked ))
{
[[UIApplication sharedApplication] openURL:pageURL];
return NO;
}
return YES;
}
```

Controls

Almost all the drawing functionality you've learned about so far also applies to controls. Most controls inherit their class from **UIControl**, and **UIControl** is a subclass of **UIView** **A**; this is how controls know how to draw themselves.

You'll never actually create instances of **UIControl** directly the way you do with **UIView**. **UIControl** is simply used to define a common set of functionality and behavior for its subclasses.

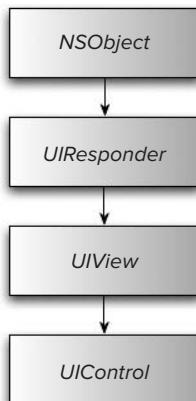
As mentioned at the beginning of this chapter, controls use the target-action mechanism to respond to touch events. Since the iPhone is a Multi-Touch device, many different events can occur, such as tapping, multitapping, dragging, and releasing. Luckily, each control has been designed to respond to only those events appropriate for its usage, and each does so in an intuitive and consistent manner.

You'll now take a closer look at the controls available to iPhone developers.

Buttons

When adding buttons to your application, you'll use the **UIButton** class **B**. The default initializer for buttons is the **buttonWithType:** method:

```
UIButton *myButton =  
→ [[UIButton buttonWithType:  
→ UIButtonTypeRoundedRect];
```



A The **UIControl** class hierarchy.



B The default button types for **UIButton**.

To be notified when a button changes state, add a target and action:

```
[myButton addTarget:self action:  
→ @selector(buttonClick:)  
→ forControlEvents:UIControlEventTouchUpInside  
→ TouchUpInside];
```

The `UIControlEventTouchUpInside` event is most commonly used for handling regular button presses.

The `UIButtonTypeCustom` type lets you create buttons with images or even draw them yourself using your own custom drawing code (as discussed earlier in the “Views” section).

To create a button with an image:

1. Specify an image for the button’s default state using `NSControlStateNormal`:

```
UIImage *buttonImage =  
→ [UIImage imageNamed:  
→ @"myButtonImage.png"];  
  
[myButton setImage:buttonImage  
→ forState:UIControlStateNormal];
```

`UIButton` will automatically apply highlight effects to indicate that the button is pressed or disabled.

2. You can also set multiple appearance properties for each of these states, including the title text, font, and color.

You can use different images for the four different states: the default (as shown in step 1), highlighted, selected, and disabled. This enables you to create buttons to represent other controls.

To create a checkbox button:

1. Assign images for both of the buttons' states:

```
[checkbox setImage:[UIImage  
→ imageNamed:@"checkbox_off.png"]  
→ forState:UIControlStateNormal];  
  
[checkbox setImage:[UIImage  
→ imageNamed:@"checkbox_on.png"]  
→ forState:UIControlStateNormalSelected  
];
```

2. Set the target method to call when the button is tapped:

```
[checkbox addTarget:self action:  
→ @selector(checkboxClick:)  
→ forControlEvents:UIControlEventTouchUpInside];
```

3. In the `checkboxClick:` method, simply flip the button's `selected` property:

```
btn.selected = !btn.selected;
```

Since you've previously defined images for the two different states, the button automatically updates to display the correct image. **Code Listing 4.29** shows the updated code.

TIP If you specify an image or title for any button type other than `UIButtonTypeRoundedRect`, the button effectively becomes a button of `UIButtonTypeCustom`.

Switches

Switches, represented by the `UISwitch` class, let you create an on/off control .

To create a switch:

1. Use the `initWithFrame:` method:

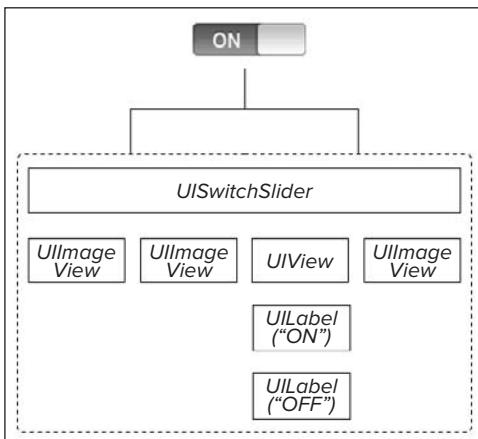
```
CGRect switchRect = CGRectMake  
→ (120,50,0,0);  
  
UISwitch *mySwitch = [[UISwitch  
→ alloc] initWithFrame:switchRect];
```

Code Listing 4.29 Creating a check box.

```
- (void)checkboxClick:(UIButton *)sender {  
    sender.selected = ! sender.selected;  
}  
  
- (void)viewDidLoad {  
    UIButton *checkbox = [UIButton buttonWithType:  
                           UIButtonTypeCustom];  
  
    CGRect checkboxRect = CGRectMake(135,150,36,36);  
    [checkbox setFrame:checkboxRect];  
  
    [checkbox setImage:[UIImage  
                     imageNamed:@"checkbox_off.png"]  
     forState:UIControlStateNormal];  
    [checkbox setImage:[UIImage  
                     imageNamed:@"checkbox_on.png"]  
     forState:UIControlStateNormalSelected];  
  
    [checkbox addTarget:self  
                  action:@selector(checkboxClick:)  
                  forControlEvents:UIControlEventTouchUpInside];  
  
    [self.view addSubview:checkbox];  
}
```



C Switches are used extensively in the Settings application of the iPhone.



D The control hierarchy that makes up a UISwitch.

Since switches are always the same size, the width and height properties are ignored.

- When you change a switch's value, it generates a `UIControlEventValueChanged` event:

```
[mySwitch addTarget:self action:  
→ @selector(switchAction:  
→ forControlEvents:UIControlEventValueChanged  
→ ValueChanged];
```

- To turn a switch on/off, call the `setOn:animated:` method:

```
[mySwitch setOn:YES animated:YES];
```

Switches don't have any properties for modifying the default visual appearance, but with a little digging, you can control a couple of elements.

Within the control hierarchy of a `UISwitch`, the "on" and "off" elements are `UILabels`

- You can manipulate the text, font, color, and more.

To alter the appearance of a switch:

- To retrieve the two `UILabels` within the switch that hold the switch's text , you can use this:

```
UIView *mainView = [[[mySwitch  
→ subviews] objectAtIndex:0]  
→ subviews] objectAtIndex:2];  
  
UILabel *onLabel = [[mainView  
→ subviews] objectAtIndex:0];  
  
UILabel *offLabel = [[mainView  
→ subviews] objectAtIndex:1];
```

continues on next page

2. Now you can change the text and color of these labels. The choice of text values is quite limited since the labels are small in size and are clipped by their containing view:

```
onLabel.text = @"YES";
offLabel.text = @"NO";
onLabel.textColor =
→ [UIColor yellowColor];
offLabel.textColor =
→ [UIColor greenColor];
```

3. When setting the text values, you should localize your replacement text wherever possible. **Code Listing 4.30** shows the updated code.

Code Listing 4.30 Customizing the switch control.

```
- (void)switchAction:(id)sender
{
    NSLog(@"switch changed");
}

- (void)viewDidLoad {
    [super viewDidLoad];

    CGRect switchRect = CGRectMake(120,50,0,0);
    UISwitch *mySwitch = [[UISwitch alloc] initWithFrame:switchRect];
    [mySwitch addTarget:self action:@selector(switchAction:) forControlEvents:UIControlEventValueChanged];

    //customize the appearance
    UIView *mainView = [[[mySwitch subviews] objectAtIndex:0] subviews] objectAtIndex:2];
    UILabel *onLabel = [[mainView subviews] objectAtIndex:0];
    UILabel *offLabel = [[mainView subviews] objectAtIndex:1];

    //change the text
    onLabel.text      = @"YES";
    offLabel.text     = @"NO";

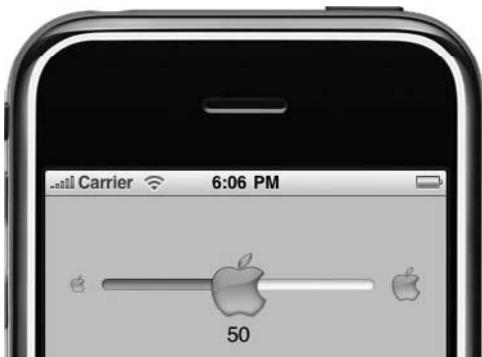
    //change the text color
    onLabel.textColor = [UIColor yellowColor];
    offLabel.textColor = [UIColor greenColor];

    [self.view addSubview:mySwitch];
}

[mySwitch release];
}
```



E The brightness slider control indicates the change in value with graphics at both ends of the control.



F A custom slider control.

Sliders

Although switches have only two possible states, sliders let you select from a range of values on a horizontal bar, or *track*, with a *thumb* indicator that can be moved from side to side to select values **E**.

Unlike the **UISwitch**, there's quite a lot you can do to customize the visual appearance of sliders, such as putting images to represent the values at either end of the track. You can also customize the thumb image and the graphics that appear on the track on both sides of the thumb as the values change **F**.

Just as with the **UISwitch**, sliders create a **UIControlEventValueChanged** event when their value is changed. By setting the **continuous** property, you can choose to have these events fired either as the slider is changed or at the end of a change. **Code Listing 4.31** demonstrates this with a custom **UISlider**, with minimum, maximum, and thumb images. In the **sliderAction:** method, you are forcing a “step” behavior, making the slider jump to the next value in increments of ten. A label added to the view displays the current slider value.

TIP Although not specified in the developer documentation, setting the thumb image of a **UISlider** also hides the tracking image. You must also set the minimum and maximum track images.

TIP The **stretchableImageWithLeftCapWidth:topCapHeight:** method lets you create an image that can stretch in the center but does not stretch on either side, as shown in the rounded edges of the track images.

Code Listing 4.31 Implementing a custom slider.

```
UILabel *lblSliderValue;

@implementation UITestViewController

-(void)sliderAction:(id)sender
{
    int stepAmount = 10;
    float stepValue = (abs([UISlider *)sender value]) / stepAmount) * stepAmount;
    [sender setValue:stepValue];

    lblSliderValue.text = [NSString stringWithFormat:@"%@",(int)stepValue];
}

-(void)viewDidLoad {
    [super viewDidLoad];

    CGRect sliderRect = CGRectMake(20,50,200,40);
    UISlider *mySlider = [[UISlider alloc] initWithFrame:sliderRect];

    mySlider.minimumValue = 0;
    mySlider.maximumValue = 100;
    mySlider.continuous = YES;

    //images
    UIImage *leftTrackImage = [[UIImage imageNamed:@"left_slider.png"] stretchableImageWithLeftCapWidth:5.0 topCapHeight:0.0];
    UIImage *rightTrackImage = [[UIImage imageNamed:@"right_slider.png"] stretchableImageWithLeftCapWidth:5.0 topCapHeight:0.0];
    UIImage *thumbImage = [UIImage imageNamed:@"apple_thumb.png"];
    UIImage *minSliderImage = [UIImage imageNamed:@"apple_min.png"];
    UIImage *maxSliderImage = [UIImage imageNamed:@"apple_max.png"];

    [mySlider setThumbImage:thumbImage forState:UIControlStateNormal];
    [mySlider setMinimumTrackImage:leftTrackImage forState:UIControlStateNormal];
    [mySlider setMaximumTrackImage:rightTrackImage forState:UIControlStateNormal];
    [mySlider setMinimumValueImage:minSliderImage];
    [mySlider setMaximumValueImage:maxSliderImage];
    [mySlider setValue:50.0f];

    //handle value change events
    [mySlider addTarget:self action:@selector(sliderAction:) forControlEvents:UIControlEventValueChanged];

    //label to show current value
    CGRect lblRect = CGRectMake(145,100,100,20);
    lblSliderValue = [[UILabel alloc] initWithFrame:lblRect];
    lblSliderValue.backgroundColor = [UIColor clearColor];

    lblSliderValue.text = [NSString stringWithFormat:@"%@",(int)mySlider.value];

    //add slider to main view
    [self.view addSubview:mySlider];
    [self.view addSubview:lblSliderValue];

    [lblSliderValue release];
    [mySlider release];
}
```



6 A segment control.

Segmented controls

The **UISegmentedControl** consists of a horizontal control divided into segments. 6. Segmented controls are useful for allowing users to pick from a group or set of values.

Each segment functions as its own button. By default, selecting a segment will deselect the others in the control (much as a radio button does in HTML). You can alter this behavior by setting the **momentary** property.

To create a segmented control:

1. Create an array of **UIImages** or **NSMutableString**s, and then call the default initializer **initWithItems:** (Code Listing 4.32).
2. Set the frame, and the control will automatically resize to accommodate its segments.

Each segment will initially be the same size.

continues on next page

Code Listing 4.32 Creating a segment control.

```
- (void)viewDidLoad {  
    NSArray *arrSegments = [[NSArray alloc] initWithObjects:  
        [NSString stringWithString:@"0"],  
        [NSString stringWithString:@"1"],  
        [NSString stringWithString:@"2"],  
        nil];  
  
    UISegmentedControl *mySegment = [[UISegmentedControl alloc]  
        initWithItems:arrSegments];  
  
    CGRect segmentRect = CGRectMake(10,50,300,40);  
    [mySegment setFrame:segmentRect];  
  
    [self.view addSubview:mySegment];  
    [arrSegments release];  
    [mySegment release];  
}
```

3. Set the width of individual segments using the `setWidth:forSegmentIndex:` method.

This will automatically resize any other segments that have not had their widths explicitly set to fit within the control.

4. Select segments using the `setSelectedSegmentIndex:` method.
5. Disable individual segments using the `setEnabled:forSegmentAtIndex:` method.
6. Add more segments using
`insertSegments WithImage:atIndex:
animated:`
or
`insertSegmentsWithTitle:atIndex:
animated:`
7. Set the `animated` property to YES so your segments will “slide in” as they are added.
8. To remove segments, use the `removeSegmentsAtIndex:animated:` method.
9. Use `removeAllSegments` to clear the entire control.

Segment control styles

Segment controls have three different styles, which can be set using the `segmentedControlStyle` property (H). Set the style to `UISegmentedControlStyleBar` to change the color of the control via the `tintColor` property, but depending on the color you use, you may not be able to see the difference between selected and unselected. **Code Listing 4.33** shows an example of how to use some of the properties of a segmented control.

Use the `UISegmentControl` to create a “glass” alternative to a `UIButton`. Use the `tintColor` property to change the color of the button.



(H) The three styles available for segmented controls.

Code Listing 4.33 Setting some of the segment control properties.

```
- (void)segmentClick:(id)sender {
    NSLog(@"%@",[sender selectedSegmentIndex]);
}

- (void)viewDidLoad {
    NSArray *arrSegments = [[NSArray alloc] initWithObjects:
        [NSString stringWithString:@"0"],
        [NSString stringWithString:@"1"],
        [NSString stringWithString:@"2"],
        nil];
    UISegmentedControl *mySegment = [[UISegmentedControl alloc]
        initWithFrame:segmentRect];
    [mySegment setItems:arrSegments];
    mySegment addTarget:self
        action:@selector(segmentClick:)
        forControlEvents:UIControlEventValueChanged];
    [mySegment setSegmentedControlStyle:UISegmentedControlStyleBar];
    [mySegment setTintColor:[UIColor darkGrayColor]];

    //select first item
    [mySegment setSelectedSegmentIndex:0];

    //change a segment size
    [mySegment setWidth:120.0 forSegmentAtIndex:1];

    //add a new segment
    [mySegment insertSegmentWithTitle:@"new" atIndex:2 animated:YES];
    [self.view addSubview:mySegment];
    [arrSegments release];
    [mySegment release];
}
```

This page intentionally left blank

5

Tabs and Tables

At the core of most iPhone applications, the view controller classes allow you to manage the visual elements that make up your application's interface in a consistent fashion while eliminating redundant code. View controllers are the C in the MVC design pattern, providing the layer between your data model and your user interface.

In this chapter, you'll see how to work with these classes. You'll start by looking at the base **UIViewController** class from which the other view controller classes inherit and see how it provides a lot of the core functionality required by any iPhone application. Then you'll see how you can add a tab-based interface using the **UITabBarController** class.

Finally, you'll investigate how to use the **UITableView** and **UINavigationController** classes to present tabular, hierarchical data.

In This Chapter

View Controllers	182
Tab Views	194
Table Views	200

View Controllers

The **UIViewController** class is the main class used to control most views (a view being a single screen within your application) and provides the base class for the **UITabBarController** and **UINavigationController** classes discussed later in this chapter.

Each view controller has a single *main* view—represented by the **view** property—which it alone owns. Main views cannot be shared between view controllers. The main view will generally have one or more *subviews* containing the actual content of your application (other views, controls, and user-interface elements). For more information on views, see Chapter 4, “iPhone User Interface Elements.”

These are a view controller’s main responsibilities:

- To manage the presentation of its views, responding to events such as changes in the iPhone’s orientation or low-memory situations
- To act as the coordinator between the user interface and the application’s data model (See the “Model View Controller” section in Chapter 1, “Objective-C and Cocoa,” for more information.)

Presenting views

As mentioned, each view controller contains a main view that you would normally use as the canvas for your application’s user interface. You can put your code in a number of places when working with view controllers:

- **loadView**—Although you should not call this method directly, if you create your own views manually, then you should override this method and assign them to the view controller’s **view** property.
- **viewDidLoad**—This method is called just after the view controller loads its views into memory and is a good place for you to perform any additional initialization. Many of the code examples in this book use this method as the point at which to create the user interface.
- **viewDidUnload**—Conversely, this is called when the view controller releases its views from memory and is a good place for you to clean up any objects you may have created within the view controller.
- **viewWillAppear:**—This is called when the view is about to be added to a window and is not yet visible. It’s a good place to perform any customization such as changing the view’s orientation. If you implement any code in this method, make sure you also call **[super viewWillAppear:]**.
- **viewDidAppear:**—This is called when the view has been added to a window and is a good place to place code that needs to run after the view has been presented. If you implement any code in this method, make sure you also call **[super viewDidAppear:]**.

continues on next page

- **viewWillDisappear:**—This is called when the view is about to be removed from a window and is a good place for you revert to any changes that may have been made when **viewWillAppear:** was called. If you implement any code in this method, make sure you also call **[super viewWillAppear:]**.
- **viewDidDisappear:**—This is called when the view is dismissed, covered, or hidden from view. If you implement any code in this method, make sure you also call **[super viewDidDisappear:]**.

Code Listing 5.1 shows an example of changing the background color and adding a button to the main view within the **viewDidLoad** method.

Responding to changes in orientation

Another responsibility of a view controller is to respond to changes in the iPhone's orientation and automatically rotate the interface where appropriate. The **shouldAutorotateToInterfaceOrientation:** method is called immediately after the iPhone is rotated. If you want to autorotate to all four orientations (up, down, left, and right), you can simply return **YES** from this method. Otherwise, you can examine the **interfaceOrientation** property and return **YES** or **NO** based on the device's current orientation.

Code Listing 5.1 Adding a button to the main view within the **viewDidLoad** method.

```
- (void)viewDidLoad {  
    CGRect labelFrame = CGRectMake(10,10,300,38);  
    orientationLabel = [[UILabel alloc] initWithFrame:labelFrame];  
    orientationLabel.textAlignment = NSTextAlignmentCenter;  
    orientationLabel.autoresizingMask = UIViewAutoresizingFlexibleWidth;  
    orientationLabel.text = [self getOrientationName];  
  
    [self.view addSubview:orientationLabel];  
}
```

To create an application that responds to changes in orientation:

1. Create a new view-based application, saving it as OrientationExample.
2. Open the OrientationExampleView Controller.h file, and create an instance variable to display the orientation ([Code Listing 5.2](#)).
3. Switch to the OrientationExampleView Controller.m file, uncomment the `viewDidLoad` method, and add the following:

```
CGRect labelFrame = CGRectMake
→ (10,10,300,38);

orientationLabel =
→ [[UILabel alloc]
→ initWithFrame:labelFrame];

orientationLabel.textAlignment =
→ NSTextAlignmentCenter;

orientationLabel.autoresizingMask =
→ UIViewAutoresizingFlexibleWidth;

orientationLabel.text =
→ [self getOrientationName];

[ self.view addSubview:
→ orientationLabel];
```

continues on next page

Code Listing 5.2 The header file of the orientation example.

```
#import <UIKit/UIKit.h>

@interface OrientationExampleViewController : UIViewController {
    UILabel *orientationLabel;
}
@end
```

You first create a label that spans the width of the screen. Notice how you set the text alignment of the label to be centered so that when the iPhone is rotated, your text will remain in the center of the label. Likewise, you set the **autoresizingMask** property so that the label stretches its width automatically when the iPhone is rotated from portrait to landscape mode. Finally, you call a method to set the text of the label.

4. Implement the **getOrientationName** method:

```
switch (self.interfaceOrientation)
{
    case UIInterfaceOrientation
        → Portrait:
        return @"Portrait";
        break;

    case UIInterfaceOrientation
        → PortraitUpsideDown:
        return @"Portrait (upside
        → down)";
        break;

    case UIInterfaceOrientation
        → LandscapeLeft:
        return @"Landscape (left)";
        break;

    case UIInterfaceOrientation
        → LandscapeRight:
        return @"Landscape (right)";
        break;
}
```

This method looks at the current orientation of the iPhone (via the view controller's **interfaceOrientation** property) and returns a descriptive string.

TIP Several methods are available for you to track changes in orientation:

willRotateToInterfaceOrientation:duration:—This is called just before rotation takes place. It's a good place for you to pause any activity that may be happening in the user interface and to disable the ability for the user to interact with the application until the interface has completed rotating.

willAnimateRotationToInterfaceOrientation:duration:—This is called with the rotation animation block itself and is a good place for you to write custom code to override the default rotation animation. For example, you could change the rotation to a cross-dissolve animation.

didRotateFromInterfaceOrientation:—This is called once the rotation has completed and is a good place for you to reenable your user interface and reallow user interactions with your application if you disabled them earlier.

5. Next, implement the **didRotateFromInterfaceOrientation:** method:

```
orientationLabel.text =  
→ [self getOrientationName];
```

This method is called just after the orientation of the iPhone changes. Here you simply update the label text.

6. Uncomment the **shouldAutorotateToInterfaceOrientation:** method, and add the following code:

```
return YES;
```

This will allow your interface to be autorotated in any direction.

7. Build and run the application.

If you rotate your iPhone, the label should automatically rotate and continue to fill the width of the screen. The text should also change to indicate the iPhone's current orientation. Try changing the allowed orientations by modifying the **shouldAutorotateToInterfaceOrientation:** method. **Code Listing 5.3** shows the completed code.

Code Listing 5.3 The completed orientation example.

```
#import "OrientationExampleViewController.h"

@implementation OrientationExampleViewController

- (NSString *)getOrientationName {

    switch (self.interfaceOrientation) {
        case UIInterfaceOrientationPortrait:
            return @"Portrait";
            break;

        case UIInterfaceOrientationPortraitUpsideDown:
            return @"Portrait (upside down)";
            break;

        case UIInterfaceOrientationLandscapeLeft:
            return @"Landscape (left)";
            break;

        case UIInterfaceOrientationLandscapeRight:
            return @"Landscape (right)";
            break;
    }

    return @"Unknown";
}

- (void)viewDidLoad {

    CGRect labelFrame = CGRectMake(10,10,300,38);
    orientationLabel = [[UILabel alloc] initWithFrame:labelFrame];
    orientationLabel.textAlignment = NSTextAlignmentCenter;
    orientationLabel.autoresizingMask = UIViewAutoresizingFlexibleWidth;
    orientationLabel.text = [self getOrientationName];

    [self.view addSubview:orientationLabel];
}

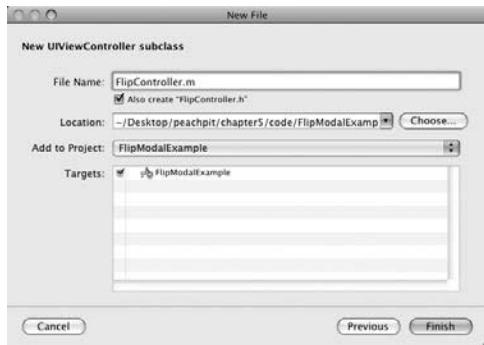
- (void)didRotateFromInterfaceOrientation:(UIInterfaceOrientation)fromInterfaceOrientation {
    orientationLabel.text = [self getOrientationName];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [orientationLabel release];
    [super dealloc];
}

@end
```



A Adding a second view controller class.

Code Listing 5.4 The header file of the flip modal example.

```
#import <UIKit/UIKit.h>
#import "FlipController.h"

@interface FlipModalExampleViewController: UIViewController
{
    FlipController *flip;
}
@end
```

Displaying modal views

You display a view modally by presenting one view controller from another view controller. Modal views are animated onto the screen, and you can determine an animation type to use. This can be helpful in providing meaning to the action you are performing. For example, it's common to provide settings for an application by “flipping” the settings view over the main view.

To display a view modally using a “flip” animation:

1. Create a new view-based application, saving it as `FlipModalExample`.
2. Select `File > New`, and add a new `UIViewController` subclass.
Save the file as `FlipViewController.m`. Make sure that “Also create ‘`FlipController.h`’” is selected **(A)**.
3. Open the `FlipModalExampleView Controller.h` file, import the header file, and create an instance variable to hold the flip view controller (**Code Listing 5.4**).

continues on next page

4. Switch to the FlipModalExampleView Controller.m file, uncomment the `viewDidLoad` method, and add the following:

```
flip = [[FlipController alloc]
→ init];
UIButton *viewButton = [UIButton
→ buttonWithType:UIButtonType
→ Rounded Rect];
CGRect buttonFrame = CGRectMake
→ (110,10,100,38);
[viewButton setFrame:buttonFrame];
[viewButton setTitle:@"Show"
→ forState:UIControlStateNormal];
[viewButton addTarget:self action:
→ @selector(buttonClick:)
→ forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:viewButton];
```

Here you create the flip controller and then add a button to your main controller view.

5. Implement the `buttonClick:` method:

```
flip.modalTransitionStyle =
→ UIModalTransitionStyleFlip
→ Horizontal;
[self presentModalViewController:
→ flip animated:YES];
```

Setting the `modalTransitionStyle` property of your flip controller to `UIModalTransitionStyleFlipHorizontal` uses a “flip” animation when the new view controller is shown modally.

6. Switch to `FlipController.m`, uncomment the `viewDidLoad` method, and add the following:

```
[self.view setBackgroundColor:  
→ [UIColor redColor]];  
  
UIButton *viewButton =  
→ [UIButton buttonWithType:  
→ UIButtonTypeRoundedRect];  
  
CGRect buttonFrame = CGRectMake  
→ (110,10,100,38);  
  
[viewButton setFrame:buttonFrame];  
[viewButton setTitle:@"Hide"  
→ forState:UIControlStateNormal];  
  
[viewButton addTarget:self action:  
→ @selector(buttonClick:)  
→ forControlEvents:UIControlEventTouchUpInside  
→ TouchUpInside];  
  
[self.view addSubview:viewButton];
```

This is almost the same code as with your main view controller, but you set the background of the view to make it obvious which view is showing.

7. Implement the second `buttonClick:` method:

```
[self dismissModalViewControllerAnimated:  
→ YES];
```

This tells the main view controller to hide the flip controller. The message is automatically forwarded to the main view controller (since it is the one dismissing the modal view). This is handy since it means you don't have to declare any public methods or create a reference to the parent view controller.

8. Build and run the application.

Pressing Show will cause the view to flip and show the other view controller's view modally. Pressing Hide will flip the view back. **Code Listing 5.5** shows the completed code.

TIP When presenting a view modally, you can pick from three animation styles:

UIModalTransitionStyleCover

Vertical is the default style and slides the view from the bottom to over the current view.

UIModalTransitionStyleFlip

Horizontal is the style used in the example here and flips the view from right to left.

UIModalTransitionStyleCross

Dissolve fades the current view out and the new view in using a cross-dissolve.

All three of these styles perform the reverse animation when the view is dismissed.

Code Listing 5.5 The completed flip modal application.

```
//  
//  FlipModalExampleViewController.m  
//  FlipModalExample  
//  
  
#import "FlipModalExampleViewController.h"  
  
@implementation FlipModalExampleViewController  
  
- (void)buttonClick:(id)sender {  
    flip.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;  
    [self presentModalViewController:flip animated:YES];  
}  
  
- (void)viewDidLoad {  
    flip = [[FlipController alloc] init];  
  
    UIButton *viewButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];  
    CGRect buttonFrame = CGRectMake(110,10,100,38);  
    [viewButton setFrame:buttonFrame];  
    [viewButton setTitle:@"Show" forState:UIControlStateNormal];  
    [viewButton addTarget:self  
        action:@selector(buttonClick:)  
        forControlEvents:UIControlEventTouchUpInside];  
    [self.view addSubview:viewButton];  
}  
  
- (void)didReceiveMemoryWarning {  
    [super didReceiveMemoryWarning];  
}  
  
- (void)dealloc {  
    [flip release];  
    [super dealloc];  
}  
  
@end  
  
//  
//  FlipController.m  
//  FlipModalExample  
//  
  
#import "FlipController.h"  
  
@implementation FlipController  
  
- (void)buttonClick:(id)sender {  
    [self dismissModalViewControllerAnimated:YES];  
}  
  
- (void)viewDidLoad {  
    [self.view setBackgroundColor:[UIColor redColor]];  
}
```

code continues on next page

Handling low-memory conditions

View controllers can automatically handle low-memory scenarios by releasing and cleaning up any views that aren't needed, such as when a view doesn't have a superview. When a memory warning is issued (via the **UIApplication** delegate), each view controller also receives a notification via the **didReceiveMemoryWarning** method. You can override this method to perform any memory cleanup within your own code. However, if you do so, make sure you also call **[super didReceiveMemoryWarning]**. You can also put your cleanup code in the **viewDidUnload** method, which will be called if the view controller decides it can release its views.

Since you would normally have multiple view controllers in your application, each with their own views and subviews, this provides a very granular memory management model where you can determine what you should release and when.

Code Listing 5.5 continued

```
UIButton *viewButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
CGRect buttonFrame = CGRectMake(110,10,100,38);
[viewButton setFrame:buttonFrame];
[viewButton setTitle:@"Hide" forState:UIControlStateNormal];
[viewButton addTarget:self
    action:@selector(buttonClick:)
    forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:viewButton];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [super dealloc];
}
@end
```

Tab Views

You can use the **UITabBarController** class to create a multipage interface. Pages are displayed by selecting a tab from a tab bar at the bottom of the screen **A**.

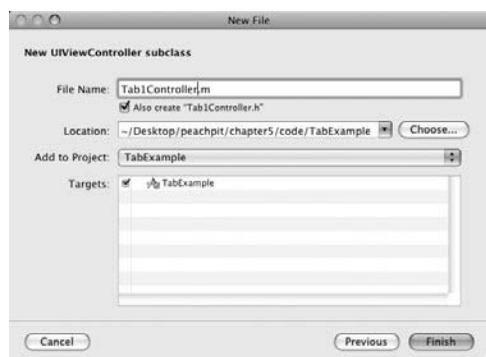
Each page of the tab bar controller is a view controller. When a tab is selected, the view of the corresponding view controller is displayed above the tab bar.

To create an application with tabs:

1. Create a new view-based application, saving it as TabExample.
2. Select File > New, and add a new **UIViewController** subclass.
Save the file as Tab1Controller.m. Make sure that “Also create ‘Tab1Controller.h’” is selected **B**.
3. Repeat step 2, adding two more **UIViewController** subclasses named **Tab2Controller** and **Tab3Controller**.
4. Open TabExampleViewController.h, import the three tab controllers, create instance variables to represent the three tabs, and change the class name to **UITabBarController** (**Code Listing 5.6**).



A The World Clock application uses tabs to switch between clock types.



B Adding a new **UIViewController** subclass.

Code Listing 5.6 The header file for the tab example.

```
#import <UIKit/UIKit.h>
#import "Tab1Controller.h"
#import "Tab2Controller.h"
#import "Tab3Controller.h"

@interface TabExampleViewController : UITabBarController {

    Tab1Controller *tab1;
    Tab2Controller *tab2;
    Tab3Controller *tab3;
}

@end
```

What happens if I have too many tabs?

In the exercises in this section, you added only a limited number of tabs to the tab bar. If you were to add so many that the tab bar ran out of room, a new special More tab would appear on the right edge of the tab bar C. Selecting this allows the user to customize the tab bar by choosing which tabs they want to display. You can restrict which tabs the user can edit by removing the corresponding view controller from the array in the `customizableViewControllers` property.



C The More tab.

5. Switch to TabExampleViewController.m, uncomment the `viewDidLoad` method, and add the following code:

```
tab1 = [[Tab1Controller alloc]
→ init];
tab2 = [[Tab2Controller alloc]
→ init];
tab3 = [[Tab3Controller alloc]
→ init];
NSArray *arrTabs = [NSArray array
→ WithObjects:tab1,tab2,tab3,nil];
[self setViewControllers:arrTabs
→ animated:YES];
```

Here you create three tabs, add them to an array, and then assign them to the `viewControllers` property, which will create a tab for each element.

6. Switch to Tab1Controller.m, uncomment the `viewDidLoad` method, and add the following code to set the background color:

```
[self.view setBackgroundColor:
→ [UIColor redColor]];
```

7. Repeat step 6 for Tab2Controller.m and Tab3Controller.m. Make sure you use a different color each time.

8. Build and run your application.

You should see a tab bar with three tabs. As you select a tab, the area above the tab bar will change color.

TIP You can select a tab in your code by using the `selectedIndex` property of the `UITabBarController`. This can be useful when you are developing your application, allowing you to launch your application on the page on which you are currently working.

Adding graphics and titles to tabs

Although this example showed you how easy it is to create an application with tabs, the tabs themselves are not particularly helpful. It would be much better if each one had a title or graphic on it so that you knew what it actually did. Luckily, **UIViewController** has a property called **tabBarItem** that is designed for exactly this purpose.

To update the application to use tab bar items:

1. Open Tab1Controller.m, and create an **init:** method:

```
UITabBarItem *tab = [[UITabBarItem  
→ alloc] initWithTitle:@"Tab 1"  
→ image:nil tag:0];  
  
[self setTabBarItem:tab];  
  
[tab release];
```

Here you create a new tab item and set its title only.

2. Open Tab2Controller.m, and again create an **init:** method:

```
UITabBarItem *tab = [[UITabBarItem  
→ alloc] initWithTabBarSystemItem:  
→ UITabBarSystemItemFavorites  
→ tag:1];  
  
[self setTabBarItem:tab];  
  
[tab release];
```

This time you are using the **initWithTabBarSystemItem:** method to create the tab item. The iPhone has a number of system tabs that can be used for common functionality and are used in many of the built-in applications. Using a system item adds both a graphic and a label to the tab. For more information on the available system items, refer to the *UITabBarItem Class Reference* in the developer documentation.



D The completed tab example application.

3. Open Tab3Controller.m, and create a third `init:` method:

```
UITabBarItem *tab = [[UITabBarItem  
alloc] initWithTitle:@"Tab 3"  
image:nil tag:2];  
  
[tab setBadgeValue:@"New!"];  
  
[self setTabBarItem:tab];  
  
[tab release];
```

Just as in step 1, you set a title, but no graphic, for the tab. However, this time, you also set the `badgeValue` property. This will add a red oval to the upper-right corner of your tab. This technique is commonly used to alert a user to a change in the contents of the tab.

4. Build and run the application D.

Code Listing 5.7 and **Code Listing 5.8** show the completed code.

TIP In this example, you created the tab bar item in the `init:` method instead of in the `viewDidLoad` method. You did this because `viewDidLoad` is not actually called until the view controller displays its view, which would mean that the tab would not draw its contents until it had been selected.

TIP You can use the delegate of a `UITabBarController` (which must implement the `UITabBarControllerDelegate` protocol) to disable tabs by implementing the `tabBarController:shouldSelectViewController:` method and returning `NO`.

Code Listing 5.7 The implementation files for the three tab view controllers.

```
@implementation Tab1Controller
- (id)init
{
    if (self = [super init])
    {
        UITabBarItem *tab = [[UITabBarItem alloc]
                               initWithTitle:@"Tab 1"
                               image:nil
                               tag:0];
        [self setTabBarItem:tab];
        [tab release];
    }
    return self;
}

- (void)viewDidLoad {
    [self.view
     setBackgroundColor:[UIColor redColor]];
}

- (void)dealloc {
    [super dealloc];
}
@end

@implementation Tab2Controller
- (id)init
{
    if (self = [super init])
    {
        UITabBarItem *tab = [[UITabBarItem alloc]
                               initWithTitle:UITabBarSystemItemFavorites
                               image:nil
                               tag:1];
        [self setTabBarItem:tab];
        [tab release];
    }
    return self;
}

- (void)viewDidLoad {
    [self.view
     setBackgroundColor:[UIColor blueColor]];
}

- (void)dealloc {
    [super dealloc];
}
@end

@implementation Tab3Controller
```

continues on next page

Code Listing 5.7 continued

```
- (id)init
{
    if (self = [super init])
    {
        UITabBarItem *tab = [[UITabBarItem alloc]
            initWithTitle:@"Tab 3"
            image:nil
            tag:2];
        [tab setBadgeValue:@"New!"];
        [self setTabBarItem:tab];
        [tab release];
    }
    return self;
}

- (void)viewDidLoad {
    [self.view
        setBackgroundColor:[UIColor greenColor]];
}

- (void)dealloc {
    [super dealloc];
}

@end
```

Code Listing 5.8 The tab bar controller implementation file.

```
#import "TabExampleViewController.h"

@implementation TabExampleViewController

- (void)viewDidLoad {
    tab1 = [[Tab1Controller alloc] init];
    tab2 = [[Tab2Controller alloc] init];
    tab3 = [[Tab3Controller alloc] init];

    NSArray *arrTabs = [NSArray arrayWithObjects:tab1,tab2,tab3,nil];
    [self setViewControllers:arrTabs animated:YES];
}

- (void)dealloc {
    [tab1 release];
    [tab2 release];
    [tab3 release];

    [super dealloc];
}

@end
```

Table Views

Table views—represented by the **UITableView** class—are the main interface element used to display lists and hierarchical data on the iPhone. Table views are used everywhere; the Settings, Clock, Notes, Mail, and Contact applications all use table views as the main interface element.

Tables views represent their data in *rows* and *sections*. A row is an individual item in the table view. Rows can be grouped into zero or more sections. Sections can have both a header and a footer **A**.

The content of a row is called a *cell* and is represented by the **UITableViewCell** class. You can use a cell's **textLabel** and **detailTextLabel** properties for displaying text, and you can use its **imageView** property for displaying images. Several different styles of cell exist, and you can also create your own custom cells.

To use a table view, you must implement a *data source* and, optionally, a *delegate*. The data source provides the table view with its contents and determines what (if anything) can be edited. The delegate manages the selection and editing of rows and the display of section information.

Several of the delegate and data source methods have an **NSIndexPath** as a parameter. Using the **section** and **row** properties of this object is how you determine exactly to what in your table view you are referring.



A The various elements of a table view.



⑧ Creating a navigation-based application.

Although you could create and add a table view much as you do with any other view, all the examples in this section use the **UITableViewcontroller** class, which is designed specifically to manage a table view. Table view controllers are already configured as both the data source and the delegate of their own table view, and they have some additional functionality for updating a navigation bar when editing a table view.

To create an application with a table view:

1. Create a new navigation-based application, saving it as TableViewExample ⑧.
2. Open RootViewController.h, and create the instance variable you will use to hold your table contents:

```
NSArray *arrCountries;
```

3. Switch to RootViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
self.title = @"Countries";  
  
arrCountries = [[NSArray alloc]  
→ initWithObjects: @"Australia",  
→ @"Canada",@"Germany",@"France",  
→ @"Great Britain",@"Italy",  
→ @"Japan",@"New Zealand",  
→ @"United States",nil];
```

Here you are setting the title of the toolbar and creating an array of country names.

4. Edit the **tableView:numberOfRowsInSection:** method to look like this:

```
return [arrCountries count];
```

This method is how the table view knows how many rows it should display. In this example, you return the size of your array.

continues on next page

5. Update the `tableView:cellForRowAtIndexPath:` method to look like the following:

```
static NSString *CellIdentifier =  
→ @"Cell";  
  
UITableViewCell *cell =  
→ [tableView dequeueReusableCellWithIdentifier:  
→ CellIdentifier:  
→ CellIdentifier];  
  
if (cell == nil) {  
  
    cell = [[[UITableViewCell  
→ alloc] initWithStyle:  
→ UITableViewCellStyleDefault  
→ reuseIdentifier:Cell  
→ Identifier] autorelease];  
  
}  
  
// Configure the cell.  
cell.textLabel.text =  
→ [arrCountries objectAtIndex:  
→ indexPath.row];  
  
NSString *imageName =  
→ [NSString stringWithFormat:  
→ @"%@.png", [arrCountries  
→ objectAtIndex:indexPath.row]];  
  
cell.imageView.image = [UIImage  
→ imageNamed:imageName];  
  
return cell;
```

Most of this code is unchanged from the template—you need to implement only the cell contents. Here you set the text by looking at the current row of the table and retrieving that value from your countries array. You also set an image for the cell; the image is a PNG file with the same name as the country. You will need to add a PNG graphic for each country's flag to your project. You can easily find these by visiting <http://images.google.com>.



⑥ The table view application.

TIP You may have noticed in `tableView:cellForRowAtIndexPath:` you gave your cell an identifier and used the `dequeue` `ReusableCellWithIdentifier:` method when you created the cell. For performance reasons, each unique type of cell is stored in a *reuse queue*. You can then check this queue to see whether a cell is available for reuse, rather than creating a new one.

6. Build and run the application.

You should see a list of countries with flags next to them **C**. **Code Listing 5.9** shows the completed code.

Code Listing 5.9 The code for the table view application.

```
#import "RootViewController.h"

@implementation RootViewController

- (void)viewDidLoad {
    self.title = @"Countries";
    arrCountries = [[NSArray alloc] initWithObjects:
        @"Australia",
        @"Canada",
        @"Germany",
        @"France",
        @"Great Britain",
        @"Italy",
        @"Japan",
        @"New Zealand",
        @"United States",nil];
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [arrCountries count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease;
    }

    // Configure the cell.
    cell.textLabel.text = [arrCountries objectAtIndex:indexPath.row];
    NSString *imageName = [NSString stringWithFormat:@"%@.png",[arrCountries objectAtIndex:indexPath.row]];
    cell.imageView.image = [UIImage imageNamed:imageName];
}

- (void)dealloc {
    [arrCountries release];
    [super dealloc];
}

@end
```

Grouping rows into sections and styles

As mentioned earlier, you can group your table view's rows into *sections*, which can be useful for displaying sets of related information. When using sections, it's also common to use the grouped style for your table view, which is the style used in the Settings application ①. This style represents cells as rounded rectangles with a shaded background.

To update the application to use sections and a grouped table view:

1. Open the RootViewController.h file, and modify the code to create three instance variables (**Code Listing 5.10**):

```
NSArray *arrAsiaPacific;  
NSArray *arrEurope;  
NSArray *arrNorthAmerica;
```

This time you are going to split the countries into three sections.

2. Switch to RootViewController.m, and update the `viewDidLoad` method:

```
arrAsiaPacific = [[NSArray alloc]  
→ in itWithObjects:@"Australia",  
→ @"Japan",@"New Zealand",nil];  
  
arrEurope = [[NSArray alloc]  
→ initWithObjects:@"Germany",  
→ @"France",@"Great Britain",  
→ @"Italy",nil];  
  
arrNorthAmerica = [[NSArray  
→ alloc] initWithObjects:@"Canada",  
→ @"United States",nil];
```

Here you are creating an array for each section of your table view.



① The Settings application uses the grouped table style.

Code Listing 5.10 The header file for the grouped table view application.

```
@interface RootViewController : UITableViewController  
{  
    NSArray *arrAsiaPacific;  
    NSArray *arrEurope;  
    NSArray *arrNorthAmerica;  
}  
  
@end
```

3. Update the `tableView:numberOfRowsInSection:` method to look like this:

```
switch(section)
{
    case 0:
        return [arrAsiaPacific count];
        break;
    case 1:
        return [arrEurope count];
        break;
    case 2:
        return [arrNorthAmerica count];
        break;
    default:
        return 0;
        break;
}
```

This checks which section is being requested and then returns the size of the corresponding array.

continues on next page

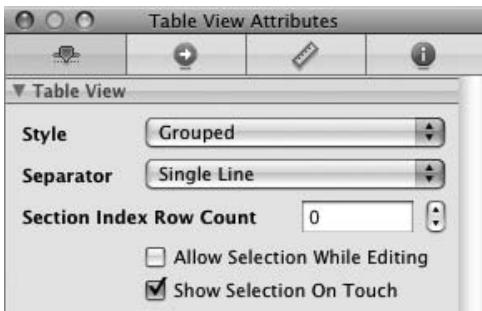
4. Similarly, you need to update the `tableView:cellForRowAtIndexPath:` method to check the correct array when you are creating the table view cell:

```
NSString *country;
switch(indexPath.section)
{
    case 0:
        country = [arrAsiaPacific
                    objectAtIndex:indexPath.row];
        break;
    case 1:
        country = [arrEurope
                    objectAtIndex:indexPath.row];
        break;
    case 2:
        country = [arrNorthAmerica
                    objectAtIndex:indexPath.row];
        break;
}
NSString *imageName = [NSString
    stringWithFormat:@"%@.png",
    country];
cell.imageView.image = [UIImage
    imageNamed:imageName];
cell.textLabel.text = country;

return cell;
```

5. Update the `numberOfSectionsInTableView:` method to return the correct number of sections:

```
return 3;
```



E Setting the table view style in Interface Builder.



F The grouped table view application.

6. Implement the `tableView:titleForHeaderInSection:` method to create the title text above each section:

```
switch(section)
{
    case 0:
        return @"Asia Pacific";
        break;
    case 1:
        return @"Europe";
        break;
    case 2:
        return @"North America";
        break;
    default:
        return nil;
        break;
}
```

7. Update the style of your table view using Interface Builder E.

Since the property is read-only, you can't edit it through code.

8. Build and run your application.

You should now see your data grouped into three sections F. **Code Listing 5.11** shows the updated code.

TIP Just as you added text above each table section, implementing the `tableView:titleForFooterInSection:` method lets you add text *below* each section. You can also add any `UIView` subclass at the top and bottom of your table view by setting the `tableHeaderView` and `tableFooterView` properties.

Code Listing 5.11 The code for the grouped table view application.

```
#import "RootViewController.h"

@implementation RootViewController

- (void)viewDidLoad {

    //create test data
    arrAsiaPacific = [[NSArray alloc] initWithObjects:@"Australia",@"Japan",@"New Zealand",nil];
    arrEurope = [[NSArray alloc] initWithObjects:@"Germany",@"France",@"Great Britain",@"Italy",nil];
    arrNorthAmerica = [[NSArray alloc] initWithObjects:@"Canada",@"United States",nil];
}

- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section {
    switch(section) {
        case 0:
            return @"Asia Pacific";
            break;
        case 1:
            return @"Europe";
            break;
        case 2:
            return @"North America";
            break;
        default:
            return nil;
            break;
    }
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 3;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    switch(section) {
        case 0:
            return [arrAsiaPacific count];
            break;
        case 1:
            return [arrEurope count];
            break;
        case 2:
            return [arrNorthAmerica count];
            break;
        default:
            return 0;
            break;
    }
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:CellIdentifier] autorelease];
    }
}
```

code continues on next page

Code Listing 5.11 *continued*

```
// Configure the cell.
NSString *country;
switch(indexPath.section) {
    case 0:
        country = [arrAsiaPacific objectAtIndex:indexPath.row];
        break;
    case 1:
        country = [arrEurope objectAtIndex:indexPath.row];
        break;
    case 2:
        country = [arrNorthAmerica objectAtIndex:indexPath.row];
        break;
}

NSString *imageName = [NSString stringWithFormat:@"%@.png",country];
cell.imageView.image = [UIImage imageNamed:imageName];
cell.textLabel.text = country;

return cell;
}

- (void)dealloc {
    [arrAsiaPacific release];
    [arrEurope release];
    [arrNorthAmerica release];

    [super dealloc];
}
@end
```

Editing and searching table views

Now you'll take a closer look at some of the data source methods you can use to edit your table view. You'll see how you can drag the table view rows to reorder them, as well as how to delete rows. You'll also add a **UISearchBar** control to the table view and write some code to allow you to filter the table view contents.

To create an editable table view that can be searched:

1. Create a new navigation-based application, saving it as EditTableViewExample.
2. Repeat steps 1–6 from the earlier exercise “To create an application with a table view.”
3. Open RootViewController.h, and create some new instance variables (**Code Listing 5.12**).

Notice that you have changed the countries array to a *mutable* array. This is necessary since you will be changing the contents of the array when you search, edit, and delete items.

Code Listing 5.12 The header file for the editable table view application.

```
@interface RootViewController : UITableViewController <UISearchBarDelegate>
{
    NSMutableArray *arrCountries;
    NSMutableArray *arrTemp;
    UISearchBar *search;
    UIBarButtonItem *searchDoneButton;
}
@end
```

4. Switch to RootViewController.m, and modify the `viewDidLoad` method:

```
arrTemp = [arrCountries mutable  
→ Copy];
```

You first create a new array, copying the countries array. This is going to be used as temporary storage when you filter the table view by searching.

```
CGRect tableFrame =  
→ [self.tableView frame];  
  
CGRect searchRect = tableFrame;  
searchRect.size.height = 40;  
  
search = [[UISearchBar alloc]  
→ initWithFrame:searchRect];  
  
search.delegate = self;  
  
self.tableView.tableHeaderView =  
→ search;
```

Next you create a search bar and attach it to the top of the table view by assigning it to the `tableHeaderView` property.

```
self.navigationItem.rightBarButtonItem  
→ Item = self.editButtonItem;
```

Then you set the navigation bar's right button to be the control that edits the table view. The `editButtonItem` automatically switches your table view between edit modes.

```
searchDoneButton =  
→ [[UIBarButtonItem alloc]  
→ initWithBarButtonSystem  
→ Item:UIBarButtonItemSystemItemDone  
→ target:self action:@selector  
→ (searchDone:)];
```

Finally, you create a bar button that you will use for searching.

continues on next page

5. Implement the `tableView:canMoveRowAtIndexPath:` method. By returning YES, you are allowing any row to be moved. You can see which the user is attempting to move by looking at the `indexPath.row` property.

6. Implement the `tableView:moveRowAtIndexPath:toIndexPath:` method that is called when the user rearranges a row:

```
NSString *name = [arrCountries  
→ objectAtIndex:fromIndexPath.row];  
  
[arrCountries removeObjectAtIndex:  
→ fromIndexPath.row];  
  
[arrCountries insertObject:name  
→ atIndex:toIndexPath.row];
```

Although the table view will reorder itself, you also need to manually update the order of your countries array.

7. Implement the `tableView:commitEditingStyle:forRowAtIndexPath:` method.

You check the editing style to see whether the user has deleted a row and, if so, remove it from the two arrays. You then call the `deleteRowsAtIndexPaths:withRowAnimation:` method to remove the row from the table view using a fade animation.

8. Implement the `searchBarTextDidBeginEditing:` delegate method:

```
self.navigationItem.rightBarButtonItem  
→ Item = searchDoneButton;
```

This method is called when the search keyboard appears. You swap the table view edit button with the search button you created earlier, allowing the user to exit from the search.

- 9.** Implement the **searchBarSearchButton Clicked:** method, which is called when the user clicks “search” on the search keyboard.

This method calls a function that swaps the edit button in and hides the search bar keyboard.

- 10.** Implement the **searchBar:textDid Change:** method, which is called when the user starts typing in the search bar.

This in turn calls the **searchDone:** method:

```
NSString *searchFor = search.text;
```

```
[arrCountries release];
arrCountries = [arrTemp
→ mutableCopy];

if ([searchFor length] > 0) {
    NSPredicate *pred =
    → [NSPredicate
    → predicateWithFormat:@"SELF
    → contains[c] %@",searchFor];
    [arrCountries filterUsing
    → Predicate:pred];
}

[self.tableView reloadData];
```

continues on next page

Here you reset the countries array so that you are dealing with the complete set of countries. You next create a predicate with the search text. A *predicate* is a special object that allows you to perform filtering of data. By calling the `filterUsingPredicate:` method of your countries array, you can filter the array to contain only the text entered in the search bar. Finally, calling `reloadData` tells your data to redraw itself, re-requesting all of its data using its data source methods. Since you have filtered the countries array, you will see only those countries that match the search text.

11. Build and run the application.

If you click the edit button, the table view will go into edit mode. You can reorder the rows by dragging using the handle on the right side of each row ⑥. Clicking the delete graphic on the left side of a row allows you to remove the row from the table view. If you tap in the search bar area, a keyboard will appear. The data in the table view will be filtered as you type. **Code Listing 5.13** shows the completed code.

TIP To prevent the delete button from appearing while still allowing the rows to be reordered, you can return `UITableViewCellStyleNone` from the `tableView:editingStyleForRowAtIndexPath:` delegate method.



⑥ The editable table view application.

Code Listing 5.13 The completed code for the editable table view application.

```
#import "RootViewController.h"

@implementation RootViewController

- (void)viewDidLoad {

    self.title = @"Countries";
    arrCountries = [[NSMutableArray alloc] initWithObjects:
                     @"Australia",
                     @"Canada",
                     @"Germany",
                     @"France",
                     @"Great Britain",
                     @"Italy",
                     @"Japan",
                     @"New Zealand",
                     @"United States",nil];
    arrTemp = [arrCountries mutableCopy];

    CGRect tableFrame = [self.tableView frame];
    CGRect searchRect = tableFrame;
    searchRect.size.height = 40;
    search = [UISearchBar alloc] initWithFrame:searchRect];
    search.delegate = self;
    self.tableView.tableHeaderView = search;

    self.navigationItem.rightBarButtonItem = self.editButtonItem;
    searchDoneButton = [[UIBarButtonItem alloc]
                        initWithBarButtonSystemItem:UIBarButtonSystemItemDone
                        target:self
                        action:@selector(searchDone:)];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [arrCountries count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                 initWithStyle:UITableViewCellStyleDefault
                 reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell.
    cell.textLabel.text = [arrCountries objectAtIndex:indexPath.row];
    NSString *imageName = [NSString stringWithFormat:@"%@.png",[arrCountries objectAtIndex:indexPath.row]];
    cell.imageView.image = [UIImage imageNamed:imageName];

    return cell;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return YES;
}
```

code continues on next page

Code Listing 5.13 *continued*

```
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath *)fromIndexPath toIndexPath:(NSIndexPath *)toIndexPath
{
    //re-order array
    NSString *name = [arrCountries objectAtIndex:fromIndexPath.row];
    [arrCountries removeObjectAtIndex:fromIndexPath.row];
    [arrCountries insertObject:name atIndex:toIndexPath.row];
}

-(void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        [arrTemp removeObject:[arrCountries objectAtIndex:indexPath.row]];
        [arrCountries removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                           withRowAnimation:UITableViewRowAnimationFade];
    }
}

-(void)searchDone:(id)sender {
    self.navigationItem.rightBarButtonItem = self.editButtonItem;
    [search resignFirstResponder];
}

-(void)doSearch {
    NSString *searchFor = search.text;

    [arrCountries release];
    arrCountries = [arrTemp mutableCopy];

    if ([searchFor length] > 0) {
        NSPredicate *pred = [NSPredicate predicateWithFormat:@"SELF contains[c] %@",searchFor];
        [arrCountries filterUsingPredicate:pred];
    }

    [self.tableView reloadData];
}

-(void)searchBarTextDidBeginEditing:(UISearchBar *)searchBar {
    self.navigationItem.rightBarButtonItem = searchDoneButton;
}

-(void)searchBar:(UISearchBar *)searchBar textDidChange:(NSString *)searchText {
    [self doSearch];
}

-(void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    [self searchDone:nil];
}

-(void)dealloc {
    [arrCountries release];
    [arrTemp release];
    [search release];
    [searchDoneButton release];

    [super dealloc];
}

@end
```



❷ A navigation controller with a navigation bar and a table view.

Drilling down in table views

So far, the examples have presented a table view with only a single level of data. It's common, however, for data to be hierarchical, and you will often want to provide an interface where the user can select an item and then be presented with a subset of related items. This "drill-down" concept is used frequently and is surprisingly easy to implement using table views.

Before you see how to do this, it's worth learning about another important class you will be using to add this functionality: the navigation controller.

A navigation controller, represented by the `UINavigationController` class, is a special type of view controller designed to help you navigate through hierarchical content known as the *navigation stack*. View controllers are *pushed* onto the navigation stack to display them and then *popped* off the navigation stack when you are finished with them.

Navigation controllers typically display a navigation bar at the top of the screen that indicates the current (topmost) controller on the stack. (You have actually been using a navigation controller in all the table view examples so far.) This bar will update as view controllers are pushed onto and popped off the navigation stack. This bar is also usually where any edit controls related to the view would sit. The view of the current view controller is displayed below this navigation bar ❸.

TIP When combining a navigation-based interface with a tab bar, it is often the case that you'll want to have the tab bar disappear when a view controller is pushed onto the navigation stack. You can accomplish this by using the `hidesBottomBarWhenPushed` property of the view controller.

To create a table view with drill-down behavior:

1. Create a new navigation-based application, saving it as DrillDownExample.
2. Select File > New, and add a new **UIViewController** subclass .

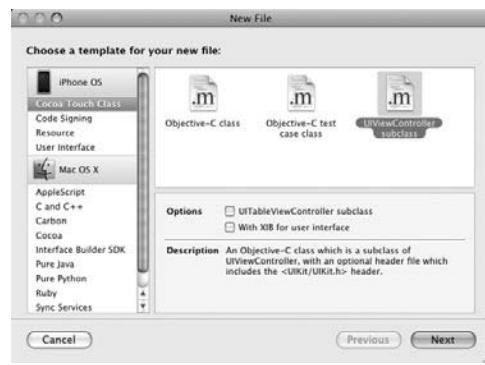
Save the file as DetailViewController.m. Make sure that “Also create ‘DetailViewController.h’” is selected.

3. Open the DetailViewController.h file, change the class type to **UITableView Controller**, and create a new property that you’ll use to hold the table view details (**Code Listing 5.14**).
4. Switch to DetailsViewController.m, uncomment the **viewWillAppear** method, and add the following code:

```
self.title = @"Details";  
[self.tableView reloadData];
```

Here you are setting the title and telling the table view to reload. Note that this time you are using the **viewDidAppear** method (instead of **viewDidLoad** as in the other exercises in this chapter). This method will be run every time your details view is shown. By using the **reloadData** method call, you know your table view will always contain the correct data.

5. Implement the rest of this class as shown in **Code Listing 5.15**, which is the same code you used for previous exercises in this chapter.
6. Switch to RootViewController.h, import the DetailsViewController.h file, and create some instance variables that you will use to hold your table data and the details view (**Code Listing 5.16**).



 Adding a second view controller class to hold your table view details.

Code Listing 5.14 The header file for the drill-down details of the table view controller.

```
#import <UIKit/UIKit.h>  
  
@interface DetailViewController : UITableViewController  
{  
    NSArray *content;  
}  
  
@property (nonatomic, retain) NSArray *content;  
  
@end
```

Code Listing 5.15 The completed code for the drill-down details of the table view controller.

```
#import "DetailViewController.h"

@implementation DetailViewController

@synthesize content;

- (void)viewWillAppear:(BOOL)animated {

    self.title = @"Details";
    [self.tableView reloadData];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [content count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Details";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:CellIdentifier] autorelease];
    }

    cell.textLabel.text = [content objectAtIndex:indexPath.row];
    return cell;
}

- (void)dealloc {
    [content release];
    [super dealloc];
}

@end
```

Code Listing 5.16 The main header file for the drill-down details application.

```
#import "DetailViewController.h"

@interface RootViewController : UITableViewController
{
    NSArray *arrCountries;
    NSArray *arrUSACities;
    NSArray *arrAustraliaCities;
    DetailViewController *details;
}
@end
```

7. Switch to RootViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
arrCountries = [[NSArray alloc]
→ initWithObjects: @"Australia",
→ @"United States",nil];
arrUSACities = [[NSArray alloc]
→ initWithObjects: @"Boston",
→ @"San Francisco",@"New York",
→ nil];
arrAustraliaCities = [[NSArray
→ alloc] initWithObjects:
→ @"Brisbane",@"Perth",@"Sydney",
→ nil];
details = [[DetailViewController
→ alloc] initWithStyle:UITableView
→ StyleGrouped];
```

Here you create three arrays: the first is your main array, and the other two are used for the details view. Notice that you can create the details view with a grouped style using the **initWithStyle:** method.



1 The details screen of the drill-down table view application.

8. Implement the `tableView:didSelectRowAtIndexPath:` delegate method that is called when you select a row:

```
if (indexPath.row == 0)
    [details setContent:
     → arrAustraliaCities];
else
    [details setContent:
     → arrUSACities];

[[self navigationController]
 → pushViewController:details
 → animated:YES];
```

Here you check which row the user has selected, assigning the appropriate array to the details view controller. You then push the view controller onto the stack of the navigation controller, causing it to be animated onto the screen.

9. The rest of the code is almost exactly as in previous exercises, except you set the `accessoryType` property when creating the cells:

```
cell.accessoryType = UITableView
→ CellAccessoryDisclosureIndicator;
```

This adds a disclosure triangle to the right side of the cell, providing a visual cue to the user that they will see detailed data by selecting the row.

10. Build and run your application 1.

Selecting either row will cause the details view to be shown. Pressing the back button in the navigation bar will take you to the main screen. **Code Listing 5.17** shows the completed code.

Code Listing 5.17 The completed drill-down table view application.

```
#import "RootViewController.h"

@implementation RootViewController

- (void)viewDidLoad {

    self.title = @"Countries";
    arrCountries = [[NSArray alloc] initWithObjects: @"Australia",@"United States",nil];
    arrUSACities = [[NSArray alloc] initWithObjects: @"Boston",@"San Francisco",@"New York",nil];
    arrAustraliaCities = [[NSArray alloc] initWithObjects: @"Brisbane",@"Perth",@"Sydney",nil];

    details = [[DetailViewController alloc] initWithStyle:UITableViewStyleGrouped];
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    if (indexPath.row == 0)
        [details setContent:arrAustraliaCities];
    else
        [details setContent:arrUSACities];

    [[self navigationController] pushViewController:details animated:YES];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {

    return [arrCountries count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:CellIdentifier] autorelease];
    }

    cell.textLabel.text = [arrCountries objectAtIndex:indexPath.row];
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

- (void)dealloc {

    [arrCountries release];
    [arrUSACities release];
    [arrAustraliaCities release];
    [details release];

    [super dealloc];
}

@end
```

Creating custom cells

So far, you've been using the default cell style in your table views (represented by the `UITableViewCellStyleDefault` type). Four styles are available **K**.

However, you are not restricted to displaying text and images. You can create your own custom cell types to fully control the appearance and contents of the cell **L**.

Next you'll learn how you can create a custom cell type that will allow you to add an on/off switch and a button to your table view.



K The four styles for table cells.



L The Settings application makes extensive use of custom cells.

To create a table view with custom cells:

1. Create a new navigation-based application, saving it as CustomCellExample.
2. Select File > New, and add a new Objective-C class.

Make sure that the Subclass dropdown is set to **UITableViewCell**, and save it as MyCustomCell.m. Make sure that “Also create ‘MyCustomCell.h’” is selected ❻.

3. Open MyCustomCell.h, and create a property (**Code Listing 5.18**). This will be the view that holds your custom control.
4. Switch to MyCustomCell.m, and add the following to the **initWithStyle:reuseIdentifier:** method:

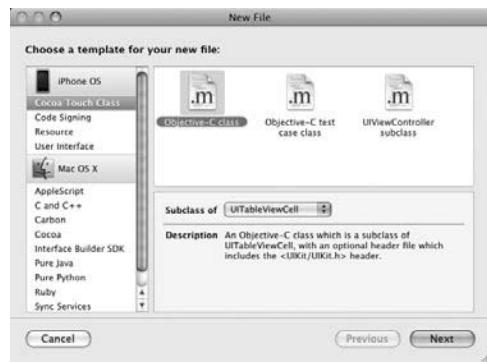
```
self.selectionStyle = UITableView  
→ CellSelectionStyleNone;
```

This disables selection of the entire cell; you don’t want the cell to be selected because you will be manipulating your control instead.

5. Implement the **setView:** method:

```
if (view)  
    [view removeFromSuperview];  
view = aView;  
[self.view retain];  
[self.contentView addSubview:aView];  
  
[self layoutSubviews];
```

This adds a view (from the view property) to the cell’s **contentView** and then tells the cell to draw its subviews.



❻ Adding a custom cell class to hold your table view details

Code Listing 5.18 The header file for the custom cell class.

```
#import <UIKit/UIKit.h>  
  
@interface MyCustomCell : UITableViewCell {  
    UIView *view;  
}  
  
@property (nonatomic, retain) UIView *view;  
  
@end
```

- 6.** Implement the `layoutSubviews` method:

```
float xOffset = 10.0;
```

```
[super layoutSubviews];  
CGRect contentRect =  
→ [self.contentView bounds];
```

```
CGRect viewFrame = CGRectMake  
→ (contentRect.size.width -  
→ self.view.bounds.size.width -  
→ xOffset,round((contentRect.size.  
→ height - self.view.bounds.size.  
→ height) / 2.0),self.view.bounds.  
→ size.width,self.view.bounds.size.  
→ height);
```

```
view.frame = viewFrame;
```

Here you add the view (defined in the `view` property) to the right side of the cell's `contentView`. **Code Listing 5.19** shows the complete code for your custom cell class.

- 7.** Open `RootViewController.h`, and import the header file for your custom cell.
- 8.** Create instance variables to hold two controls for your custom cells and an array to hold the table view content (**Code Listing 5.20**).

continues on page 227

Code Listing 5.19 The completed custom cell class.

```
#import "MyCustomCell.h"

@implementation MyCustomCell

@synthesize view;

- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier {
    if (self = [super initWithStyle:style reuseIdentifier:reuseIdentifier]) {
        self.selectionStyle = UITableViewCellStyleNone;
    }
    return self;
}

- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    [super setSelected:selected animated:animated];
}

- (void)setView:(UIView *)aView {
    if (view)
        [view removeFromSuperview];
    view = aView;
    [self.view retain];
    [self.contentView addSubview:aView];
    [self layoutSubviews];
}

- (void)layoutSubviews {
    float xOffset = 10.0;
    [super layoutSubviews];
    CGRect contentRect = [self.contentView bounds];

    CGRect viewFrame = CGRectMake(contentRect.size.width - self.view.bounds.size.width - xOffset,
                                  round((contentRect.size.height - self.view.bounds.size.height) / 2.0),
                                  self.view.bounds.size.width,
                                  self.view.bounds.size.height);
    view.frame = viewFrame;
}

- (void)dealloc {
    [view release];
    [super dealloc];
}

@end
```

Code Listing 5.20 The header file for the main table view controller class of the custom cell example.

```
#import "MyCustomCell.h"

@interface RootViewController : UITableViewController
{
    UISwitch *mySwitch;
    UIButton *myButton;
    NSArray *arrCells;
}
@end
```

9. Switch to RootViewController.m, uncomment the `viewDidLoad` method, and add the following code:

```
mySwitch = [[UISwitch alloc]
→ initWithFrame:CGRectMakeZero];
[mySwitch addTarget:self action:
→ @selector(switchAction:)
→ forControlEvents:UIControlEventValueChanged];
[mySwitch setOn:YES];

myButton = [UIButton
→ buttonWithType:
→ UIButtonTypeRoundedRect];
[myButton setFrame:CGRectMake
→ (0,0,80,30)];
[myButton addTarget:self action:
→ @selector(buttonAction:)
→ forControlEvents:UIControlEventTouchUpInside];
[myButton setTitle:@"Tap Me!"
→ forState:UIControlStateNormal];
arrCells = [[NSArray alloc]
→ initWithObjects:@"Item 1",
→ @"Item 2",@"Item 3",nil];
```

Here you create a switch and a button that you are going to be adding to your custom cells. Notice how you define the target-action methods for both of them within this class also. In this example, you just log the actions to the console. You also create an array to hold the text for the table view just as you did in previous exercises.

continues on next page

10. Implement the `tableView:numberOfRowsInSection:`

method to return the size of your array, and implement the `tableView:cellForRowAtIndexPath:` method:

```
UITableViewCell *cell = nil;  
static NSString  
→ *CellIdentifier = @"Cell";  
static NSString *ViewCell  
→ Identifier = @"ViewCell";  
  
switch (indexPath.row) {  
  
case 0:  
    cell = [tableView dequeueReusableCellWithIdentifier:  
→ ViewCellIdentifier];  
    if (cell == nil)  
        cell = [[[MyCustomCell  
→ alloc] initWithStyle:  
→ UITableViewCellStyle  
→ Default reuseIdentifier:  
→ ViewCellIdentifier]  
→ autorelease];  
  
    cell.textLabel.text =  
→ [arrCells objectAtIndex:  
→ indexPath.row];  
    ((MyCustomCell *)cell).view =  
→ mySwitch;  
    break;  
  
case 1:  
    cell = [tableView dequeueReusableCellWithIdentifier:  
→ ReusableCellWithIdentifier:  
→ ViewCellIdentifier];  
    if (cell == nil)
```

```

cell = [[[MyCustomCell
→ alloc] initWithFrame:
→ UITableViewCellStyle
→ Default reuseIdentifier:
→ ViewCellIdentifier]
→ autorelease];

cell.textLabel.text =
→ [arrCells objectAtIndex:
→ indexPath.row];
((MyCustomCell *)cell).view =
→ myButton;
break;
default:
cell = [tableView dequeueReusableCellWithIdentifier:
→ ReusableCellWithIdentifier:
→ CellIdentifier];
if (cell == nil)
cell = [[[UITableViewCell
→ alloc] initWithFrame:
→ UITableViewCellStyle
→ Default reuseIdentifier:
→ CellIdentifier]
→ autorelease];

cell.textLabel.text =
→ [arrCells objectAtIndex:
→ indexPath.row];
break;
}

return cell;

```

continues on next page

Since you are using custom cells, you need to determine which row is being requested via `indexPath.row`. For the first two rows, you create a custom cell. Notice how you set the view of the custom cell to the switch or the button. For other rows (the `default` case), the code is the same as in the previous exercise “To create an application with a table view.”

11. Build and run the application.

The completed application displays two custom cells (N). Selecting the switch or tapping the button will log a message to the console. **Code Listing 5.21** shows the completed code.

TIP This is obviously a simple example, adding only a single view to the cell, but you should be able to see how you could extend it to add multiple controls to your own custom cells. For more information on working with table views, see the *Table View Programming Guide for iPhone OS* in the developer documentation.



N The completed custom cell application.

Code Listing 5.21 The completed code for the custom cell example.

```
#import "RootViewController.h"

@implementation RootViewController

- (void)switchAction:(id)sender
{
    NSLog(@"switch changed");
}

- (void)buttonAction:(id)sender
{
    NSLog(@"button tapped");
}

- (void)viewDidLoad {

    mySwitch = [[UISwitch alloc] initWithFrame:CGRectMakeZero];
    [mySwitch addTarget:self action:@selector(switchAction:) forControlEvents:UIControlEventValueChanged];
    [mySwitch setOn:YES];

    myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [myButton setFrame:CGRectMake(80, 80, 30)];
    [myButton addTarget:self action:@selector(buttonAction:) forControlEvents:UIControlEventTouchUpInside];
    [myButton setTitle:@"Tap Me!" forState:UIControlStateNormal];

    arrCells = [[NSArray alloc] initWithObjects:@"Item 1",@"Item 2",@"Item 3",nil];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [arrCells count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = nil;
    static NSString *CellIdentifier = @"Cell";
    static NSString *ViewCellIdentifier = @"ViewCell";

    switch (indexPath.row) {

        case 0:
            cell = [tableView dequeueReusableCellWithIdentifier:ViewCellIdentifier];
            if (cell == nil)
                cell = [[[MyCustomCell alloc]
                         initWithFrame:CGRectMake(0, 0, 300, 44)]
                        reuseIdentifier:ViewCellIdentifier] autorelease];
            cell.textLabel.text = [arrCells objectAtIndex:indexPath.row];
            ((MyCustomCell *)cell).view = mySwitch;
            break;

        case 1:
            cell = [tableView dequeueReusableCellWithIdentifier:ViewCellIdentifier];
            if (cell == nil)
                cell = [[[MyCustomCell alloc]
                         initWithFrame:CGRectMake(0, 0, 300, 44)]
                        reuseIdentifier:ViewCellIdentifier] autorelease];
            cell.textLabel.text = [arrCells objectAtIndex:indexPath.row];
            ((MyCustomCell *)cell).view = myButton;
            break;

        default:
            cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
            if (cell == nil)
                cell = [[[UITableViewCell alloc]
                         initWithFrame:CGRectMake(0, 0, 300, 44)]
                        reuseIdentifier:CellIdentifier] autorelease];
    }
}
```

code continues on next page

Code Listing 5.21 *continued*

```
cell.textLabel.text = [arrCells objectAtIndex:indexPath.row];
break;
}
return cell;
}
- (void)dealloc {
[arrCells release];
[mySwitch release];
[super dealloc];
}
@end
```

6

Files and Networking

The iPhone runs a slimmed-down version of the Mac OS X operating system, so the file system is similar to its desktop cousin. However, iPhone applications run within a security *sandbox*, a unique area on the iPhone reserved for an application where it can run without affecting the system or any other iPhone applications. So, the files and directories you can actually read and write are somewhat limited. Of course, this also means that other applications can't gain access to your application's files!

The iPhone also has some fairly extensive networking capabilities by way of its Wi-Fi, Bluetooth, and 3G connectivity options. In this chapter, you'll learn the classes, methods, and functions used to perform many of the common file and networking tasks encountered as an iPhone developer.

In This Chapter

Files	234
Networking	248

Files

As you learned in Chapter 1, “Objective-C and Cocoa,” several Cocoa classes have methods to automatically deal with reading and writing files (**Code Listing 6.1**). Some of these classes are as follows:

- **UIImage**—The `imageNamed:` method will automatically try to load an image from the resources directory of your application bundle.
- **NSString**—The `stringWithContentsOfFile:` and `writeToFile:` methods let you load and save strings as files. Additionally, **NSString** has some other commonly used methods for dealing with file paths.
- **NSDictionary** and **NSArray**—The `dictionaryWithContentsOfFile:`, `arrayWithContentsOfFile:`, and `writeToFile:` methods allow you to load and save dictionaries and arrays of objects. Note that the objects in your dictionary or array must be property list objects (**NSNumber**, **NSString**, **NSData**, **NSArray**, or **NSDictionary**).
- **NSData**—This is normally used when dealing with binary files.

Code Listing 6.1 Many common classes have methods to load and save files.

```
- (void)writeStringToDirectory:(NSString *)dir {
    //create filename
    NSString *f = @"somefile.txt";
    NSString *path = [dir stringByAppendingPathComponent:f];

    //create string
    NSString *foo = @"some content";

    //save
    [foo writeToFile:path atomically:YES encoding:NSUTF8StringEncoding error:nil];

    //read back in and log
    NSString *results = [NSString stringWithContentsOfFile:path encoding:NSUTF8StringEncoding error:nil];
    NSLog(@"%@",results);
}

- (void)writeDictionaryToDirectory:(NSString *)dir
{
    //create file
    NSString *f = @"somefile.txt";
    NSString *path = [dir stringByAppendingPathComponent:f];

    //create dictionary
    NSString *string1 = @"some value";
    NSString *string2 = @"another value";
    NSNumber *num = [NSNumber numberWithInt:12345];
    NSArray *arr = [NSArray arrayWithObjects:
                    @"item1",
                    @"item2",
                    @"item3",nil];
    NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
                          string1,@"item1",
                          string2, @"item2",
                          num, @"item3",
                          arr, @"item4", nil];

    //save
    [dict writeToFile:path atomically:YES];

    //read back and log
    NSDictionary *results = [NSDictionary dictionaryWithContentsOfFile:path];
    NSLog(@"%@",results);
}
```

The file system

NSFileManager wraps many of the common actions you will need to perform when dealing with the iPhone file system including creating, deleting, moving, and copying files. You can also use it for getting directory listings, reading and writing file attributes, and more.

- For example, to copy a file from one location to another, you could write this:

```
NSFileManager *fm = [NSFileManager  
→ defaultManager];  
  
NSError *error;  
  
BOOL ok = [fm copyItemAtPath:  
→ @"somePath" toPath:  
→ @"someOtherPath" error:&error];  
  
if (!ok)  
    NSLog(@"%@", error.localizedDescription  
→ Description);
```

- To get an array of filenames for a path, you can use this:

```
NSFileManager *fm = [NSFileManager  
→ defaultManager];  
  
NSArray *fileContents =  
→ [fm directoryContentsAtPath:@"/"];
```

TIP You can read more in the **NSFileManager** class reference of the developer documentation.

The screenshot shows the Xcode Debugger Console window titled "FilesExample - Debugger Console". The console output is as follows:

```
2009-07-30 17:09:05.310 FilesExample[6591:20b] (
    Documents,
    "FilesExample.app",
    Library,
    tmp
)
```

Below the console, a status bar indicates "FilesExample launched" and "Succeeded".

A The contents of the home directory.

Common directories

The iPhone is quite locked down in regard to which directories you can read and write to, and in practice, you will normally be working with only a couple of directories.

You can retrieve the path of your application by using the **NSHomeDirectory()** function:

```
NSString *homeDir =
→ NSHomeDirectory();
```

If you log the contents of this directory to the console, you'll notice three directories in addition to your application bundle A:

- **tmp**—This is the temporary directory, which can also be retrieved using the **NSTemporaryDirectory()** function:

```
NSString *tempDir =
→ NSTemporaryDirectory();
```

You can use this directory for reading and writing any temporary files you may need in your application. Files in this directory are not guaranteed to continue to exist once you exit your application (they are automatically cleaned out periodically by the system) and are not copied to your computer when you sync your iPhone with iTunes.

- **Documents**—This directory is the designated area for your application to read and write files to. Any files written here are copied to your computer when you sync with iTunes. You can retrieve the documents directory by using the following code:

```
NSArray *paths = NSSearchPathFor
→ DirectoriesInDomains(NSDocument
→ Directory, NSUserDomainMask,
→ YES);
```

```
NSString *docDir = [paths
→ objectAtIndex:0];
```

continues on next page

- **Library**—This directory contains cache and preference files used by **NSUserDefaults**. Although you can read and write to this directory, you will not normally need to do so.

You can also read files contained within your application bundle, such as images, movies, sounds, or other files that you have added to your project. These will generally be copied into the resources directory of the bundle. For example, to get the path of an image named apple.png, you can write the following:

```
NSString *resourcesDir =  
→ [[NSBundle mainBundle]  
→ resourcePath];  
  
NSString *imageDir =  
→ [resourcesDir  
→ stringByAppendingPathComponent:  
→ @"apple.png"];
```

TIP Although both your application bundle and the home directory can be written to in the iPhone Simulator, they cannot be written to on an actual iPhone. For this reason, you should use the tmp or documents directory if you need to save any files.

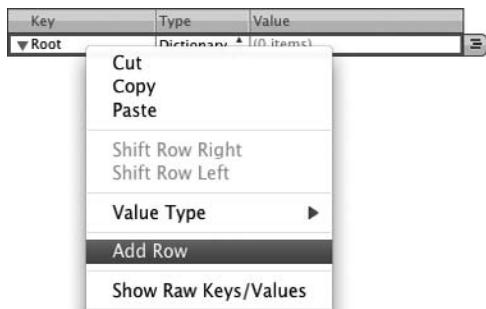
TIP Setting the `UIFileSharingEnabled` key to YES in your applications `info.plist` will enable file sharing so that users can get access to the contents of the documents directory of your application through iTunes **B**. This is an extremely handy way of moving files between your computer and your iPhone applications.



B Once enabled, file sharing can be accessed via iTunes.



C Creating a property list.



D Adding a new row to a property list.

Working with files

So far, you've seen some of the classes, methods, and functions you can use when managing files on the iPhone. Now you'll look at an example where being able to load and save an **NSDictionary** as a file may come in handy. You'll create an application (a simple username and password screen) that populates its user interface from an external settings file. Any changes the user makes to the interface will be saved to the settings file so that they can be restored the next time the application is launched.

One way to do this is to ship your application with a default settings file. When the application launches for the first time, you copy these default settings to the documents directory. It is this copy you then use in your application for loading and saving the user's settings. Since the documents directory is copied to the user's computer each time they sync with iTunes, you know that any changes the user has made will remain safe.

To create an application to load settings from an external file:

1. Create a new view-based application, saving it as `SettingsExample`.
2. Choose `File > New`. In the `New File` dialog box, select the `Other` category, select `Property List`, and click `Next` C.
3. Name the file `settings.plist`, and click `Finish`.
4. Select the `settings.plist` file in the `Groups & Files` pane to show the file in the property list editor view.
5. Right-click the `Root` key, and select `Add Row` D.

continues on next page

6. Change the key from New Item to username, and leave the type as String.
7. Repeat steps 5–6, adding a second item called password. If you want, you can enter a default value for both keys **E**.
8. Open the FilesExampleViewController.h file, and create the instance variables you'll use to hold the settings dictionary, filename, and user interface elements (**Code Listing 6.2**).
9. Switch to FilesExampleViewController.m, and create the method **getSettingsDictionary:**, which will be used to read the settings from the file:

```

NSArray *paths = NSSearchPathFor
→ DirectoriesInDomains(NSDocument
→ Directory, NSUserDomainMask,
→ YES);

NSString *documentsDir = [paths
→ objectAtIndex:0];

NSString *filePath =
→ [documentsDir
→ stringByAppendingPathComponent:
→ settingsFileName];

NSFileManager *fm =
→ [NSFileManager defaultManager];

BOOL exists =
→ [fm fileExistsAtPath: filePath];

```

You first get the user's documents directory and use it to construct the filename for the settings file.

```

NSString *resourcesDir =
→ [[NSBundle mainBundle]
→ resourcePath];

NSString *sourcePath =
→ [resourcesDir
→ stringByAppendingPathComponent:
→ fileName];

[fm copyItemAtPath:sourcePath
→ toPath:filePath error:NULL];

```

Key	Type	Value
Root	Dictionary	(2 items)
username	String	<enter username>
password	String	

F The completed property list.

Code Listing 6.2 The header file for the Settings example.

```

#import <UIKit/UIKit.h>

@interface SettingsExampleViewController: UIViewController
{
    NSMutableDictionary *dictSettings;
    NSString *settingsFileName;
    UITextField *txtUserName;
    UITextField *txtPassword;
}
@end

```

You next check whether the file exists, and if it doesn't, you copy the default settings file (created in step 4) from the application bundle into the documents directory.

```
return [NSDictionary  
→ dictionaryWith  
→ ContentsOfFile:filePath];
```

You then create a dictionary from this file and return it from the method.

10. Create the **createUI** method used to build the user interface.

There is nothing unusual here: You are adding two **UILabels**, two **UITextField**s, and a button to the main view. The text values for the username and password fields are retrieved from the **dictSettings** dictionary.

11. Uncomment and implement the **viewDidLoad** method:

```
settingsFileName =  
→ [[NSString alloc] initWithString:  
→ @"settings.plist"];  
  
dictSettings =  
→ [[NSMutableDictionary alloc]  
→ initWithDictionary:[self  
→ settingsDictionary]];  
  
[self createUI];
```

Here you set the settings filename and retrieve and initialize a dictionary from the file in the documents directory (which will be automatically created if necessary). Then you store the dictionary in an instance variable before creating the user interface.

12. The final step is to implement the **saveClick:** method that is called when the user taps the Save button:

continues on next page

```

NSArray *paths = NSSearchPathFor
→ DirectoriesInDomains(NSDocument
→ Directory, NSUserDomainMask,
→ YES);

NSString *documentsDir = [paths
→ objectAtIndex:0];

NSString *filePath = [documents
→ Dir stringByAppendingPathComponent:
→ Component:settingsFileName];

[dictSettings
→ setValue:txtUserName.
→ text forKey:@"username"];

[dictSettings setValue:txtPassword.
→ text forKey:@"password"];

[dictSettings writeToFile:filePath
→ atomically:NO];

```

This is almost the reverse of the `getSettingsDictionary:` method. You set the username and password keys of the settings dictionary to the text field values and then write it to the documents directory. **Code Listing 6.3** shows the completed code.

TIP In this example, you saved not only the username but also the user's password in the settings file. In a real-world application, you would normally store sensitive information such as passwords in a secure database called the *keychain*. For more information, refer to the *Keychain Services Programming Guide* in the developer documentation.

Code Listing 6.3 The completed code.

```

#import "SettingsExampleViewController.h"

@implementation SettingsExampleViewController

- (void)saveClick:(id)sender
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDir = [paths objectAtIndex:0];

    NSString *filePath = [documentsDir stringByAppendingPathComponent:settingsFileName];

    [dictSettings setValue:txtUserName.text forKey:@"username"];
    [dictSettings setValue:txtPassword.text forKey:@"password"];

    //save
    [dictSettings writeToFile:filePath atomically:NO];
}

- (NSDictionary *)getSettingsDictionary {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDir = [paths objectAtIndex:0];
    NSString *filePath = [documentsDir stringByAppendingPathComponent:settingsFileName];
}

```

code continues on next page

Code Listing 6.3 continued

```
//check if file exists in doc dir
NSFileManager *fm = [NSFileManager defaultManager];
BOOL exists = [fm fileExistsAtPath:filePath];

if (!exists)
{
    NSString *resourcesDir = [[NSBundle mainBundle] resourcePath];
    NSString *sourcePath =[resourcesDir stringByAppendingPathComponent:settingsFileName];
    [fm copyItemAtPath:sourcePath toPath:filePath error:NULL];
}

return [NSDictionary dictionaryWithContentsOfFile:filePath];
}

-(void)createUI
{
    UILabel *lblUserName = [[UILabel alloc] initWithFrame:CGRectMake(10,10,100,40)];
    lblUserName.backgroundColor = [UIColor clearColor];
    lblUserName.text = @"Username:";
    [self.view addSubview:lblUserName];
    [lblUserName release];

    txtUserName = [[UITextField alloc] initWithFrame:CGRectMake(115,20,185,24)];
    txtUserName.backgroundColor = [UIColor whiteColor];
    txtUserName.text = [dictSettings objectForKey:@"username"];
    [self.view addSubview:txtUserName];

    UILabel *lblPassword = [[UILabel alloc] initWithFrame:CGRectMake(10,50,100,40)];
    lblPassword.backgroundColor = [UIColor clearColor];
    lblPassword.text = @"Password:";
    [self.view addSubview:lblPassword];
    [lblPassword release];

    txtPassword = [[UITextField alloc] initWithFrame:CGRectMake(115,60,185,24)];
    txtPassword.backgroundColor = [UIColor whiteColor];
    txtPassword.text = [dictSettings valueForKey:@"password"];
    txtPassword.clearsOnBeginEditing = YES;
    txtPassword.secureTextEntry = YES;
    [self.view addSubview:txtPassword];

    UIButton *btnSave = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    btnSave.frame = CGRectMake(200, 100, 100, 34);
    [btnSave setTitle:@"Save" forState:UIControlStateNormal];
    [btnSave addTarget:self action:@selector(saveClick:) forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:btnSave];
}

-(void)viewDidLoad {
    settingsFileName = [[NSString alloc] initWithString:@"settings.plist"];
    dictSettings = [[NSMutableDictionary alloc] initWithDictionary:[self getSettingsDictionary]];
    [self createUI];
}

-(void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

-(void)dealloc {
    [txtUserName release];
    [txtPassword release];
    [settingsFileName release];
    [dictSettings release];
    [super dealloc];
}

@end
```

Previewing documents

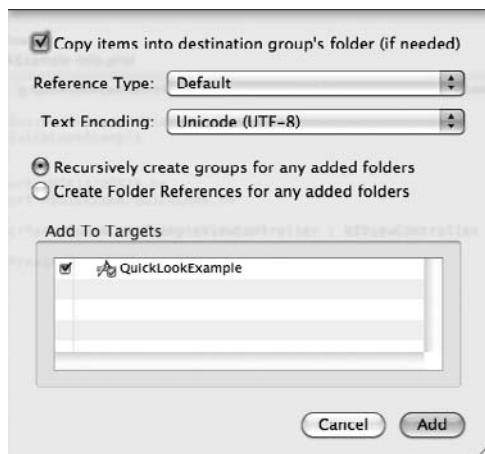
Just like on the Macintosh, the iPhone SDK contains a technology known as *Quick Look* for previewing documents. By using a `CLPreviewController` object, you can easily view multiple documents of several formats, including Microsoft Office, Apple iWork, PDF, rich text, images, and more.

Next you'll see how, with just a couple of lines of code, you can create a simple PDF viewer, complete with paging and touch-enabled resizing. This example can, of course, be extended to work with any of the document types supported by Quick Look.

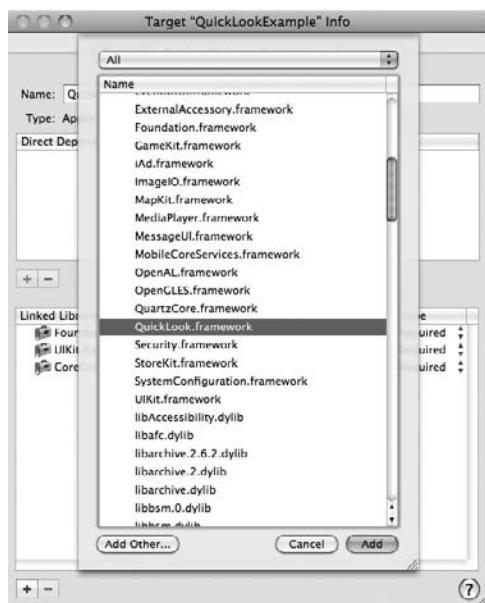
In this example, you'll view the iPhone Users Guide, which can be downloaded from Apple at http://manuals.info.apple.com/en/iphone_user_guide.pdf.

To create a PDF viewer:

1. Create a new view-based application, saving it as QuickLookExample.
2. Drag the iPhone Users Guide PDF into the Resources section in the Groups & Files pane. Make sure that the “copy items into destination group's folder (if needed)” check box is selected ⑤.
3. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
4. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add the QuickLook.framework ⑥.



⑤ Copying the PDF into the project.



⑥ Adding the Quick Look framework.

5. Open QuickLookExampleViewController.h, include the QuickLook.h header, add the protocol **QLPreviewControllerDataSource**, and create an instance variable to hold your Quick Look view controller ([Code Listing 6.4](#)).
6. Next, switch to QuickLookExample ViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
qlViewController =  
→ [[QLPreviewController alloc]  
→ init];  
  
qlViewController.dataSource =  
→ self;  
  
[self createUI];
```

Here you are simply creating a Quick Look preview controller and setting its data source. You also call the **createUI** method, which adds the button that will be used to show the Quick Look preview controller.

7. Next, create the **showPreview** method:

```
[self presentModalViewController:  
→ qlViewController animated:YES];
```

which simply shows the Quick Look preview controller on the screen.

continues on next page

Code Listing 6.4 The header file for the PDF viewer.

```
#import <UIKit/UIKit.h>  
#import <QuickLook/QuickLook.h>  
  
@interface QuickLookExampleViewController : UIViewController <QLPreviewControllerDataSource> {  
    QLPreviewController *qlViewController;  
}  
  
@end
```

8. Now you need to implement the two `QLPreviewControllerDataSource` data source methods so that the Quick Look preview controller knows what to display. First you implement the `numberOfPreviewItemsInPreviewController`: method. In this example, since we have only a single PDF file, you simply return the number 1.

```
return 1;
```

You then implement the `previewController:previewItemAtIndex:` method:

```
NSString *documentLocation =  
→ [[NSBundle mainBundle]  
→ pathForResource:@"iphone_user_  
→ guide" ofType:@"pdf"];  
  
NSURL *myQLDocument = [NSURL file  
→ URLWithPath:documentLocation];  
  
return myQLDocument;
```

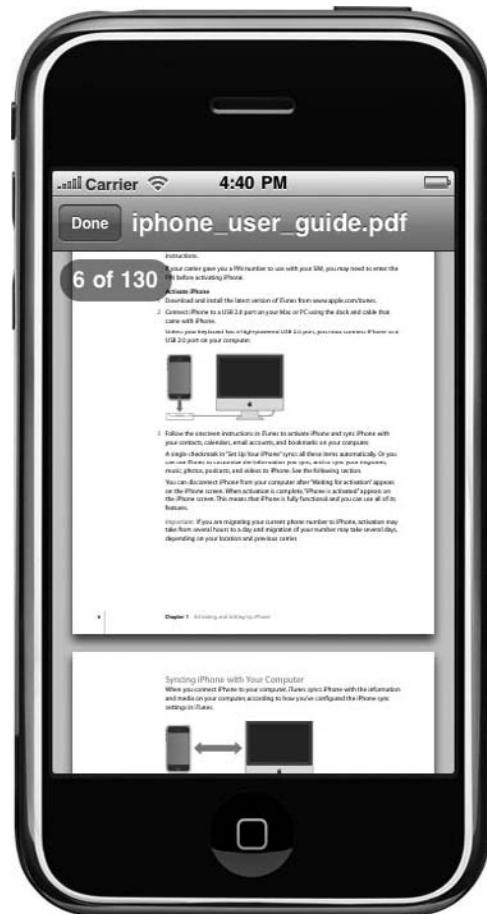
Here you retrieve the path of your PDF and use it to create and return an `NSURL` object.

Code Listing 6.5 shows the completed code.

9. Build and run the application.

Tapping the Show Preview button causes the Quick Look preview controller to load. You can navigate through the pages of the PDF by flicking your finger up and down on the screen. Note how your current page is displayed in the top-right corner and the document name is displayed in the title bar **H**. By pinching or double-tapping the screen, you should be able to zoom in to read the document in greater detail.

For more information on Quick Look, refer to the *Quick Look Framework Reference* in the developer documentation.



H The PDF Viewer application.

Code Listing 6.5 The completed PDF viewer code.

```
#import "QuickLookExampleViewController.h"

@implementation QuickLookExampleViewController

- (NSInteger)numberOfPreviewItemsInPreviewController:(QLPreviewController *)controller {
    return 1;
}

- (id <QLPreviewItem>)previewController:(QLPreviewController *)controller
    previewItemAtIndex:(NSInteger)index {
    NSString *documentLocation = [[NSBundle mainBundle]
        pathForResource:@"iphone_user_guide" ofType:@"pdf"];
    NSURL *myQLDocument = [NSURL fileURLWithPath:documentLocation];
    return myQLDocument;
}

- (void)showPreview:(id)sender {
    [self presentModalViewController:qlViewController animated:YES];
}

- (void)createUI
{
    UIButton *btnShowQuickLook = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    btnShowQuickLook.frame = CGRectMake(68, 400, 200, 34);
    [btnShowQuickLook setTitle:@"Show Preview" forState:UIControlStateNormal];
    [btnShowQuickLook addTarget:self action:@selector(showPreview:)
        forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:btnShowQuickLook];
}

- (void)viewDidLoad {
    qlViewController = [[QLPreviewController alloc] init];
    qlViewController.dataSource = self;

    [self createUI];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [qlViewController release];
    [super dealloc];
}

@end
```

Networking

Now you'll learn how you can accomplish some of the more common networking-related tasks on the iPhone. You'll see how to retrieve content from and send content to web pages. You'll also see how easy it is to work with web pages that require authentication. Finally, you'll use the Game Kit API to create a simple peer-to-peer chat application.

Retrieving content from web pages

The easiest way to get the HTML contents of a web page is by using the **stringWithContentsOfURL:** method of **NSString**. Next you'll take a look at using this method to retrieve some stock quote data from the Yahoo! finance web page and display it in your application **A**.

To retrieve stock quotes from a web page:

1. Create a new view-based application, saving it as GetWebContent.
2. Open GetWebContentViewController.h, and add an instance variable to display the web page you will be retrieving:

```
UITextView *resultsView;
```



A Retrieving and displaying stock quotes in an application.

- 3.** Switch to GetWebContentViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
CGRect resultsFrame = CGRectMake  
→ (10,10,300,100);  
  
resultsView = [[UITextView alloc]  
→ initWithFrame:resultsFrame];  
  
resultsView.font = [UIFont  
→ systemFontOfSize:14.0];  
  
[self.view addSubview:resultsView];
```

You first create a **UITextView** control that you'll use to display the stock quote information.

```
NSString *symbol = @"AAPL,GOOG,  
→ MSFT,YHOO,PALM";  
  
NSString * urlString = [NSString  
→ stringWithFormat:  
→ @"http://finance.yahoo.com/d/  
→ quotes.csv?f=no&s=%@", symbol];
```

You next create a string containing the URL of the Yahoo! finance page. Note how **stringWithFormat:** is used to append the list of stock symbols to the query string.

```
NSURL *url = [NSURL  
→ URLWithString:urlString];  
  
NSString *quotes = [NSString  
→ stringWithContentsOfURL:url  
→ encoding:NSUTF8StringEncoding  
→ error:nil];  
  
resultsView.text = quotes;
```

Finally, you create an **NSURL** object and use it to populate a string with the contents of the web page before displaying it in your text view.

Code Listing 6.6 shows the completed code. Notice that there are **NSLog()** statements at the beginning and end of the method. If you examine the console, you will see that the web page content actually takes some time to be downloaded (depending on your Internet connection speed, of course). You may also have noticed that your application was unresponsive while the download was taking place. This is because **stringWithContentsOfURL:** uses a *synchronous* connection, so your application will effectively pause until the download is completed. Worse still, if the page cannot be loaded (either as a result of a bad URL or the server not responding), you won't be told of any errors. You can check this for yourself by changing the URL to an invalid page.

Code Listing 6.6 Retrieving stock quotes from a web page.

```
#import "GetWebContentViewController.h"

@implementation GetWebContentViewController

- (void)viewDidLoad {
    NSLog(@"started");
    CGRect resultsFrame = CGRectMake(10,10,300,100);
    UITextView *resultsView = [[UITextView alloc] initWithFrame:resultsFrame];
    resultsView.font = [UIFont systemFontOfSize:14.0];
    [self.view addSubview:resultsView];

    NSString *symbol = @"AAPL,GOOG,MSFT,YHOO,PALM";
    NSString *urlString = [NSString stringWithFormat:@"http://finance.yahoo.com/d/quotes.csv?f=nos&s=%@", symbol];
    NSURL *url = [NSURL URLWithString:urlString];
    NSString *quotes = [NSString stringWithContentsOfURL:url encoding:NSUTF8StringEncoding error:nil];

    resultsView.text = quotes;
    NSLog(@"done");
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [resultsView release];
    [super dealloc];
}

@end
```

Apple provides a more robust way of retrieving web pages—the **NSURLConnection** class—which allows you to use an *asynchronous* connection as well as deal with situations such as timeouts, server redirects, and errors.

To update the application to use an asynchronous connection:

1. Open GetWebContentViewController.h, and add an instance variable to hold the web page data:

```
NSMutableData *receivedData;
```

2. Switch to GetWebContentViewController.m, and modify the **viewDidLoad** method, replacing the call to **stringWithContentOfURL**:

```
receivedData=[[NSMutableData  
→ alloc] initWithData:nil];  
  
NSURLRequest *req=[[NSURLRequest  
→ alloc] initWithURL:url];  
  
NSURLConnection *conn =  
→ [[NSURLConnection alloc]  
→ initWithRequest:req  
→ delegate:self];
```

Here you first initialize an instance variable to hold the web page contents as it's received. Next you create an **NSURLRequest** object, assigning it the same **NSURL** you used last time, and then you use this **NSURLRequest** object to create an **NSURLConnection** object. **NSURLConnection** asynchronously loads the web page and sends messages to its delegate at various points during the process.

continues on next page

- 3.** Finally, you need to implement the following three delegate methods of **NSURLConnection**:

The **connection:didReceiveData:** delegate method will be called whenever some data is received from the **NSURLConnection**. Here you simply append the data to any that has already been received.

The **connection:didFailWithError:** delegate method will be called if an error occurs. In this example, you display the error, but in a larger application, you might want to clean up your **receivedData** object.

The **connectionDidFinishLoading:** delegate method is called when the **NRURLConnection** has finished loading the web page. You then create a string for the **receivedData** object and display it in the text view as before.

Code Listing 6.7 shows the updated application. If you again put **NSLog()** statements at the beginning and end of the **viewDidLoad** method, you should notice that the time stamp difference is almost zero and that your application starts up and displays the text view even before the stock data has finished downloading. If you also change the URL to an invalid web page, you should now see the error being handled and displayed in an alert view.

Apple recommends that any application that requires an Internet connection should first test for availability of the connection and inform the user if one isn't available. Although the code to perform this check is beyond the scope of this book, Apple provides a sample application (the "Reachability" example) that shows how this can be done. You can download it from the "Coding How-Tos" section of the *iPhone Developer Connection* Web site.

Code Listing 6.7 The code, updated to retrieve stock quotes asynchronously.

```
#import "GetWebContentViewController.h"

@implementation GetWebContentViewController

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
    [receivedData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error {
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Error"
                                                    message:[error localizedDescription]
                                                    delegate:nil
                                               cancelButtonTitle:@"OK"
                                               otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    NSString *quotes = [[NSString alloc]
                        initWithBytes:[receivedData bytes]
                        length:[receivedData length]
                        encoding:NSUTFStringEncoding];
    resultsView.text = quotes;
    [quotes release];
}

- (void)viewDidLoad {
    NSLog(@"started");

    CGRect resultsFrame = CGRectMake(10,10,300,100);
    resultsView = [[UITextView alloc] initWithFrame:resultsFrame];
    resultsView.font = [UIFont systemFontOfSize:14.0];
    [self.view addSubview:resultsView];

    NSString *symbol = @"AAPL,GOOG,MSFT,YHOO,PALM";
    NSString *urlString = [NSString stringWithFormat:@"http://finance.yahoo.com/d/quotes.csv?f=nos-s=%@", symbol];
    NSURL *url = [NSURL URLWithString:urlString];

    receivedData=[[NSMutableData alloc] initWithData:nil];

    NSURLRequest *req=[[NSURLRequest alloc] initWithURL:url];
    NSURLConnection *conn=[[NSURLConnection alloc] initWithRequest:req delegate:self];
    [req release];
    [conn release];

    NSLog(@"done");
}

- (void)dealloc {
    [resultsView release];
    [receivedData release];

    [super dealloc];
}

@end
```

Parsing XML

In the previous example, the stock quotes were returned as a comma-separated list. It's much more common, however, to have to deal with XML data when communicating with Web services or Internet-related data.

For both retrieving and parsing XML data, you can use the **NSXMLParser** class. Once an **NSXMLParser** object is initialized with a source of XML data, you simply call its `parse` method to process the data. Delegates of this class are notified as the XML is processed, and you can capture details about each XML element or attribute as the XML is parsed by implementing the corresponding delegate methods.

Next you'll look at a simple example of how you might go about using an **NSXMLParser** object to retrieve information from an RSS feed (in this case, the daily news feed from the CNN Web site), adding each parsed RSS record to an array. This example has only basic error handling, and the resulting array is simply logged to the console after the XML has been parsed. In a real-world example, you would likely save this data or display it in your application.

Each XML record of the RSS feed is in the following format:

```
<item>
    <title>Article Title</title>
    <guid>Article URL</guid>
    <pubDate>publication
        → Date</pubDate>
</item>
```

To build an application to parse an RSS feed:

1. Create a new view-based application, saving it as XMLEExample.
2. Open XMLEExampleViewController.h, add the **NSXMLParserDelegate** protocol, and create some instance variables (Code Listing 6.8):

```
NSXMLParser *parser;
NSMutableArray *articles;
NSString *currentElement;
NSMutableString *currentTitle;
NSMutableString *currentGuid;
NSMutableString *pubDate;
BOOL itemActive;
```

The **parser** variable is your XML parser object. The **articles** array will hold your parsed data, and the rest of the variables are used during parsing of the XML data.

continues on next page

Code Listing 6.8 The header file for the RSS reader.

```
#import <UIKit/UIKit.h>

@interface XMLEExampleViewController : UIViewController <NSXMLParserDelegate>
{
    NSXMLParser *parser;
    NSMutableArray *articles;

    NSString *currentElement;
    NSMutableString *currentTitle;
    NSMutableString *currentGuid;
    NSMutableString *pubDate;

    BOOL itemActive;
}

@end
```

- 3.** Switch to XMLExampleViewController.m, uncomment `viewDidLoad`, and add the following code:

```
articles = [[NSMutableArray alloc]
→ init];
```

You first create the array you will use to hold your parsed data. The array is defined as mutable since you don't know how much data you will be parsing.

```
NSURL *url = [NSURL URLWithString:
→ @"http://rss.cnn.com/rss/edition.
→ rss"];
parser = [[NSXMLParser alloc]
→ initWithContentsOfURL:url];
parser.delegate = self;
[self createUI];
```

You then create your `NSXMLParser` object, initialize it with the URL of the RSS feed, and set the delegate of the parser. Finally, you call a method to create the UI of the application. As with the earlier examples, this simply consists of a button to begin parsing the XML.

- 4.** Next you implement the parser delegate methods:

`parserDidStartDocument`: is called when the parser begins parsing.

```
[articles removeAllObjects];
```

Here you make sure the articles array is empty.

`parser:validationErrorOccurred`: and `parser:parseErrorOccurred`: will be called if the XML data is invalid or cannot be parsed at all. In this example, you show an alert dialog if either error occurs.

`parser:didStartElement:namespaceURI:
qualifiedName:attributes`: will be

called each time a new XML element is parsed.

```
currentElement =[elementName;
if ([elementName isEqualToString:
→ @"item"])
{
    itemActive = YES;
    currentTitle = [[NSMutableString
→ alloc] init];
    currentGuid = [[NSMutableString
→ alloc] init];
    pubDate = [[NSMutableString alloc]
→ init];
}
```

You first store the element name in an instance variable (to be used later) and then check to see whether the element is called **item** (indicating the start of a new RSS record). If yes, you initialize some variables that you will use to store the RSS record data. You also set a flag—**itemActive**—to indicate that an RSS record has started parsing. This flag allows you to ignore any XML data that may be parsed outside of an **<item>..</item>** block.

parser:foundCharacters: is called as the XML data is parsed:

```
if (itemActive)
{
    NSString *fixedString = [string
→ stringByTrimmingCharactersInSet:
[NCharacterSet whitespaceAnd
→ NewlineCharacterSet]];
    if ([currentElement
→ isEqualToString:@"title"])
        [currentTitle appendString:
→ fixedString];
```

5. After checking that you are currently parsing an RSS record (using the **itemActive** variable created earlier), you check the current element and append the incoming string to the relevant instance variable. Note how the string is first trimmed of any invalid whitespace and newline characters first.

6. Next you implement **parser:didEndElement:namespaceURI:qualifiedName:**, which is called when parsing reaches the end of an XML element:

```
if ([elementName isEqualToString:  
→ @"item"]){  
  
    NSDictionary *record =  
→ [NSDictionary dictionaryWith  
→ ObjectsAndKeys:  
→ currentTitle,@"articleTitle",  
→ currentGuid,@"articleURL",  
→ pubDate,@"publicationDate", nil];  
  
    [articles addObject:record];
```

Here again you check to see whether you've reached the end of the RSS record (indicated by an element named **item**), and if so, you create a dictionary of the parsed RSS record and add it to your articles array.

7. Finally, you implement **parserDidEndDocument:**, which is called when the parser finishes parsing the XML data:

```
 NSLog(@"%@",articles);
```

Here you just log the contents of the **articles** array to the console.

Code Listing 6.9 shows the completed code.

continues on page 261

Code Listing 6.9 Retrieving XML data from an RSS feed.

```
#import "XMLExampleViewController.h"

@implementation XMLExampleViewController

- (void)parser:(NSXMLParser *)parser validationErrorOccurred:(NSError *)err {
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Validation Error"
                                                       message:err.localizedDescription
                                                       delegate:nil
                                                       cancelButtonTitle:@"OK"
                                                       otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)err {
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Fatal Error"
                                                       message:err.localizedDescription
                                                       delegate:nil
                                                       cancelButtonTitle:@"OK"
                                                       otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)parserDidStartDocument:(NSXMLParser *)parser {
    [articles removeAllObjects];
}

- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
          namespaceURI:(NSString *)namespaceURI
         qualifiedName:(NSString *)qualifiedName
        attributes:(NSDictionary *)attributeDict {
    currentElement = elementName;
    if ([elementName isEqualToString:@"item"])
    {
        itemActive = YES;
        currentTitle = [[NSMutableString alloc] init];
        currentGuid = [[NSMutableString alloc] init];
        pubDate = [[NSMutableString alloc] init];
    }
}

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {
    if (itemActive)
    {
        NSString *fixedString = [string stringByTrimmingCharactersInSet:
                                 [NSCharacterSet whitespaceAndNewlineCharacterSet]];
        if ([currentElement isEqualToString:@"title"])
            [currentTitle appendString:fixedString];
        if ([currentElement isEqualToString:@"guid"])
            [currentGuid appendString:fixedString];
        if ([currentElement isEqualToString:@"pubDate"])
            [pubDate appendString:fixedString];
    }
}
```

code continues on next page

Code Listing 6.9 *continued*

```
- (void)parser:(NSXMLParser *)parser
didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName {
    if ([elementName isEqualToString:@"item"])
    {
        NSDictionary *record = [NSDictionary dictionaryWithObjectsAndKeys:
                               currentTitle,@"articleTitle",
                               currentGuid,@"articleURL",
                               pubDate,@"publicationDate",
                               nil];
        [articles addObject:record];
        [currentTitle release];
        [currentGuid release];
        [pubDate release];
        itemActive = NO;
    }
}

- (void)parserDidEndDocument:(NSXMLParser *)parser {
    NSLog(@"%@",articles);
}

- (void)retrieveXML:(id)sender {
    [parser parse];
}

- (void)createUI {

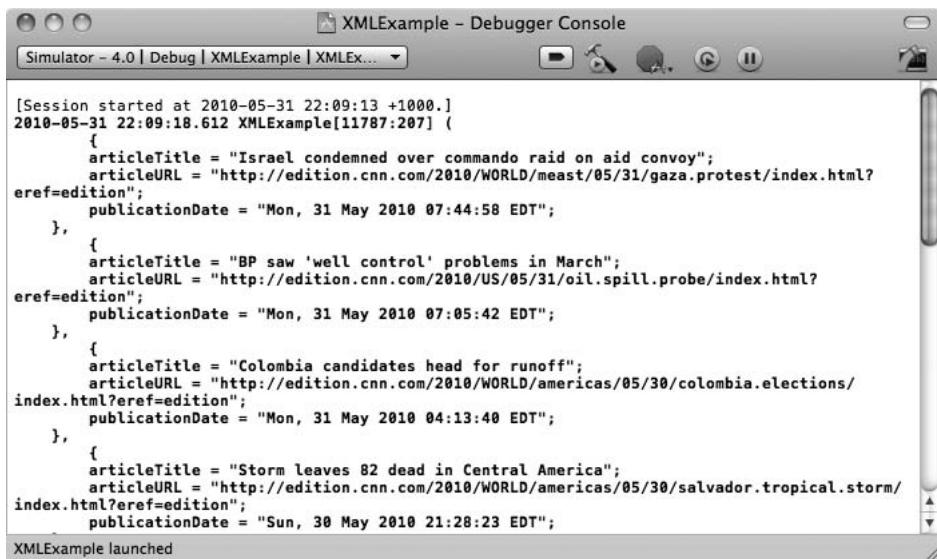
    UIButton *btnShowQuickLook = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    btnShowQuickLook.frame = CGRectMake(100,200,120,34);
    [btnShowQuickLook setTitle:@"Retrieve XML" forState:UIControlStateNormal];
    [btnShowQuickLook addTarget:self action:@selector(retrieveXML:)
        forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:btnShowQuickLook];
}

- (void)viewDidLoad {
    [super viewDidLoad];
    articles = [[NSMutableArray alloc] init];
    NSURL *url = [NSURL URLWithString:@"http://rss.cnn.com/rss/edition.rss"];
    parser = [[NSXMLParser alloc] initWithContentsOfURL:url];
    parser.delegate = self;
    [self createUI];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [articles release];
    [parser release];
    [super dealloc];
}
```

- 8.** Build and run your application. Tapping the Retrieve XML button will parse the RSS feed, and the resulting array will be logged to the console **B**.



The screenshot shows the Xcode debugger console window titled "XMLExample - Debugger Console". The title bar includes the text "Simulator - 4.0 | Debug | XMLExample | XMLEx...". The main pane displays the following log output:

```
[Session started at 2010-05-31 22:09:13 +1000.]
2010-05-31 22:09:18.612 XMLExample[11787:207] {
    {
        articleTitle = "Israel condemned over commando raid on aid convoy";
        articleURL = "http://edition.cnn.com/2010/WORLD/meast/05/31/gaza.protest/index.html?eref=edition";
        publicationDate = "Mon, 31 May 2010 07:44:58 EDT";
    },
    {
        articleTitle = "BP saw 'well control' problems in March";
        articleURL = "http://edition.cnn.com/2010/US/05/31/oil.spill.probe/index.html?eref=edition";
        publicationDate = "Mon, 31 May 2010 07:05:42 EDT";
    },
    {
        articleTitle = "Colombia candidates head for runoff";
        articleURL = "http://edition.cnn.com/2010/WORLD/americas/05/30/colombia.elections/index.html?eref=edition";
        publicationDate = "Mon, 31 May 2010 04:13:40 EDT";
    },
    {
        articleTitle = "Storm leaves 82 dead in Central America";
        articleURL = "http://edition.cnn.com/2010/WORLD/americas/05/30/salvador.tropical.storm/index.html?eref=edition";
        publicationDate = "Sun, 30 May 2010 21:28:23 EDT";
    }
}
XMLEExample launched
```

- B** Logging the contents of the articles array to the console.

Sending data to Web pages

So far, you've seen how to retrieve data (an HTTP "GET") from a Web page, using the query string to pass information to the page. You may, however, want to write an application that simulates the user completing and submitting a form on a Web page (usually done with an HTTP "POST").

In this example, you'll create an application that submits a search form to Wikipedia and displays the results in a Web view C. Note that Wikipedia could also be searched by passing the search criteria on the query string, a technique you've already learned in the "To retrieve stock quotes" exercise earlier in this chapter, so this example is for illustrative purposes only.

To create an application to search Wikipedia:

1. Create a new view-based application, saving it as PostWebContent.
2. Open PostWebContentController.h, and create some instance variables (Code Listing 6.10):

```
NSMutableData *receivedData;  
UIWebView *resultsView;  
NSString *baseURL;
```

Again, the **receivedData** variable will hold the web content, **resultsView** will be used to display the resulting HTML, and finally **baseURL** will be used to correctly resolve any links in the HTML.



C Searching Wikipedia and displaying the results.

Code Listing 6.10 The header file for the Wikipedia search application.

```
#import <UIKit/UIKit.h>  
  
@interface PostWebContentViewController : UIViewController {  
    NSMutableData *receivedData;  
    UIWebView *resultsView;  
    NSString *baseURL;  
}  
  
@end
```

- 3.** Switch to PostWebContentController.m, uncomment the `viewDidLoad` method, and add the following code:

```
CGRect resultsFrame = CGRectMake
→ (10,10,300,440);
resultsView = [[UIWebView alloc]
→ initWithFrame:resultsFrame];
[self.view addSubview:resultsView];
```

You first create a web view and add it to your main view. The web view will display the search results.

```
baseURL = @"http://en.wikipedia.
→ org";
NSString * urlString = [baseURL
→ stringByAppendingString:@"/w/
→ index.php"];
NSURL * url = [NSURL URL
→ WithString:urlString];
```

Next construct your URL just as you did before.

```
receivedData=[[NSMutableData
→ alloc] initWithData:nil];
NSMutableURLRequest * req=
→ [[NSMutableURLRequest alloc]
→ initWithURL:url cachePolicy:
→ NSURLRequestReloadIgnoringLocal
→ CacheData timeoutInterval:30.0];
```

continues on next page

Now, initialize an instance variable to hold the content as it's received. This time, however, you create an **NSMutableURLRequest** because you will be setting some properties on the request. You also tell the request to ignore any caching that may be happening and to time out after 30 seconds.

```
[req setHTTPMethod: @"POST"];
[req setHTTPBody:[@"search=iPhone"
→ dataUsingEncoding:NSUTF8StringEncoding
→ StringEncoding]];
NSURLConnection *conn =
→ [[NSURLConnection alloc]
→ initWithRequest:req
→ delegate:self];
```

Finally, you set the method of the request to be a POST, set a key-value pair to match the form file contained on the page, and then create an **NSURLConnection** just as you did earlier.

The delegates are implemented in the same way as earlier—only this time you load the resulting HTML into your web view and display the page. Notice that you also implement a fourth delegate, **connection:didReceiveResponse:**. This method is called each time a new response is received, which allows you to handle any page redirects that might happen. In this example, you reset the **receivedData** variable so that you have only the contents of the final page. **Code Listing 6.11** shows the completed code.

Code Listing 6.11 Posting content to a web page.

```
#import "PostWebContentViewController.h"

@implementation PostWebContentViewController

- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSHTTPURLResponse *)response {
    [receivedData setLength:0];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
    [receivedData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error {
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Error"
                                                    message:[error localizedDescription]
                                                    delegate:nil
                                               cancelButtonTitle:@"OK"
                                               otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    NSString *results = [[NSString alloc]
                         initWithBytes:[receivedData bytes]
                         length:[receivedData length]
                         encoding:NSUTF8StringEncoding];
    [resultsView loadHTMLString:results baseURL:[NSURL URLWithString:baseURL]];
    [results release];
}

- (void)viewDidLoad {
    CGRect resultsFrame = CGRectMake(10,10,300,440);
    resultsView = [[UIWebView alloc] initWithFrame:resultsFrame];
    [self.view addSubview:resultsView];

    baseURL = @"http://en.wikipedia.org";
    NSString *urlString = [baseURL stringByAppendingString:@"/w/index.php"];
    NSURL *url = [NSURL URLWithString:urlString];
    NSMutableURLRequest *req=[[NSMutableURLRequest alloc]
                             initWithURL:url
                             cachePolicy:NSURLRequestReloadIgnoringLocalCacheData
                             timeoutInterval:30.0];
    receivedData=[[NSMutableData alloc] initWithData:nil];
    [req setHTTPMethod: @"POST"];
    [req setHTTPBody:[@"search=iPhone" dataUsingEncoding:NSUTF8StringEncoding]];
    NSURLConnection *conn = [[NSURLConnection alloc] initWithRequest:req delegate:self];
    [req release];
    [conn release];
}

- (void)dealloc {
    [resultsView release];
    [receivedData release];
    [super dealloc];
}
```

Responding to HTTP Authentication

So far, you've seen how to get data from and send data to web pages. Next you'll learn how you deal with pages that require authentication by posting a status update to Twitter (D).

To create an application to update your status on Twitter:

1. Create a new view-based application, saving it as PostTweet.
2. Open PostTweetViewController.h, and create some instance variables (Code Listing 6.12):

```
NSMutableData *receivedData;  
UITextField *myTextField;  
UIButton *myButton;
```

Just as in previous examples, `receivedData` will hold the response from the Twitter server when you post your status update.

3. Switch to PostTweetViewController.m, uncomment the `viewDidLoad` method, and add the following code:

```
[self createUI];  
  
receivedData=[[NSMutableData  
→ alloc] initWithData:nil];
```

You first call a method to create your user interface (a text field for the status update and a button to send the update) and then create the `receivedData` object.



D Posting a status update to Twitter.

Code Listing 6.12 The header file for the Twitter application.

```
#import <UIKit/UIKit.h>  
  
@interface PostTweetViewController : UIViewController {  
  
    NSMutableData *receivedData;  
    UITextField *myTextField;  
    UIButton *myButton;  
}  
  
@end
```

4. Implement the `tweetClick:` method:

```
[receivedData setLength:0];  
NSString *twitterURL = @"http://  
→ twitter.com/statuses/update.xml";  
NSURL *url = [NSURL  
→ URLWithString:twitterURL];  
NSMutableURLRequest *req=  
→ [[NSMutableURLRequest alloc]  
→ initWithURL:url cachePolicy: NSURL  
→ RequestReloadIgnoringLocal  
→ CacheData timeoutInterval:30.0];  
[req setHTTPMethod: @"POST"];  
NSString *twitterBody=[NSString  
→ stringWithFormat:@"status=%@",  
→ myTextField.text];  
[req setHTTPBody:[twitterBody  
→ dataUsingEncoding:NSUTF8StringEncoding  
→ StringEncoding]];  
  
NSURLConnection *conn =  
→ [[NSURLConnection alloc]  
→ initWithRequest:req  
→ delegate:self];  
[req release];  
[conn release];
```

This is almost the same code as you saw in the previous exercise. After creating and populating an `NSURL` object with the URL of the Twitter update page, you create a key-value pair containing your text field's text and post it to the server.

continues on next page

5. The delegates are implemented just as previously, but this time, since Twitter requires authentication, you also need to implement the `connection:didReceiveAuthenticationChallenge:` delegate:

```
NSString *userName = @"";
NSString *userPassword = @"";
if ([chg previousFailureCount]
    → == 0)
{
    NSURLCredential *newCredential;
    newCredential=[NSURLCredential
        → credentialWithUser:userName
        → password:userPassword
        → persistence:NSURLCredential
        → PersistenceNone];
    [[chg sender] useCredential:
        → newCredentialforAuthentication
        → Challenge:chg];
}
else
    [[chg sender]
        → cancelAuthentication
        → Challenge:chg];
```

By looking at the `previousFailureCount`, you can see how many times you have attempted to authenticate and have failed. Assuming you have never attempted to authenticate, you create an `NSURLCredential` object (using your Twitter username and password) and send these using the `useCredential:forAuthenticationChallenge:` method to the connection you created earlier. [Code Listing 6.13](#) shows the completed code.

Code Listing 6.13 Posting a status update on Twitter.

```
#import "PostTweetViewController.h"

@implementation PostTweetViewController

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
    [receivedData appendData:data];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error {
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Error"
                                                    message:[error localizedDescription]
                                                    delegate:nil
                                                   cancelButtonTitle:@"OK"
                                                   otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    //log response
    NSString *result = [[NSString alloc]
                        initWithBytes:[receivedData bytes]
                        length:[receivedData length]
                        encoding:NSUTF8StringEncoding];
    NSLog(@"%@",result);
    [result release];

    //assume success
    UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Success"
                                                    message:@"Tweet Posted successfully"
                                                    delegate:nil
                                                   cancelButtonTitle:@"OK"
                                                   otherButtonTitles:nil];
    [myAlert show];
    [myAlert release];
}

- (void)connection:(NSURLConnection *)connection didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)chg {
    NSString *userName = @"";
    NSString *userPassword = @_;

    if ([chg previousFailureCount] == 0)
    {
        NSURLCredential *newCredential;
        newCredential=[NSURLCredential credentialWithUser:userName
                                                password:userPassword
                                                persistence:NSURLCredentialPersistenceNone];
        [[chg sender] useCredential:newCredential forAuthenticationChallenge:chg];
    }
    else
        [[chg sender] cancelAuthenticationChallenge:chg];
}

- (void)tweetClick:(id)sender
{
    [receivedData setLength:0];

    NSString *twitterURL = @"http://twitter.com/statuses/update.xml";
    NSURL *url = [NSURL URLWithString:twitterURL];

   NSMutableURLRequest *req=[[NSMutableURLRequest alloc]
                           initWithURL:url
                           cachePolicy:NSMutableURLRequestReloadIgnoringLocalCacheData
                           timeoutInterval:30.0];
}
```

code continues on next page

Code Listing 6.13 *continued*

```
[req setHTTPMethod:@"POST"];
NSString *twitterBody=[NSString stringWithFormat:@"status=%@",myTextField.text];
[req setHTTPBody:[twitterBody dataUsingEncoding:NSUTF8StringEncoding]];

NSURLConnection *conn=[[NSURLConnection alloc] initWithRequest:req delegate:self];
[req release];
[conn release];
}

-(void)createUI {

CGRect textRect = CGRectMake(10,10,230,32);
myTextField = [[UITextField alloc] initWithFrame:textRect];
myTextField.borderStyle = UITextBorderStyleRoundedRect;
[self.view addSubview:myTextField];

CGRect buttonRect = CGRectMake(250,10,60,32);
myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[myButton setFrame:buttonRect];
[myButton setTitle:@"Tweet" forState:UIControlStateNormal];
[myButton addTarget:self action:@selector(tweetClick:) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:myButton];
}

-(void)viewDidLoad {
    [self createUI];
    receivedData=[[NSMutableData alloc] initWithData:nil];
}

-(void)dealloc {
    [myTextField release];
    [receivedData release];
    [super dealloc];
}

@end
```

In a real-world example, you would likely store your username and password in the device keychain. Likewise, you would probably try to authenticate several times. In the previous exercise, when the `cancelAuthenticationChallenge:` method is called, the `connection:didFailWithError:` delegate can be used to handle the failed authentication.

Using a delegate for authentication works well in situations such as these. If your `NSURLConnection` request requires authentication, then the delegate method is called—otherwise, it won't be. You don't need to worry about constantly checking to see whether the authentication credentials are still valid on each request.

Notice also how in the `connectionDidFinishLoading:` delegate the `receivedData` object is logged to the console. Although this object was not strictly necessary for the example to work, you can use it to parse the Twitter server response and provide some type of meaningful feedback.

Creating peer-to-peer applications

By using the Game Kit API, you can easily create peer-to-peer applications with very little code. All the networking complexity is handled for you, letting you connect devices over Bluetooth without having to do any pairing.

The `GKSession` class is built on top of the Bonjour networking service and allows you to create both client/server and peer-to-peer connections between devices. In a peer-to-peer scenario, you are notified of the state of the session (such as when a connection is successfully made) via its delegate methods. The `GKSession` class also handles all the sending and receiving of data between peers.

For selecting which peer to communicate with, you can use the **GKPeerPickerController** class. This presents a standard UI showing you a list of the other peers **E**. It also allows you to accept or decline a peer-to-peer connection request from another iPhone **F**.

Apple also provides an API to handle voice communication; we won't cover the API here, but you can implement it without too much code. Sample code for building a peer-to-peer voice chat is available in the "Coding How-Tos" section of the *iPhone Developer Connection* Web site.

Although the Game Kit API is geared toward in-game communication and voice, you can use it for other peer-to-peer applications. In this section, you'll use the Game Kit API to see how little code is needed to create a simple peer-to-peer chat application.



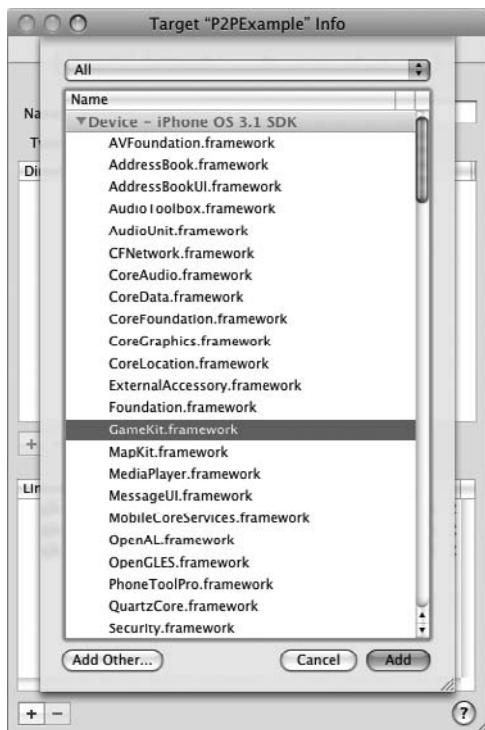
E The peer picker controller lets you browse for other peers to communicate with.



F Using the peer picker controller to accept an incoming chat request.



G The peer-to-peer chat application.



H Adding the Game Kit framework.

Unfortunately, in the current version of the iPhone development tools, the simulator cannot communicate with an iPhone over Bluetooth. This leaves the option of using either two iPhones or two computers both running the simulator. In the example presented here, the code is running in the simulator on two computers G.

To create a peer-to-peer chat application:

1. Create a new view-based application, saving it as P2PEExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add GameKit.framework H.

continues on next page

4. Open P2PExampleViewController.h, include the GameKit.h header, add the **GKPeerPickerControllerDelegate** and **GKSessionDelegate** protocol declarations, and create some instance variables (**Code Listing 6.14**):

```
GKPeerPickerController *myPicker;  
GKSession *mySession;  
UITextView *myTextView;  
UITextField *myTextField;
```

You create a peer picker controller that provides an interface to select who to chat with and a session object that will be used to connect and communicate with the other peer. You also create the objects that you'll use to create your user interface.

5. Switch to P2PExampleViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
[self createUI];  
  
myPicker = [[GKPeerPickerController alloc] init];  
  
myPicker.connectionTypesMask =  
    GKPeerPickerControllerConnectionType  
    | Nearby;  
  
myPicker.delegate = self;  
  
[myPicker show];
```

You first call a method to create the user interface (a text view for the conversation and a text field and button to send messages). Next you create the peer picker. Setting **connectionTypesMask** to **GKPeerPickerControllerConnectionTypeNearby** will make the iPhone use Bluetooth to look for other peers.

Code Listing 6.14 The header file for the peer-to-peer chat application.

```
#import <UIKit/UIKit.h>  
#import <GameKit/GameKit.h>  
  
@interface P2PExampleViewController : UIViewController  
    <GKPeerPickerControllerDelegate, GKSessionDelegate>  
  
{  
    GKPeerPickerController *myPicker;  
    GKSession *mySession;  
    UITextView *myTextView;  
    UITextField *myTextField;  
}  
  
@end
```



1 Selecting a peer from the peer picker controller.

6. Implement the peer picker delegate method, `peerPickerController:sessionForConnectionType:`, which is called when the peer picker is shown:

```
mySession = [[GKSession alloc]
→ initWithSessionID:@"p2pTest"
→ displayName:nil
sessionMode:GKSessionModePeer];
mySession.delegate = self;
return mySession;
```

This method creates and returns a new peer-to-peer session, giving it the ID p2pTest. Any other peers with the same session ID will show up in the peer picker 1.

7. You need to implement a delegate method used by your `GKSession` object. The `session:didChangeState:` method is called whenever your session changes state (in this example, when you get a connection to another peer):

```
[mySession setDataReceiveHandler:
→ self withContext:nil];
[myPicker dismiss];
[self sendChatText:@"connected"];
```

Here you tell the session to look in the current object for its receive handler. You also tell the peer picker to hide (since you now have a connection to another peer) and send an initial message to the other iPhone.

continues on next page

8. Implement the receive handler method

```
receiveData:fromPeer:inSession:  
context: that was set up in the  
previous step:  
  
NSString *receivedText =  
→ [[NSString alloc] initWithData:  
→ dataencoding:NSUTF8StringEncoding  
→ Encoding];  
  
NSString *peerName = [[NSString  
→ alloc] initWithString:[session  
→ displayNameForPeer:peer]];  
  
myTextView.text = [NSString  
→ stringWithFormat:@"%@\n%@",  
→ %@",myTextView.text,peerName,  
→ receivedText];
```

You create a string from the received data (the text sent from the other peer), get the name of the peer, and then append everything to your text view to show the conversation.

9. Now that you've written the receive method, the final step is to implement the send method (called by the Send button):

```
NSData *sendData = [newText  
→ dataUsingEncoding:NSUTF8  
→StringEncoding];  
  
[mySession sendDataToAllPeers:  
→ sendData withDataMode:  
→ GKSendDataReliable error:nil];  
  
myTextView.text = [NSString  
→ stringWithFormat:@"%@\n[ME]:  
→ %@",myTextView.text,newText];
```

This is just the reverse of step 8; here you're creating an **NSData** object from your message string and sending the data to the other peer. Again, you append this text to the text view to show the conversation. **Code Listing 6.15** shows the completed code.

Code Listing 6.15 The completed code for the peer-to-peer chat application.

```
#import "P2PEexampleViewController.h"

@implementation P2PEexampleViewController

-(void)sendChatText:(NSString *)newText {
    NSData *sendData = [newText dataUsingEncoding:NSUTF8StringEncoding];
    [mySession sendDataToAllPeers:sendData withMode:GKSendDataReliable error:nil];
    myTextView.text = [NSString stringWithFormat:@"%@\n[ME]: %@", myTextView.text,newText];
}

-(void)buttonClick:(id)sender {
    [self sendChatText:myTextField.text];
    [myTextField setText:@""];
}

-(void)createUI {
    myTextView = [[UITextView alloc] initWithFrame:CGRectMake(10,10,300,180)];
    [myTextView setEditable:NO];
    [self.view addSubview:myTextView];

    myTextField = [[UITextField alloc] initWithFrame:CGRectMake(10,200,230,30)];
    myTextField.borderStyle = UITextBorderStyleRoundedRect;
    [self.view addSubview:myTextField];

    UIButton *myButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [myButton setFrame:CGRectMake(250,200,60,32)];
    [myButton setTitle:@"Send" forState:UIControlStateNormal];
    [myButton addTarget:self action:@selector(buttonClick:) forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:myButton];
}

-(void)viewDidLoad {
    [self createUI];

    myPicker = [[GKPeerPickerController alloc] init];
    myPicker.delegate = self;
    myPicker.connectionTypesMask = GKPeerPickerControllerTypeNearby;
    //myPicker.connectionTypesMask = GKPeerPickerControllerTypeNearby | GKPeerPickerControllerTypeOnline;
    [myPicker show];
}

-(GKSession *)peerPickerController:(GKPeerPickerController *)picker sessionForConnectionType:(GKPeerPickerControllerType)connectionType {
    mySession = [[GKSession alloc] initWithSessionID:@"p2pTest" displayName:nil sessionMode:GKSessionModePeer];
    mySession.delegate = self;

    return mySession;
}

-(void)session:(GKSession *)session peer:(NSString *)peerID didChangeState:(GKPeerConnectionState)state {
    switch (state) {
        case GKPeerStateConnected:
            [mySession setDataReceiveHandler:self withContext:nil];
            [myPicker dismiss];
            [self sendChatText:@"connected"];
            break;

        case GKPeerStateDisconnected:
            break;
    }
}
```

code continues on next page

Code Listing 6.15 *continued*

```
- (void)receiveData:(NSData *)data fromPeer:(NSString *)peer inSession: (GKSession *)session context:(void *)ctx
{
    NSString *receivedText = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];
    NSString *peerName = [[NSString alloc] initWithString:[session displayNameForPeer:peer]];
    myTextView.text = [NSString stringWithFormat:@"%@\n%@", myTextView.text, peerName, receivedText];

    [peerName release];
    [receivedText release];
}

- (void)dealloc {
    [myPicker release];
    [mySession release];

    [super dealloc];
}
@end
```

7

Touches, Shakes, and Orientation

The iPhone's primary interface is its large Multi-Touch display. Since it doesn't have a physical keyboard, everything is accomplished via this screen. The iPhone takes things much further than a simple keyboard, however, allowing you to interact with your applications in a very natural and intuitive way. Objects onscreen can be moved, zoomed in or out, and scrolled using simple gestures.

The iPhone can also respond to changes in orientation; it can automatically switch and resize the display to portrait or landscape when you rotate the phone, and it can react to shakes and tilting.

In this chapter you'll see how easy it is to add both single-touch and Multi-Touch support to your applications, including responding to tapping, pinching, rotating, and zooming. You'll learn how you can respond to shake motions before looking at the iPhone's accelerometer and how to control your application UI based on changes in the phone's orientation.

In This Chapter

Touch	280
Multi-Touch Gestures	292
The iPhone Accelerometer	298

Touch

In the iPhone SDK, all **UIKit** classes that descend from **UIView** are what are known as *responder objects*—subclasses of the **UIResponder** class. Among other things, this class is responsible for providing access to and handling touch and motion events within an iPhone application.

Touch-based events are handled via four main methods:

- **touchesBegan:withEvent:**—Sent at the beginning of a touch life cycle, when the user first touches the screen of the iPhone.
- **touchesMoved:withEvent:**—Sent as the user moves their finger or fingers around on the screen of the iPhone.
- **touchesEnded:withEvent:**—Sent when the user lifts their fingers off the iPhone screen and ends the touch.
- **touchesCancelled:withEvent:**—Sent when the system receives a cancellation event. This can occur in situations such as a low-memory warning or when a phone call is received. Generally, you would use this method to perform any necessary code cleanup for objects and data generated by the other touch methods.

All four of these methods receive a **UIEvent** object containing **UITouch** objects for each finger that is interacting with the screen (or, in the case of the **touchesEnded:withEvent:** method, for each finger that has just been lifted from the screen).

The **UITouch** object allows you to determine not only *what* is being touched but *when* and *where* it was touched. In the case of movement, it also lets you know where the touched object was and where it was moved.

Code Listing 7.1 The header file of the touch-based application.

```
//  
// TouchExampleViewController.h  
// TouchExample  
//  
  
#import <UIKit/UIKit.h>  
  
@interface TouchExampleViewController : UIViewController  
{  
    UIView *redBox;  
}  
  
@end
```

To create a touch-based application:

1. Create a new view-based application, saving it as TouchExample.
2. Open the TouchExampleViewController.h file, and create an instance variable to hold the view you will be controlling by touch (**Code Listing 7.1**).
3. Switch to the TouchExampleView Controller.m file, uncomment the **viewDidLoad** method, and add the following code:

```
float boxSize = 100.0;  
  
CGRect redBoxRect = CGRectMake  
→ (110,180,boxSize,boxSize);  
  
redBox = [[UIView alloc]  
→ initWithFrame:redBoxRect];  
  
redBox.backgroundColor =  
→ [UIColor redColor];  
  
[self.view addSubview:redBox];
```

Here you are simply creating a **UIView** with a red background, setting its dimensions, and adding it to the main.

4. Implement **touchesMoved:withEvent:** to handle the touch movement:

```
UITouch *touch = [[event touches  
→ ForView:redBox] anyObject];  
  
CGPoint currentPoint = [touch  
→ locationInView:self.view];
```

You first retrieve the touch events for the red box view, and then you calculate its location (in relation to its containing view).

```
[touch view].center =  
→ currentPoint;
```

continues on next page

Then you set the center of the red box view to the point being touched. **Code Listing 7.2** shows the completed application code.

5. Build and run the application.

You should be able to move the red box by touching the iPhone's screen.

Code Listing 7.2 The completed touch-based application.

```
#import "TouchExampleViewController.h"

@implementation TouchExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    float boxSize = 100.0;
    CGRect redBoxRect = CGRectMake(110,180,boxSize,boxSize);
    redBox = [[UIView alloc] initWithFrame:redBoxRect];
    redBox.backgroundColor = [UIColor redColor];

    [self.view addSubview:redBox];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event touchesForView:redBox] anyObject];
    CGPoint currentPoint = [touch locationInView:self.view];

    if (touch.view != self.view)
        [touch view].center = currentPoint;
}

- (void)dealloc {
    [redBox release];
    redBox = nil;

    [super dealloc];
}

@end
```

Code Listing 7.3 The updated header file for the touch-based application.

```
//  
// TouchExampleViewController.h  
// TouchExample  
//  
  
#import <UIKit/UIKit.h>  
  
@interface TouchExampleViewController : UIViewController  
{  
    UIView *redBox;  
    UIView *blueBox;  
    UIView *greenBox;  
}  
  
@end
```

This simple exercise shows how easy it is to add touch support to applications. Now you'll extend the example to deal with multiple views onscreen, allowing them to be moved around independently of each other. You'll also add a simple zoom animation effect to show which view you are currently touching.

To update the touch-based application:

1. Open TouchExampleViewController.h, and create two more instance variables (**Code Listing 7.3**).
2. Modify the **viewDidLoad** method to create three different-colored boxes, and add them to the main view.
3. Implement the **touchesBegan:withEvent:** method:

```
UITouch *touch = [touches  
→ anyObject];  
  
if (touch.view != self.view)  
{...
```

This time, since you have multiple views being touched, you need to check that you are not receiving a touch event from the main, containing view:

```
[self.view bringSubviewToFront:  
→ touch.view];
```

You then make sure the view you are touching moves to the front of the screen:

```
float zoom = -25.0;  
  
CGRect newRect = CGRectMakeInset  
→ ([touch.view frame], zoom, zoom);  
  
[UIView beginAnimations:nil  
→ context:NULL];  
  
[UIView setAnimationDuration:0.2];  
[touch.view setFrame:newRect];  
[UIView commitAnimations];
```

continues on next page

Next, you create a **CGRect** that has a frame 25 pixels larger than the touched view and tell the touched view to resize itself. Doing this inside a **beginAnimations...commitAnimations** block makes the resize animate nicely.

- Finally, you implement the **touchesEnded: withEvent:** method, which is essentially the reverse of the previous step.

Notice that this time you specify a positive value for the **CGRectInset** function, creating a smaller **CGRect**. **Code Listing 7.4** shows the updated code.

Code Listing 7.4 The updated touch-based application.

```
#import "TouchExampleViewController.h"

@implementation TouchExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    float boxSize = 100.0;
    CGRect redBoxRect = CGRectMake(0,180,boxSize,boxSize);
    redBox = [[UIView alloc] initWithFrame:redBoxRect];
    redBox.backgroundColor = [UIColor redColor];

    CGRect blueBoxRect = CGRectMake(110,180,boxSize,boxSize);
    blueBox = [[UIView alloc] initWithFrame:blueBoxRect];
    blueBox.backgroundColor = [UIColor blueColor];

    CGRect greenBoxRect = CGRectMake(220,180,boxSize,boxSize);
    greenBox = [[UIView alloc] initWithFrame:greenBoxRect];
    greenBox.backgroundColor = [UIColor greenColor];

    [self.view addSubview:redBox];
    [self.view addSubview:blueBox];
    [self.view addSubview:greenBox];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    CGPoint currentPoint = [touch locationInView:self.view];

    if (touch.view != self.view)
        [touch.view].center = currentPoint;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];

    if (touch.view != self.view) {
        [self.view bringSubviewToFront:touch.view];
        [self.view beginAnimations:@"Resizing", context:nil];
        [self.view setFrame:CGRectMake(0, 180, 100, 100)];
        [self.view setCenter:CGPointMake(110, 180)];
        [self.view endAnimations];
    }
}
```

code continues on next page



A Moving multiple views by touch.

5. Build and run your application.

Touching a colored box should cause it to come to the front and animate to a larger size A. You can move the box around the screen, and when you let go, it will animate back down to its original size.

Adding tapping support

The **tapCount** property of a **touch** object is the key to adding tap and long-touch support to your applications.

If you revisit the **touchesBegan:withEvent:** method of the previous example and add the following line:

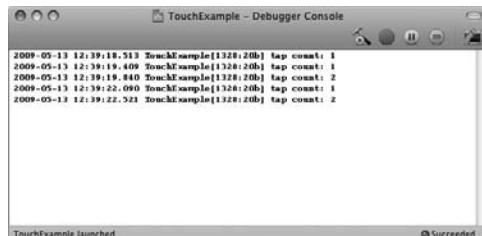
```
NSLog(@"tap count: %i", [touch  
→ tapCount]);
```

Code Listing 7.4 continued

```
if (touch.view != self.view) {  
    [self.view bringSubviewToFront:touch.view];  
  
    float zoom = -25.0;  
    CGRect newRect = CGRectMakeInset([touch.view frame], zoom, zoom);  
  
    [UIView beginAnimations:nil context:NULL];  
    [UIView setAnimationDuration:0.2];  
    [touch.view setFrame:newRect];  
    [UIView commitAnimations];  
}  
}  
  
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {  
  
    UITouch *touch = [[event allTouches] anyObject];  
    if (touch.view != self.view) {  
        float zoom = 25.0;  
        CGRect newRect = CGRectMakeInset([touch.view frame], zoom, zoom);  
  
        [UIView beginAnimations:nil context:NULL];  
        [UIView setAnimationDuration:0.2];  
        [touch.view setFrame:newRect];  
        [UIView commitAnimations];  
    }  
}  
  
- (void)dealloc {  
  
    [redBox release];  
    [blueBox release];  
    [greenBox release];  
  
    [super dealloc];  
}  
@end
```

you can see the tap counts being logged to the console **B**. There's a problem here, however: Along with double-tap events, you still see events for the single-tap. You need to figure out a way of telling these two apart and ignoring the single-tap events when you double-tap.

One simple way to do this is to add a small delay to when you actually deal with the tap events, canceling earlier tap events when later tap events occur. This sounds confusing, but it's actually very easy to do.



B Logging tap counts to the Debugger Console.

Other uses for CGRect

In the previous exercise “To update the touch-based application,” you use `CGRectInset` to increase or decrease the size of the `CGRect`. There are many other useful functions that make working with `CGRects` easier; among them are the following:

`CGRectOffset` creates a rectangle with the origin offset to a source `CGRect`:

```
float offset = 25.0;
CGRect r1 = CGRectMake(100,100,100,100);
CGRect r2 = CGRectOffset(r1,offset,offset);
```

`CGRectIntersectsRect` lets you determine whether two rectangles intersect each other:

```
CGRect r1 = CGRectMake(100,100,100,100);
CGRect r2 = CGRectMake(150,150,100,100);
if (CGRectIntersectsRect(r1,r2)
    NSLog(@"intersecting");
```

`NSStringFromCGRect` is useful for logging `CGRects` to the console:

```
CGRect r1 = CGRectMake(100,100,100,100);
NSLog(@"rect: %@", NSStringFromCGRect(r1));
```

Likewise, `CGRectFromString` allows you to create a `CGRect` from a string:

```
NSString *r = @"{0,0},{100,100}";
CGRect r1 = CGRectFromString(r);
```

See the `CGGeometry` reference entry in the Apple developer documentation for more details.

To add single-tap and double-tap support:

1. Open TouchExampleViewController.m, and modify the **touchesBegan:withEvent:** method (Code Listing 7.5):

```
float tapDelay = 0.4;
```

This is the amount of time you wait before processing a tap event, which gives you time to check whether it's a single-tap or a double-tap:

```
switch ([touch tapCount])
```

continues on next page

Code Listing 7.5 Handling double-taps.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    if (touch.view != self.view)
    {
        float tapDelay = 0.4f;
        switch ([touch tapCount])
        {
            case 1:
                [self performSelector:@selector(singleTap:) withObject:touch.view afterDelay:tapDelay];
                break;
            case 2:
                [NSObject cancelPreviousPerformRequestsWithTarget:self selector:@selector(singleTap:) object:touch.view];
                [self doubleTap:touch.view];
                break;
        }
        [self.view bringSubviewToFront:touch.view];
        float zoom = -25.0;
        CGRect newRect = CGRectMakeInset([touch.view frame], zoom,zoom);
        [UIView beginAnimations:nil context:NULL];
        [UIView setAnimationDuration:0.2];
        [touch.view setFrame:newRect];
        [UIView commitAnimations];
    }
}
```

2. Next you check to see how many taps you have received:

case 1:

```
[self performSelector:@selector  
→ (singleTap) withObject:touch.view  
→ afterDelay:tapDelay];
```

If you've received a single tap, you make a delayed call to the `singleTap` method:

case 2:

```
[NSObject cancelPreviousPerform  
→ RequestsWithTarget:  
→ self selector:@selector  
→ (singleTap) object:touch.view];  
  
[self doubleTap:touch.view];
```

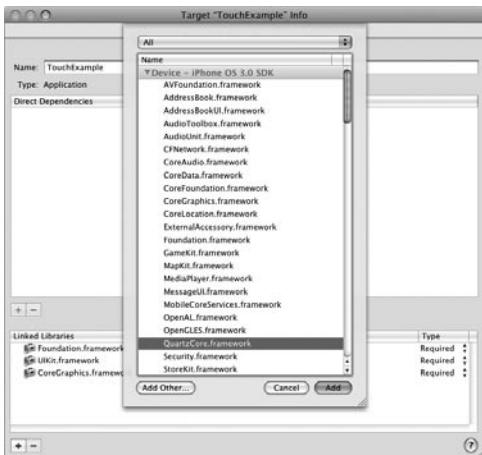
If you get a double-tap, you first cancel the call to the `singleTap` method before calling `doubleTap`.

TIP Notice how this code could easily be extended to add support for triple or even quadruple taps; however, you may have to increase the value for `tapDelay` to give the user time to perform more than a double-click.

Adding long-touch support

One other useful touch-based interaction is the touch-and-hold, or *long-touch*, effect—as seen on the iPhone home screen when you move applications around or delete them.

Just as in the double-tap example, you make a delayed call to the long-touch method in the `touchesBegan:withEvent:` method; however, this time, since you are effectively dealing with a single-tap only, you cancel the call in the `touchesEnded:withEvent:` method instead.



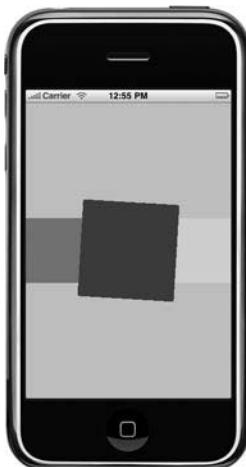
C Adding the QuartzCore framework to the project.

Code Listing 7.6 The updated header file.

```
//
// TouchExampleViewController.h
// TouchExample
//

#import <UIKit/UIKit.h>
#import <QuartzCore/QuartzCore.h>

@interface TouchExampleViewController : UIViewController
{
    UIView *redBox;
    UIView *blueBox;
    UIView *greenBox;
}
@end
```



D The simulator showing the animated view.

For bonus points, you'll now make the view "wiggle" just like on the iPhone home screen.

To add long-touch support:

- In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
- Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add QuartzCore.framework **C**.
- Open the TouchesExampleViewController.h file, and import the QuartzCore framework (**Code Listing 7.6**).
- Back in TouchesExampleViewController.m, modify the **touchesBegan:withEvent:** method:


```
float tapDelay = 1.0f;
[self performSelector:@selector
  → (startWiggle:) withObject:touch.
  → view afterDelay:tapDelay];
```

 Here you set the tap delay to be a slightly larger value than earlier (one second) before calling the **startWiggle:** method, passing the view you want to animate.
- In the **touchesEnded:withEvent:** method, you make sure to cancel any pending touches, and then you tell the touched view to stop animating.
- Finally, you implement the **startWiggle:** and **stopWiggle:** methods that cause the view to animate.
- Code Listing 7.7** shows the completed code.

- 7.** Build and run the application.

If you long-touch one of the views, it should start animating **D**.

Code Listing 7.7 The completed touch-based application.

```
#import "TouchExampleViewController.h"

@implementation TouchExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    float boxSize = 100.0;
    CGRect redBoxRect = CGRectMake(0,180,boxSize,boxSize);
    redBox = [[UIView alloc] initWithFrame:redBoxRect];
    redBox.backgroundColor = [UIColor redColor];

    CGRect blueBoxRect = CGRectMake(110,180,boxSize,boxSize);
    blueBox = [[UIView alloc] initWithFrame:blueBoxRect];
    blueBox.backgroundColor = [UIColor blueColor];

    CGRect greenBoxRect = CGRectMake(220,180,boxSize,boxSize);
    greenBox = [[UIView alloc] initWithFrame:greenBoxRect];
    greenBox.backgroundColor = [UIColor greenColor];

    [self.view addSubview:redBox];
    [self.view addSubview:blueBox];
    [self.view addSubview:greenBox];
}

- (void)startWiggle:(UIView *)theView {
    CALayer *viewLayer = [theView layer];
    CABasicAnimation *anim = [CABasicAnimation animationWithKeyPath:@"transform"];
    anim.duration = 0.1;
    anim.repeatCount = 1e100f;
    anim.autoreverses = YES;
    anim.fromValue = [NSValue valueWithCATransform3D:CATransform3DRotate(viewLayer.transform, -0.1,0,0,0,0,0.5)];
    anim.toValue = [NSValue valueWithCATransform3D:CATransform3DRotate(viewLayer.transform, 0.1,0,0,0,0,0.5)];
    [viewLayer addAnimation:anim forKey:@"wiggle"];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    CGPoint currentPoint = [touch locationInView:self.view];

    if (touch.view != self.view)
        [touch view].center = currentPoint;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    if (touch.view != self.view)
    {
        //long-tap support
        float tapDelay = 1.0f;
        [self performSelector:@selector(startWiggle:)
                      withObject:touch.view
                        afterDelay:tapDelay];
        [self.view bringSubviewToFront:touch.view];
        float zoom = -25.0;
        CGRect newRect = CGRectMakeInset([touch.view frame], zoom,zoom);
    }
}
```

code continues on next page

Code Listing 7.7 *continued*

```
[UIView beginAnimations:nil context:NULL];
[UIView setAnimationDuration:0.2];
[touch.view setFrame:newRect];
[UIView commitAnimations];
}
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [[event allTouches] anyObject];
    if (touch.view != self.view)
    {
        //cancel any pending long-touches
        [NSObject cancelPreviousPerformRequestsWithTarget:self
                                                    selector:@selector(startWiggle:)
                                                    object:touch.view];

        //make sure we are no longer wiggling
        [self stopWiggle:touch.view];

        [self.view bringSubviewToFront:touch.view];

        float zoom = 25.0;
        CGRect newRect = CGRectMakeInset([touch.view frame], zoom,zoom);

        [UIView beginAnimations:nil context:NULL];
        [UIView setAnimationDuration:0.2];
        [touch.view setFrame:newRect];
        [UIView commitAnimations];
    }
}

- (void)dealloc {
    [redBox release];
    [blueBox release];
    [greenBox release];
    [super dealloc];
}
@end
```

Multi-Touch Gestures

So far you've looked at dealing with single-touch events only. However, the real power of the iPhone becomes apparent once you start thinking about Multi-Touch events.

Features such as pinching and zooming can be accomplished only when you use multiple fingers to interact with the display of the iPhone.

You'll now create an application similar to the previous example, but this time you'll have it support Multi-Touch events.

To create an application that supports Multi-Touch gestures:

1. Create a new view-based application, and save it as MultiTouchExample.
2. Open the MultiTouchExampleView Controller.h file, and create an instance variable to hold the Multi-Touch-compliant view ([Code Listing 7.8](#)).
3. Switch to the MultiTouchExample ViewController.m file, uncomment the `viewDidLoad` method, and add the following code:

```
float boxSize = 100.0;  
CGRect redBoxRect = CGRectMake  
→ (110,180,boxSize,boxSize);  
  
redBox = [[UIView alloc]  
→ initWithFrame:redBoxRect];  
  
redBox.backgroundColor =  
→ [UIColor redColor];  
  
redBox.mutipleTouchEnabled = YES;  
  
[self.view addSubview:redBox];
```

This is just the same code you used in the touch-based application except that this time you've enabled Multi-Touch by setting the `multipleTouchEnabled` property of the view to `YES`.

- 4.** Next, back in the **touchesBegan:withEvent:** method, you retrieve the set of touches for the red box view:

```
for (UITouch *touch in [event  
→ touchesForView:redBox])
```

Then you log the location **CGPoint** of the touch event to the console. **Code Listing 7.9** show the completed code.

continues on next page

Code Listing 7.8 The header file for the Multi-Touch application.

```
//  
// MultiTouchExampleViewController.h  
// MultiTouchExample  
//  
#import <UIKit/UIKit.h>  
  
@interface MultiTouchExampleViewController : UIViewController  
{  
    UIView *redBox;  
}  
  
@end
```

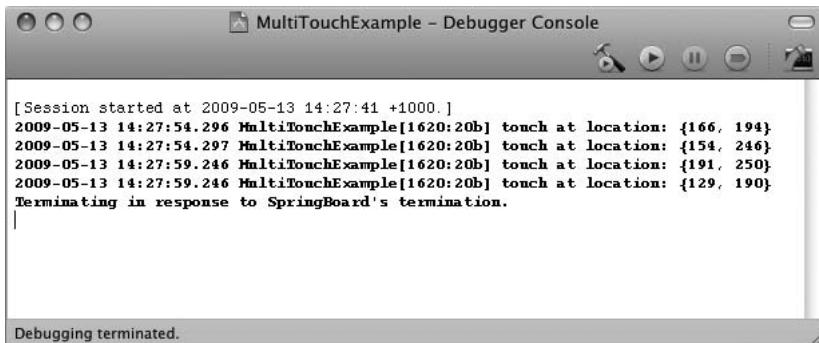
Code Listing 7.9 Logging multiple touches to the console.

```
#import "MultiTouchExampleViewController.h"  
  
@implementation MultiTouchExampleViewController  
  
- (void)viewDidLoad {  
  
    float boxSize = 100.0;  
    CGRect redBoxRect = CGRectMake(110,180,boxSize,boxSize);  
    redBox = [[UIView alloc] initWithFrame:redBoxRect];  
    redBox.backgroundColor = [UIColor redColor];  
    redBox.multipleTouchEnabled = YES;  
  
    [self.view addSubview:redBox];  
}  
  
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {  
  
    for (UITouch *touch in [event touchesForView:redBox])  
        NSLog(@"touch at location: %@",  
              NSStringFromCGPoint([touch locationInView:self.view]));  
}  
  
- (void)dealloc {  
    [super dealloc];  
}  
  
@end
```

- Build and run the application, and try touching the red box view with multiple fingers.

You should see multiple touch events being logged in the console **A**. If you are using the simulator, you can simulate multiple touches by holding down the Option key. The simulator is limited to only two simultaneous touches.

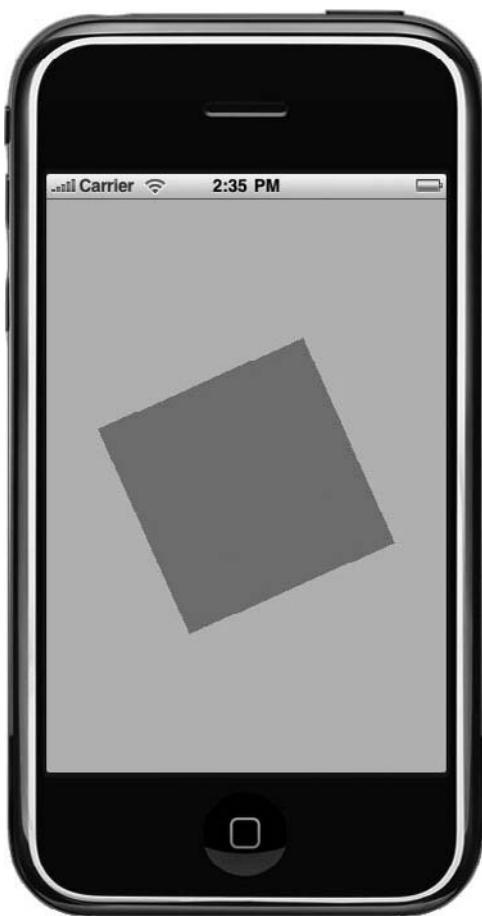
Having multiple touches is what gives the iPhone its pinch, rotate, and zoom gestures. You'll now update the application to add this functionality.



The screenshot shows the Xcode Debugger Console window titled "MultiTouchExample - Debugger Console". The log output displays several touch events:

```
[Session started at 2009-05-13 14:27:41 +1000.]  
2009-05-13 14:27:54.296 MultiTouchExample[1620:20b] touch at location: {166, 194}  
2009-05-13 14:27:54.297 MultiTouchExample[1620:20b] touch at location: {154, 246}  
2009-05-13 14:27:59.246 MultiTouchExample[1620:20b] touch at location: {191, 250}  
2009-05-13 14:27:59.246 MultiTouchExample[1620:20b] touch at location: {129, 190}  
Terminating in response to SpringBoard's termination.  
|  
Debugging terminated.
```

- A** Logging touch locations to the Debugger Console.



B Rotating the view.

To add pinch, rotate, and zoom gestures:

1. Open MultiTouchExampleView Controller.m, and modify `touchesMoved:withEvent:`

```
if ([[event touchesForView:redBox]
→ count] == 1)
{
    UITouch *touch = [[[event
→ touchesForView:redBox]
→ allObjects] objectAtIndex:0];
    redBox.transform = CGAffineTransformRotate(redBox.transform,
→ touch,redBox,self.view);
}
```

If there is only a single-touch, you call a function to rotate the view.

```
if ([[event touchesForView:redBox]
→ count] == 2)
{
    UITouch *touch1 = [[[event
→ touchesForView:redBox]
→ allObjects] objectAtIndex:0];
    UITouch *touch2 = [[[event
→ touchesForView:redBox]
→ allObjects] objectAtIndex:1];
    redBox.transform = CGAffineTransformScale(redBox.transform,
→ touch1, touch2);
}
```

If there are two touches, you call a function to scale the view. **Code Listing 7.10** shows the completed code.

2. Build and run the application.

To rotate the red box, use a single finger. To zoom in and out, use two fingers and a zoom or pinch gesture. Now check out the application after a zoom and rotation have taken place **B**.

Code Listing 7.10 The updated code for the Multi-Touch application. You can rotate the view by touching with a single finger and scale by touching with two fingers.

```
#import "MultiTouchExampleViewController.h"

@implementation MultiTouchExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    float boxSize = 100.0;
    CGRect redBoxRect = CGRectMake(110,180,boxSize,boxSize);
    redBox = [[UIView alloc] initWithFrame:redBoxRect];
    redBox.backgroundColor = [UIColor redColor];
    redBox.multipleTouchEnabled = YES;

    [self.view addSubview:redBox];
}

CGFloat distanceBetweenPoints(CGPoint pt1, CGPoint pt2) {
    CGFloat distance;

    CGFloat xDifferenceSquared = pow(pt1.x - pt2.x, 2);
    CGFloat yDifferenceSquared = pow(pt1.y - pt2.y, 2);
    distance = sqrt(xDifferenceSquared + yDifferenceSquared);

    return distance;
}

CGAffineTransform transformWithScale(CGAffineTransform oldTransform,
                                    UITouch *touch1,
                                    UITouch *touch2) {

    CGPoint touch1Location = [touch1 locationInView:nil];
    CGPoint touch1PreviousLocation = [touch1 previousLocationInView:nil];
    CGPoint touch2Location = [touch2 locationInView:nil];
    CGPoint touch2PreviousLocation = [touch2 previousLocationInView:nil];

    // Get distance between points
    CGFloat distance = distanceBetweenPoints(touch1Location,
                                              touch2Location);

    CGFloat prevDistance = distanceBetweenPoints(touch1PreviousLocation,
                                                touch2PreviousLocation);

    // Figure new scale
    CGFloat scaleRatio = distance / prevDistance;

    CGAffineTransform newTransform = CGAffineTransformScale(oldTransform, scaleRatio, scaleRatio);

    // Return result
    return newTransform;
}
}
```

code continues on next page

Code Listing 7.10 *continued*

```
CGAffineTransform transformWithRotation(CGAffineTransform oldTransform,
                                      UITouch *touch,
                                      UIView *view,
                                      id superview) {
    CGPoint pt1 = [touch locationInView:superview];
    CGPoint pt2 = [touch previousLocationInView:superview];
    CGPoint center = view.center;
    CGFloat angle1 = atan2( center.y - pt2.y, center.x - pt2.x );
    CGFloat angle2 = atan2( center.y - pt1.y, center.x - pt1.x );

    CGAffineTransform newTransform = CGAffineTransformRotate(oldTransform, angle2-angle1);

    // Return result
    return newTransform;
}

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    if ([[event touchesForView:redBox] count] == 1)
    {
        UITouch *touch = [[[event touchesForView:redBox] allObjects] objectAtIndex:0];
        redBox.transform = transformWithRotation(redBox.transform,touch,redBox,self.view);
    }

    if ([[event touchesForView:redBox] count] == 2)
    {
        UITouch *touch1 = [[[event touchesForView:redBox] allObjects] objectAtIndex:0];
        UITouch *touch2 = [[[event touchesForView:redBox] allObjects] objectAtIndex:1];
        redBox.transform = transformWithScale(redBox.transform, touch1, touch2);
    }
}

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in [event touchesForView:redBox])
        NSLog(@"%@",NSStringFromCGPoint([touch locationInView:self.view]));
}

-(void)dealloc {
    [redBox release];
    [super dealloc];
}

@end
```

The iPhone Accelerometer

The iPhone's accelerometer allows it to automatically detect movement of the phone such as tilting or shaking. The iPhone knows when the phone is rotated between portrait mode and landscape mode, and it's even able to tell whether it is face-up or face-down.

Having access to this information enables you to provide a very rich user experience that can dynamically change. For example, the display can automatically adjust when the phone is rotated, and characters in games can be controlled by simply tilting the phone.

You'll now learn how you go about using the accelerometer in your own applications.

Detecting shakes

Motion events are similar to the touch events described earlier but are much simpler to deal with. An event is generated only when a motion *starts* or *stops*—you can't track individual motions as you can with touch events.

Handling a shake motion is accomplished via three methods:

- **`motionBegan:withEvent:`**—Called when a motion event begins.
- **`motionEnded:withEvent:`**—Called when a motion event ends.
- **`motionCancelled:withEvent:`**—Called if the system thinks the motion is not a shake. Shakes are determined to be approximately a second or so in length.

To create an application that supports shakes:

1. Create a new view-based application, saving it as ShakeExample.
2. Open the ShakeExampleViewController.m file, uncomment the `viewDidAppear:` method, and add the following code:

```
[self becomeFirstResponder];
```

For the view controller to receive motion events, it needs to be the *first responder*—the object at the start of the chain of **UIResponder** subclasses. You also need to implement one more method to make the view controller the first responder:

```
- (BOOL)canBecomeFirstResponder {  
    return YES;  
}
```

3. Implement the three methods responsible for handling motion events. **Code Listing 7.11** shows the completed code.
4. Build and run the application.

If you shake your iPhone, you should see messages being logged to the console. If you are running the application in the simulator, you can generate a shake by pressing Shift+Command+Z.

Determining orientation

To determine which way the iPhone is facing, you use the **UIDevice** singleton and its **orientation** property. If you register for the **UIDeviceOrientationDidChangeNotification** notification, you are told not only when the phone is rotated between portrait and landscape modes but also when it's face-up or face-down.

Code Listing 7.11 The completed code for the shake application.

```
#import "ShakeExampleViewController.h"

@implementation ShakeExampleViewController

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

-(BOOL)canBecomeFirstResponder {
    return YES;
}

-(void)motionBegan:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    if (event.type == UIEventTypeMotion && event.subtype == UIEventSubtypeMotionShake)
        NSLog(@"shake began");
}

-(void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    if (event.type == UIEventTypeMotion && event.subtype == UIEventSubtypeMotionShake)
        NSLog(@"shake ended");
}

-(void)motionCancelled:(UIEventSubtype)motion withEvent:(UIEvent *)event {
    NSLog(@"shake cancelled");
}

-(void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc that aren't in use.
}

-(void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

-(void)dealloc {
    [super dealloc];
}

@end
```

You'll now look at some code to detect a change in orientation. In this simple example, you'll change the background color when the phone orientation changes.

To create an application that detects orientation:

1. Create a new view-based application, and save it as OrientationExample.
2. Open OrientationExampleView Controller.m, and uncomment the **viewDidLoad** method. Add the following code:

```
[[NSNotificationCenter
→ defaultCenter] addObserver:
→ self selector:@selector
→ (noteOrientationChanged:) name:
→ UIDeviceOrientationDidChange
→ Notification object:nil];
[[UIDevice currentDevice]
→ beginGeneratingDevice
→ OrientationNotifications];
```

You first add yourself as an observer to the **UIDeviceOrientationDidChange** **Notification** notification, which is sent every time the orientation changes. You then tell the iPhone to start generating notifications for orientation changes.

3. Finally, you need to write the code to change the background whenever you receive notification of an orientation change.

Code Listing 7.12 shows the completed code.

4. Build and run the application.

Try rotating your iPhone to see the background change color. If you are running the code in the simulator, you can rotate by pressing Command+left arrow or Command+right arrow.

Code Listing 7.12 Changing the background color in response to a change in orientation.

```
#import "OrientationExampleViewController.h"

@implementation OrientationExampleViewController

-(void)setBackgroundForOrientation:(UIInterfaceOrientation)orientation
{
    switch (orientation) {
        case UIDeviceOrientationPortrait:
            [self.view setBackgroundColor:[UIColor brownColor]];
            break;

        case UIDeviceOrientationLandscapeLeft:
            [self.view setBackgroundColor:[UIColor greenColor]];
            break;

        case UIDeviceOrientationLandscapeRight:
            [self.view setBackgroundColor:[UIColor blueColor]];
            break;

        case UIDeviceOrientationFaceUp:
            [self.view setBackgroundColor:[UIColor yellowColor]];
            break;

        case UIDeviceOrientationFaceDown:
            [self.view setBackgroundColor:[UIColor blackColor]];
            break;
    }
}

-(void)noteOrientationChanged:(NSNotification *)aNotification {
    [self setBackgroundForOrientation:[UIDevice currentDevice].orientation];
}

-(void)viewDidLoad {
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(noteOrientationChanged:)
                                               name:UIDeviceOrientationDidChangeNotification
                                              object:nil];
    [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
}

-(void)dealloc {
    [[UIDevice currentDevice] endGeneratingDeviceOrientationNotifications];
    [super dealloc];
}

@end
```

Redrawing the interface when the orientation changes

When a user rotates their iPhone between landscape and portrait modes, you are likely to want the application interface to redraw itself. For example, the user might be browsing the Web, and a wider screen would be a more desirable way to view additional content rather than scrolling from side to side.

One way to accomplish this might be to use the `UIDeviceOrientationDidChangeNotification` notification and orientation property of `UIDevice` that you just looked at, manually redrawing the interface each time you detect a change in orientation.

The iPhone SDK offers a much more elegant solution, however: *autorotation*.

Autorotation is handled in your view controller via the method `shouldAutorotateToInterfaceOrientation`. This method returns a Boolean value indicating whether the current orientation should be autorotated.

To force an application to stay in portrait mode (the default), simply return `NO` from this method. If you want to support all orientations, you can simply return `YES`. Otherwise, you can just inspect the `interfaceOrientation` parameter and determine whether you want to autorotate.

You'll now update the application to use autorotation. You'll draw a box in the center of the screen with some text on it that will automatically rotate when you change the iPhone's orientation. The box will maintain its position in the center of the screen after rotation.

To update the application to use autorotation:

1. Open OrientationExampleView Controller.h, and create an instance variable to hold the red box view (**Code Listing 7.13**).
2. Back in OrientationExampleView Controller.m, remove all of the code to change the background of the screen. You can also remove the call to **beginGenerating DeviceOrientation Notifications**. Modify your **viewDidLoad** method to create a box that you will be rotating:

```
float boxSize = 100.0;

redTextBox = [[UITextView alloc]
→ initWithFrame:CGRectMake(110,180,
→ boxSize,boxSize)];

redTextBox.backgroundColor =
→ [UIColor redColor];

redTextBox.textColor = [UIColor
→ whiteColor];

redTextBox.text = @"Hello World";

redTextBox.textAlignment =
→ NSTextAlignmentCenter;
```

Nothing special is going on here—you are just creating a **UITextView** and setting its size, text, alignment, background, and text color properties.

Code Listing 7.13 The header file of the autorotation application.

```
#import <UIKit/UIKit.h>

@interface OrientationExampleViewController : UIViewController {

    UITextView *redTextBox;
}

@end
```

The next step is important, however:

```
redTextBox.autoresizingMask =
→ (UIViewAutoresizingFlexible
→ RightMargin
→ |
→ UIViewAutoresizing
→ FlexibleLeftMargin
→ |
→ UIViewAutoresizingFlexible
→ TopMargin
→ |
→ UIViewAutoresizing
→ FlexibleBottomMargin);
```

When switching between portrait and landscape modes, you want the red box view to keep its position relative to the new orientation. You accomplish this by setting all of its margins to resize automatically—in effect keeping the view in the center of the screen.

3. Finally, uncomment the `shouldAutorotate` `ToInterfaceOrientation`: method to enable autorotation.

Since you want to support all orientations, you simply return YES from this method. **Code Listing 7.14** shows the updated code.

4. Build and run the application.

Try changing the iPhone between portrait and landscape modes—the red box view will automatically rotate, remaining in the center of the screen. Try removing or commenting out the code that sets the `autoresizingMask` to see the effect on rotating: Although the red box view will still have the same x and y coordinates, the orientation change also changes the width and height of the screen, resulting in the box no longer being drawn at the center of the screen.

Code Listing 7.14 The updated autorotation code. The box in the center of the screen will automatically rotate when the iPhone is switched between portrait and landscape modes.

```
#import "OrientationExampleViewController.h"

@implementation OrientationExampleViewController

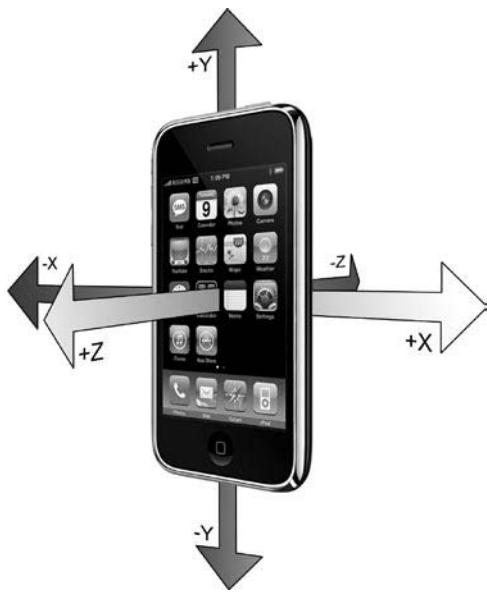
- (void)viewDidLoad {
    float boxSize = 100.0;
    redTextBox = [[UITextView alloc] initWithFrame:CGRectMake(110,180,boxSize,boxSize)];
    redTextBox.backgroundColor = [UIColor redColor];
    redTextBox.textColor = [UIColor whiteColor];
    redTextBox.text = @"Hello World!";
    redTextBox.textAlignment = UITextAlignmentCenter;
    redTextBox.autoresizingMask = (UIViewAutoresizingFlexibleRightMargin
        |
        UIViewAutoresizingFlexibleLeftMargin
        |
        UIViewAutoresizingFlexibleTopMargin
        |
        UIViewAutoresizingFlexibleBottomMargin);

    [self.view addSubview:redTextBox];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    return YES;
}

- (void)dealloc {
    [redTextBox release];
    [super dealloc];
}

@end
```



A The iPhone accelerometer can respond to changes on three axes.

Responding to the accelerometer

So far you've looked at how to detect and make your interfaces orient themselves automatically when the user changes their phone from portrait mode to landscape mode.

The iPhone's accelerometer is a lot more powerful than this, however, and is capable of giving you live data for all three dimensions (x, y, and z). This means, for example, you could control onscreen elements simply by tilting the phone in the direction you want.

Accelerometer data is delivered to the applications by the **UIAccelerometer** singleton class and a single delegate method, **accelerometer:didAccelerate:**, which gives you the three axes as **UIAcceleration** objects. Under normal gravity, each of these will have a value between -1 and +1, with 0 being the middle or "flat" point A. If you increase gravity's effect on the iPhone by moving it rapidly (for instance in a "flicking" motion), then these values will increase.

You'll now create an application that allows you to control the red box from the earlier examples simply by tilting the phone.

To create a tilt-sensitive application:

1. Create a new view-based application, and save it as *TiltingExample*.
2. Open *TiltingExampleViewController.h*, add the **UIAccelerometerDelegate** protocol declaration, and create an instance variable to hold the tilt-controlled view (**Code Listing 7.15**).

Code Listing 7.15 The header file for the tilt application.

continues on next page

```
#import <UIKit/UIKit.h>

@interface TiltingExampleViewController : UIViewController <UIAccelerometerDelegate>
{
    UIView *redBox;
}
@end
```

- 3.** Back in TiltingExampleViewController.m, uncomment the `viewDidLoad` method, and set up the accelerometer:

```
[[UIAccelerometer shared  
    → Accelerometer]  
    → setDelegate:self];  
  
[[UIAccelerometer shared  
    → Accelerometer] setUpdateInterval:  
    → (1/40.0)];
```

This code tells the accelerometer to deliver events to you every 1/40th of a second.

You then create a red box view and add it to the main view, as in the earlier examples.

- 4.** Next, you need to create a method, `moveBoxWithX:andY:`, to move the red box around on the screen. First add this code:

```
CGPoint boxCenter = redBox.center;  
boxCenter.x += xAmount;  
boxCenter.y += yAmount;  
  
if (boxCenter.x < 50.0)  
    boxCenter.x = 50.0;  
if (boxCenter.x > 270.0)  
    boxCenter.x = 270.0;  
if (boxCenter.y < 50.0)  
    boxCenter.y = 50.0;  
if (boxCenter.y > 410.0)  
    boxCenter.y = 410.0;  
  
redBox.center = boxCenter;
```

This method gets the current center point (a `CGPoint`) of the red box view and increments its x and y values. After a couple of checks to make sure you haven't moved the box view offscreen, you tell the view to move to the new location.

5. The final step is to implement the `UIAccelerometer` delegate method `accelerometer:didAccelerate:`, as shown here:

```
float sensitivity = 25.0f;  
float xDistance = acceleration.x *  
→ sensitivity;  
float yDistance = acceleration.y *  
→ -sensitivity;  
[self moveBoxWithX:xDistance  
→ andY:yDistance];
```

You first set up a variable that will control how far you move by tilting; increasing this value will make the view move faster and/or further when you tilt. You next take the incoming x and y accelerometer values and calculate how many x and y pixels you should move before telling the red box view to move. A small tilt results in small x and y values and thus a small movement. A large tilt results in larger x and y values and so a larger movement. **Code Listing 7.16** shows the completed code.

6. Build and run the application (you will need to install it on an iPhone to test it).

Hold your iPhone so that it is face-up. You should be able to control the red box by tilting the phone.

TIP You may notice the red box is difficult to keep completely still. You've really created only a very simple example of how to deal with accelerometer data. In reality, you would normally want to implement some type of filter (often called a *low-pass* or *high-pass* filter) to "smooth out" the accelerometer data. Apple provides sample code on the Apple Developer Connection Web site (<http://developer.apple.com>) demonstrating this functionality.

TIP In the previous exercise, note how you negate the y value. This is because the iPhone coordinate system starts with 0 at the top of the screen and the accelerometer at the bottom of the screen.

Code Listing 7.16 The completed code for the tilt application.

```
#import "TiltingExampleViewController.h"

@implementation TiltingExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    //setup our accelerometer to update every 1/40th of a second
    [[UIAccelerometer sharedAccelerometer] setUpdateInterval:(1/40.0)];
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

    //create our red box view
    redBox = [[UIView alloc] initWithFrame:CGRectMake(110,180,100,100)];
    redBox.backgroundColor = [UIColor redColor];

    [self.view addSubview:redBox];
}

-(void)moveBoxWithX:(float)xAmount andY:(float)yAmount {
    CGPoint boxCenter = redBox.center;

    boxCenter.x += xAmount;
    boxCenter.y += yAmount;

    //don't allow box to go off-screen
    if (boxCenter.x < 50.0)
        boxCenter.x = 50.0;
    if (boxCenter.x > 270.0)
        boxCenter.x = 270.0;

    if (boxCenter.y < 50.0)
        boxCenter.y = 50.0;
    if (boxCenter.y > 410.0)
        boxCenter.y = 410.0;

    redBox.center = boxCenter;
}

- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAcceleration *)acceleration {
    float sensitivity = 25.0f;
    float xDistance = acceleration.x * sensitivity;
    float yDistance = acceleration.y * -sensitivity;

    [self moveBoxWithX:xDistance andY:yDistance];
}

- (void)dealloc {
    [redBox release];
    redBox = nil;

    [super dealloc];
}
```

8

Location and Mapping

Having an iPhone means that you need never get lost again. With the iPhone's built-in Global Positioning System (GPS) hardware and some innovative location and mapping software, not only does your iPhone know where you are at any time, but it can also *show* you.

The iPhone uses a technology known as *assisted GPS* to work out where you are. In addition to the built-in GPS receiver, the iPhone uses triangulation information from cellular towers and Wi-Fi hot spots to increase the accuracy of the location data it delivers to your applications.

In this chapter, you'll first take a look at Core Location—the framework that lets you quickly and easily find your current location. You'll then look at the Map Kit framework, which enables you to add maps powered by the popular Google Maps engine to your own applications. You'll see how easy it is to perform reverse geocoding to get the address of a location, too. Finally, you'll combine all of these ideas into a mapping application that reproduces much of the functionality of the native Maps application.

In This Chapter

About Core Location	312
About Map Kit	325
Putting It All Together	341

About Core Location

Core Location is the framework used to add location awareness to your iPhone applications. It's deceptively simple in its design.

The **CLLocationManager** class and its delegates provide the mechanism by which location information gets delivered to your application. Simply create an instance, optionally set some accuracy properties, and then call the **startUpdatingLocation** method.

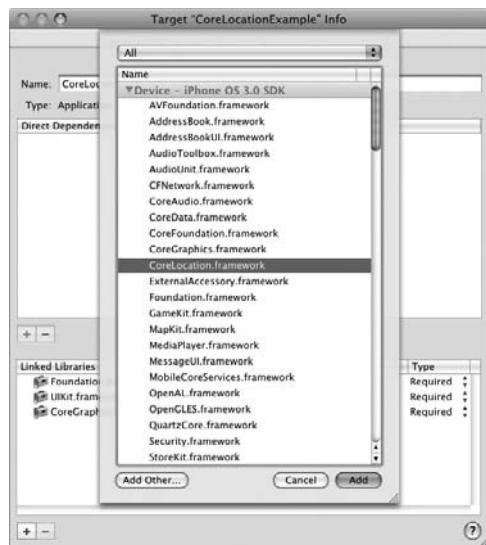
CLLocation events are generated via the delegate methods of your **CLLocationManager** instance. **CLLocation** objects encapsulate not only geographical coordinates but also information such as speed, altitude, heading, and accuracy.

To make your application location aware:

1. Create a new view-based application, saving it as **CoreLocationExample** **A**.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and choose Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add CoreLocation.framework **B**.
4. Open the CoreLocationExampleView Controller.h file, import the **CoreLocation** framework, add the **CLLocationManagerDelegate** protocol declaration, and create an instance variable to hold your location manager (**Code Listing 8.1**).



A Creating a view-based application.



B Adding the Core Location framework to your project.

5. Switch to the CoreLocationExample ViewController.m file, uncomment the **viewDidLoad** method, and add the following code:

```
lm = [[CLLocationManager alloc]
→ init];
lm.delegate = self;
[lm startUpdatingLocation];
```

You create your **CLLocationManager** instance, setting the delegate to be the view controller (**self**) and then telling the location manager to begin sending location events.

Code Listing 8.2 shows the completed application code.

Code Listing 8.1 The header file for the core location application.

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>

@interface CoreLocationExampleViewController : UIViewController <CLLocationManagerDelegate>
{
    CLLocationManager *lm;
}
@end
```

Code Listing 8.2 A bare-bones core location application.

```
#import "CoreLocationExampleViewController.h"

@implementation CoreLocationExampleViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    lm = [[CLLocationManager alloc] init];
    lm.delegate = self;
    [lm startUpdatingLocation];
}
```

Handling location updates

Believe it or not, that's it—your application is now location aware. However, you still don't have any way of dealing with the information the location manager is giving you. For this you must implement the required delegate methods (recall that you added the **CLLocationManagerDelegate** protocol back in the CLLocationManager.h file).

CLLocationManager has just two delegate methods. The first, **locationManager:didUpdateToLocation:fromLocation:**, is called whenever the location manager updates a new location. Both the previous location and the new location are passed unless this is the first location event, in which case **fromLocation:** will be **nil**.

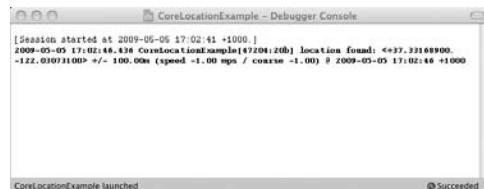
The second delegate method, **locationManager:didFailWithError:**, lets you deal with any errors that have occurred with the location manager trying to retrieve a location value. In this situation, you'd normally want to stop your location manager with a call to **stopUpdatingLocation** in order to save battery power.

Code Listing 8.3 shows the updated Core LocationExampleViewController.m file.

Output is sent to the console, showing latitude/longitude coordinates, accuracy of the result (in this case plus or minus 100m), your speed, your course or direction, and a time stamp of when the location event occurred .

Code Listing 8.3 The core location application updated to log to the console.

```
#import "CoreLocationExampleViewController.h"
@implementation CoreLocationExampleViewController
- (void)viewDidLoad
{
    [super viewDidLoad];
    lm = [[CLLocationManager alloc] init];
    lm.delegate = self;
    [lm startUpdatingLocation];
}
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLoc
fromLocation:(CLLocation *)oldLoc {
    NSLog(@"%@",newLoc.description);
    [lm stopUpdatingLocation];
}
- (void)locationManager:(CLLocationManager *)manager
didFailWithError:(NSError *)error {
    NSLog(@"%@",error);
    [lm stopUpdatingLocation];
}
- (void)dealloc {
    [super dealloc];
}
@end
```



 The Debugger Console displaying location information.

TIP Notice how, to preserve battery power, you call `stopUpdatingLocation` on the location manager as soon as you have either a location or an error.

TIP When the location manager is first started (via `startUpdatingLocation`), it will normally generate a number of location updates in quick succession. These are generally baseline or cached locations that are often inaccurate and can be ignored. You will investigate a technique to handle this situation later in the chapter.

TIP The CoreLocation framework also offers two other methods—`startMonitoringSignificantLocationChanges` and `stopMonitoringSignificantLocationChanges`—which work in exactly the same way as the method described here but result in fewer location updates being generated. These methods use significantly less power, so Apple recommends using them when your application does not need to be updated frequently.

Testing outside the simulator

If you are navigationally savvy, you may have noticed that when you run the previous example for yourself, the latitude and longitude do not correspond to your own location (that is, unless you are currently working for Apple). When Core Location-based code is run in the iPhone Simulator, it will always report a location of “1 Infinite Loop, Cupertino” (Apple’s headquarters).

Obviously, this makes testing elements such as course and speed impossible in the simulator. Additionally, if you were walking around with your iPhone testing your application, you wouldn’t be able to view console output on your computer screen.

From now on, you'll assume the examples in this chapter are running on an actual device rather than in the simulator. You'll add a **UITextView** to your iPhone's main view and write some code to log output to it instead of to the console.

You could just have easily hooked up **UILabel** controls or other, more attractive UI elements—this is just a convenient shortcut that allows you to easily display logging information with minimal effort.

To add logging of location data to the iPhone screen:

1. In CoreLocationExampleViewController.h, create an instance variable to hold your view (**Code Listing 8.4**):

```
UITextView *logView;
```

2. Switch to CoreLocationExampleViewController.m, and create the **logToScreen**: method you will use to log text to the iPhone's screen:

```
CGPoint pt = logView.  
→ contentOffset;
```

You first set a variable to hold the **UIView**'s current content position in its scroll view:

```
logView.text = [NSString  
→ stringWithFormat:@"%@",  
→ logView.text,output];
```

Code Listing 8.4 The updated header file.

```
#import <UIKit/UIKit.h>  
#import <CoreLocation/CoreLocation.h>  
  
@interface CoreLocationExampleViewController : UIViewController <CLLocationManagerDelegate>  
{  
    CLLocationManager *lm;  
    UITextView *logView;  
}  
  
@end
```



D Logging information is displayed on the iPhone.

Next you append the new text and make sure the view scrolls properly if the new text is too large for a single screen:

```
[logView setContentOffset:pt  
→ animated:NO];  
  
[logView scrollRangeToVisible:  
→ NSMakeRange([logView.text  
→ length], 0)];
```

3. Update the `viewDidLoad` method to create and add the new view:

```
logView = [[UITextView alloc]  
→ initWithFrame:[self.view  
→ bounds]];  
  
logView.editable = NO;  
logView.userInteractionEnabled =  
→ YES;  
  
[self.view addSubview:logView];
```

4. Finally, to make things work as closely to `NSLog()` as possible, define the `DCLog` macro at the top of the file:

```
#define DCLog(format, ...)  
→ [self logToScreen:[NSString  
→ stringWithFormat:format,  
→ ## __VA_ARGS__]];
```

You can now replace calls to `NSLog()` with a call to `DCLog()`. Logging information will be displayed on the iPhone's screen D.

Increasing the accuracy

Up to this point, you may have noticed that the location information you are receiving isn't particularly accurate. This is because the location manager often returns cached or baseline "best-guess" values when it's first started, becoming increasingly accurate over time.

You can do a number of things to filter out these unwanted location events and get a more accurate result.

You've used only the **description** property of the **CLLocation** events so far, but you could use a number of other properties:

- **altitude**—A positive or negative value depending on whether you are above or below sea level
- **coordinate**—A **CLLocationCoordinate** type containing latitude and longitude information
- **course, speed**—The direction and speed (in meters) in which the iPhone is traveling
- **horizontalAccuracy, verticalAccuracy**—How accurate the coordinate and altitude values are

TIP course is measured in degrees, with due north being 0, east being 90, south being 180, and west being 270. You probably noticed in the first example that both the course and speed values were -1.0. Having only a single location means these values can't be calculated.

TIP A negative value for horizontal Accuracy or verticalAccuracy generally indicates that the location event can be ignored.

Adding a timeout

You also want to think about saving iPhone power by timing out and turning off the location manager if you don't receive a valid location within a reasonable amount of time.

To add timeouts to the location manager:

1. In CoreLocationExampleViewController.h, add a new instance variable to hold your timeout:

```
NSTimer *timer;
```

- 2.** Switch to CoreLocationExampleView Controller.m, and add the following new code to your **viewDidLoad** method:

```
lm.desiredAccuracy =  
→ kCLLocationAccuracyBest;  
  
lm.distanceFilter =  
→ kCLLocationAccuracyNone;
```

This tells the location manager how accurate it should be and how far (in meters) the user must move laterally before a new location event is generated.

- 3.** Now set your timer to expire after 1 minute:

```
timer = [NSTimer scheduledTimer  
→ WithTimeInterval:60 target:self  
→ selector:@selector  
→ (locationManagerDidTimeout:  
→ userInfo:) userInfo:nil  
→ repeats:false];
```

The **desiredAccuracy** code tells the location manager how accurate it should attempt to be when generating location events. Possible values for this property range from “best” to 3km. It’s worth noting, however, that although the location manager will attempt to achieve the defined accuracy, it is not guaranteed.

In this example, you’re using the most accurate settings for both **desired Accuracy** and **distanceFilter**. It’s important to remember that using these settings may take longer to return a location and will have more of an impact on the iPhone battery life.

continues on next page

4. Update the location manager delegate to check the age of the location event:

```
NSTimeInterval eventAge =  
→ [newLocation.timestamp  
→ timeIntervalSinceNow];  
  
if (abs(eventAge) < 5.0)  
  
You are interested only in events that  
occurred within the last five sec-  
onds. This will filter out any cached  
location data.  
  
If an event is new enough, you next  
check the accuracy:  
  
if ([newLocation horizontal  
→ Accuracy] > 0.0f && [newLocation  
→ horizontalAccuracy] <= 100.0f)
```

which will give only those events that are accurate to within 100m of your actual position (making sure to also ignore negative, invalid values).

Lastly, you stop the location manager and timer and output the result.

5. There are still two more things to do. First you need to handle when a location can't be found:

```
(void)locationManagerDidTimeout:  
→ (NSTimer *)aTimer userInfo:(id)  
→ userInfo
```

This simply stops the location manager and logs an error to your screen.

6. Finally, make sure that the timer is stopped if an error occurs by adding the following to the **locationManager:didFailWithError:** delegate:

```
[timer invalidate];
```

Code Listing 8.5 shows the completed code.

Code Listing 8.5 The completed core location code.

```
#import "CoreLocationExampleViewController.h"

#define DCLog(format, ...) [self logToScreen:[NSString stringWithFormat:format, ## _VA_ARGS__]];

@implementation CoreLocationExampleViewController

- (void)logToScreen:(NSString *)output {
    CGPoint pt = logView.contentOffset;

    //write text
    logView.text = [NSString stringWithFormat:@"%@\n%@", logView.text,output];

    //tell view to scroll if necessary
    [logView setContentOffset:pt animated:NO];
    [logView scrollRangeToVisible:NSMakeRange([logView.text length], 0)];
}

- (void)viewDidLoad {
    [super viewDidLoad];

    //create logger view
    logView = [[UITextView alloc] initWithFrame:[self.view bounds]];
    logView.editable = NO;
    logView.userInteractionEnabled = YES;
    [self.view addSubview:logView];

    lm = [[CLLocationManager alloc] init];

    lm.delegate = self;

    lm.desiredAccuracy = kCLLocationAccuracyBest; //try to be as accurate as possible
    lm.distanceFilter = kCLDistanceFilterNone; //report all movement

    timer = [NSTimer scheduledTimerWithTimeInterval:60
                                              target:self
                                                selector:@selector(locationManagerDidTimeout:userInfo:)
                                              userInfo:nil
                                             repeats:false];

    [lm startUpdatingLocation];
}

- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation
fromLocation:(CLLocation *)oldLocation {

    //how old (in seconds) is the event?
    NSTimeInterval eventAge = [newLocation.timestamp timeIntervalSinceNow];

    //deal with cached locations by ignoring anything older than 5 seconds
    if (abs(eventAge) < 5)
    {
        //only look at locations that are within 100meters
        if ([newLocation horizontalAccuracy] > 0.0f && [newLocation horizontalAccuracy] <= 100.0f)
        {
            //stop the location manager
            [lm stopUpdatingLocation];

            //stop the timer
            [timer invalidate];

            DCLog(@"FINAL location found:\nCoords (%f, %f)\nAlt:%f\nSpeed:%f\nHoriz Accuracy:%f\nVert Accuracy:%f\n\n",
                  newLocation.coordinate.latitude,
                  newLocation.coordinate.longitude,
                  newLocation.speed,
                  newLocation.horizontalAccuracy,
                  newLocation.verticalAccuracy);
        }
    }
}
```

code continues on next page

Code Listing 8.5 *continued*

```
    }
    else
    {
        DCLog(@"APPROX Location found: %@",newLocation.description);
    }
}

- (void)locationManager:(CLLocationManager *)manager
didFailWithError:(NSError *)error
{
    //stop listening to location events
    [lm stopUpdatingLocation];

    //stop the timer
    [timer invalidate];

    DCLog(@"location failed with error: %@",error);
}

- (void)dealloc {
    [logView release];
    [lm release];

    [timer invalidate];
    [super dealloc];
}

@end
```



E The results of a location search.

Now you'll see the result of a successful location search E. Try running this code on your iPhone—you should see the location events as they arrive (you may need to increase the timeout value if you are indoors), increasing in accuracy over time. Try changing the values for **desiredAccuracy** and **distanceFilter** to see how they affect the results of changing or removing the check for cached locations.

TIP Since the location manager works by gradually increasing accuracy and “zeroing in” on a location, it actually may be the case that the first result, although reported as inaccurate, is in fact correct.

TIP There's always going to be a trade-off between displaying potentially inaccurate information quickly versus making the user wait longer for a more accurate result. The code in the previous example is just one approach. Another idea might be to decrease the **desiredAccuracy** level and instead adopt a strategy of counting location events, waiting for a number to arrive before accepting one as valid. The method you choose to use will be determined by your application's requirements.

Accessing the compass

If your iPhone contains a compass, you can receive heading information in much the same way as location updates.

Heading updates are generated by the location manager as **CLHeading** objects, containing attributes for heading values in relation to both magnetic and true north. These attributes are measured in degrees, with **0** representing north and **180** representing south.

To check whether the iPhone supports heading updates, you use the **headingAvailable** class method of the location manager. To start and stop listening to heading events, you call the **startUpdatingHeading** and **stopUpdatingHeading** methods accordingly.

Just as with location updates, heading updates are managed via delegate methods:

- **locationManager:didUpdateHeading:**—Called whenever a new heading is received that differs from the previous heading by more than the value specified in the **headingFilter** property.
- **locationManagerShouldDisplayHeadingCalibration:manager**—Called if your iPhone needs to be calibrated. This will typically happen only with a new iPhone where the compass has never been used. Returning **YES** will cause the compass calibration panel to appear. You can dismiss the panel by calling **dismissHeadingCalibrationDisplay**, or you can prevent it from ever showing by returning **NO** to this method.

In the examples so far, you have concentrated on getting a *single* location as accurately as you can before stopping the location manager. You might want to have a real-world application that updates locations continuously, displaying the course, speed, and heading information as you walk or drive around. You will see an example of this in the “Putting It All Together” section later in this chapter.



A Adding the Map Kit framework to your project.

Code Listing 8.6 The header file for the map application.

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>

@interface MappingExampleViewController : UIViewController
{
    MKMapView *map;
}
@end
```

About Map Kit

Map Kit, the mapping framework based on the Google Maps engine, gives you the ability to add interactive maps to your applications. Maps can be scrolled and zoomed to any region on the planet and can have pins, or *annotations*, added to the map to display additional information.

To add a map to your application:

1. Create a new view-based application, saving it as **MappingExample**.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add **MapKit.framework** A.
4. Open the **MappingExampleView Controller.h** file, import the **MapKit** framework, and create an instance variable to hold your map view (**Code Listing 8.6**).

continues on next page

5. Switch to the `MappingExampleViewController.m` file, uncomment the `viewDidLoad` method, and add the following code:

```
map = [[MKMapView alloc]
→ initWithFrame:[self.view
→ bounds]];
[self.view addSubview:map];
```

Code Listing 8.7 shows the updated code.

6. Build and run the application ❸.

That's it—two lines of code, and you have a map! You should be able to navigate around the map and zoom in/out by using the “pinch” gesture.



❸ The application with a full-screen `MKMapView`.

Code Listing 8.7 The completed code for a bare-bones map application.

```
#import "MappingExampleViewController.h"

@implementation MappingExampleViewController

- (void)viewDidLoad {
    map = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    [self.view addSubview:map];
}

- (void)dealloc {
    [map release];
    [super dealloc];
}

@end
```



➊ Adding the Core Location framework to your project.

You'll now update this code to make it a little more interesting.

To show your location on the map:

1. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
2. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add CoreLocation.framework ➋. Although this is not strictly necessary for this example, you will be using some convenience functions within the CoreLocation framework.
3. Open MappingExampleViewController.m, and update your viewDidLoad method to set the map type to a satellite view and tell it to show the current location, indicated by an animated blue marker:

```
map.mapType = MKMapTypeSatellite;  
map.showsUserLocation = YES;
```

The outer circle on this marker indicates the accuracy of the location data—the wider the circle, the less accurate.

```
CLLocationCoordinate2D coords =  
    → CLLocationCoordinate  
    → 2DMake(37.331689, -122.03071);
```

continues on next page

4. Next you create a variable to hold your map center coordinate, in this case Apple's headquarters:

```
float zoomLevel = 0.002;  
  
MKCoordinateRegion region =  
→ MKCoordinateRegionMake(coords,  
→ MKCoordinateSpanMake(zoomLevel,  
→ zoomLevel));  
  
[map setRegion:[map regionThatFits:  
→ region] animated:YES];
```

5. To zoom into a map, you need to create an **MKCoordinateRegion**.

This structure contains not only the coordinates the map should center on but also a *span*, which is comprised of a horizontal and vertical distance determining how much of the map (in degrees) should be shown. A large span creates a zoomed-out view; a small span creates a zoomed-in view

D. **Code Listing 8.8** shows the updated **viewDidLoad** method.



D Displaying the current location on a map.

Code Listing 8.8 Updating the code to set and show a location.

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    map = [[MKMapView alloc] initWithFrame:[self.view bounds]];  
  
    map.mapType = MKMapTypeSatellite;  
    map.showsUserLocation = YES;  
  
    CLLocationCoordinate2D coords = CLLocationCoordinate2DMake(37.331689,-122.03071);  
  
    float zoomLevel = 0.002;  
    MKCoordinateRegion region = MKCoordinateRegionMake(coords,  
→ MKCoordinateSpanMake(zoomLevel,zoomLevel));  
    [map setRegion:[map regionThatFits:region] animated:YES];  
  
    [self.view addSubview:map];  
}
```

TIP Just as with the iPhone's native Maps application, you can display three possible maps with the `mapType` property:

MKMapTypeStandard—Shows a normal map containing street and road names. This is the default map type if none is specified.

MKMapTypeSatellite—Shows a satellite view.

MKMapTypeHybrid—Shows a combination of the two, in other words, a satellite view with road and street information overlaid.

When setting span values, depending on which map type you are using, you may be able to zoom in or out to a greater degree. For example, maps of `MKMapTypeSatellite` generally contain greater detail and will allow you to zoom in a lot more than `MKTypeStandard`.

Map Overlays

Map Kit allows you to overlay content over arbitrary areas of your maps. By using the `MKCircle`, `MKPolygon`, or `MKPolyline` class, you can easily define and draw shapes on your maps defined by coordinates.

Creating an overlay is a two-step process:

1. Create an overlay object (an object that conforms to the `MKOverlay` protocol), to which you assign one or more coordinate data points.
2. Create an overlay view object (an `MKOverlayView` subclass), which is used to visually represent the data in your overlay object.

Now you'll look at how you might go about updating the application, using overlays to draw a route on your map.

To draw a route on a map using overlays:

1. Open MappingExampleViewController.h, and add the `MKMapViewDelegate` protocol declaration to the `@interface`:

```
@interface MappingExampleView
→ Controller : UIViewController
→ <MKMapViewDelegate>
```

2. Open MappingExampleViewController.m, and add the following to the end of your `viewDidLoad` method:

```
map.delegate = self;
[self createRoute];
```

First you set the delegate of your `MKMapView` object, necessary since the map overlay is drawn in a delegate method.

You then call a custom method used to create the route information:

```
CLLocationCoordinate2D
→ routeCoords[5];
routeCoords[0] =
→ CLLocationCoordinate2DMake
→ (37.331689, -122.03071);
routeCoords[1] =
→ CLLocationCoordinate2DMake
→ (37.331689, -122.03221);
routeCoords[2] =
→ CLLocationCoordinate2DMake
→ (37.330259, -122.03221);
routeCoords[3] =
→ CLLocationCoordinate2DMake
→ (37.330259, -122.03171);
routeCoords[4] =
→ CLLocationCoordinate2DMake
→ (37.330519, -122.03055);
```



E Displaying a route on the map.

TIP By using the `MKCircle` or `MKPolygon` class, you can define contiguous or closed regions on a map. These can then be filled with a color to indicate such things as country boundaries or other areas of interest.

```
MKPolyline *routeLine =  
→ [MKPolyline polylineWith  
→ Coordinates:routeCoords count:  
→ pointCount];  
  
[map addOverlay:routeLine];  
  
[routeLine release];
```

You first create and populate an array with the coordinates of the points on your route. You then create an `MKPolyline` object and pass it your coordinates array. This will allow you to plot and draw lines between the coordinate points of your route. Finally, you add this to the map as overlay.

3. Implement the `mapView:viewForOverlay:` delegate method:

```
MKPolylineView *plView =  
→ [[MKPolylineView alloc]  
→ initWithOverlay:overlay];  
  
plView.strokeColor =  
→ [UIColor redColor];  
  
plView.lineWidth = 5.0;  
  
return [plView autorelease];
```

This method is where you define how your route will actually be drawn.

You first create an `MKPolylineView` object, initializing it by passing it the overlay (the `MKPolyline`) you have already defined.

You then set the color and width of your line view and return it from the method, causing it to be drawn on the map.

Code Listing 8.9 shows the updated application.

4. Build and run the application.

TIP You should see the same map as in the previous example, with a red line plotting a route **E**.

Code Listing 8.9 The code updating to show a route.

```
- (void)createRoute {
    int pointCount = 5;
    CLLocationCoordinate2D routeCoords[pointCount];

    routeCoords[0] = CLLocationCoordinate2DMake(37.331689, -122.03071);
    routeCoords[1] = CLLocationCoordinate2DMake(37.331689, -122.03221);
    routeCoords[2] = CLLocationCoordinate2DMake(37.330259, -122.03221);
    routeCoords[3] = CLLocationCoordinate2DMake(37.330259, -122.03171);
    routeCoords[4] = CLLocationCoordinate2DMake(37.330519, -122.03055);

    MKPolyline *routeLine = [MKPolyline polylineWithCoordinates:routeCoords count:pointCount];
    [map addOverlay:routeLine];
    [routeLine release];
}

- (void)viewDidLoad {
    [super viewDidLoad];

    map = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    map.mapType = MKMapTypeSatellite;
    map.showsUserLocation = YES;

    CLLocationCoordinate2D coords = CLLocationCoordinate2DMake(37.331689,-122.03071);

    float zoomLevel = 0.002;
    MKCoordinateRegion region = MKCoordinateRegionMake(coords,
                                                       MKCoordinateSpanMake(zoomLevel,zoomLevel));
    [map setRegion:[map regionThatFits:region] animated:YES];

    [self.view addSubview:map];
    map.delegate = self;
    [self createRoute];
}

-(MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(id)overlay {
    MKPolylineView *plView = [[MKPolylineView alloc] initWithOverlay:overlay];
    plView.strokeColor = [UIColor redColor];
    plView.lineWidth = 5.0;

    return [plView autorelease];
}

- (void)dealloc
{
    [map release];
    [super dealloc];
}

@end
```

Adding annotations

To make your mapping applications richer and more interesting, you will often want to overlay information onto the map, and this is where annotations come in.

Map Kit contains support for adding not only simple “pin” annotations (as seen in the native iPhone Maps application) but also your own custom annotations that can have their own look and feel.

Adding annotations to a map involves a little more work than you’ve had to do so far.

To add an annotation to your map:

1. In the `MappingExampleView` Controller.m file, you need to create your own custom class that implements the **MKAnnotation** protocol. At the very least, this class must implement the **coordinate** property ([Code Listing 8.10](#)).
2. In `viewDidLoad`, set the delegate, and call a method to create your annotation by adding the following:

```
customAnnotation *annotation =  
→ [[customAnnotation alloc]  
→ initWithCoordinate:coords];  
annotation.title = @"The Title";  
annotation.subtitle = @"Subtitle";  
[map addAnnotation:annotation];  
[annotation release];
```

You’ve removed the `map.showsUserLocation = YES` line so that you can see your annotation. (Otherwise, the location marker and annotation will appear in the same place on the map.)

continues on next page

3. Finally, implement the `mapView:viewForAnnotation:`

`ForAnnotation:` delegate method to display the annotation as a pin:

```
MKPinAnnotationView *pinView =
→ (MKPinAnnotationView *) [map
→ dequeueReusableAnnotationView
→ WithIdentifier:annotation.title];

if (pinView == nil)
pinView = [[[MKPinAnnotationView
→ alloc] initWithAnnotation:
→ annotation reuseIdentifier:
→ annotation.title] autorelease];
else
    pinView.annotation =
→ annotation;
```

Code Listing 8.11 shows the updated code.

Code Listing 8.10 The custom annotation class.

```
@interface customAnnotation: NSObject <MKAnnotation>
{
    CLLocationCoordinate2D coordinate;
    NSString *title;
    NSString *subtitle;
}

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, retain) NSString *title;
@property (nonatomic, retain) NSString *subtitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)coords;

@end

@implementation customAnnotation

@synthesize coordinate, title, subtitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)coords {
    if (self = [super init])
        coordinate = coords;

    return self;
}

- (void)dealloc {
    [title release];
    [subtitle release];

    [super dealloc];
}

@end
```

Code Listing 8.11 The code updated to add a custom annotation.

```
#import "MappingExampleViewController.h"

@interface customAnnotation : NSObject <MKAnnotation>
{
    CLLocationCoordinate2D coordinate;
    NSString *title;
    NSString *subtitle;
}

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, retain) NSString *title;
@property (nonatomic, retain) NSString *subtitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)coords;

@end

@implementation customAnnotation

@synthesize coordinate, title, subtitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)coords {
    if (self = [super init])
        coordinate = coords;

    return self;
}

- (void)dealloc {
    [title release];
    [subtitle release];

    [super dealloc];
}

@end

@implementation MappingExampleViewController

- (void)createAnnotationWithCoords:(CLLocationCoordinate2D)coords
{
    customAnnotation *annotation = [[customAnnotation alloc] initWithCoordinate:coords];
    annotation.title = @"The title";
    annotation.subtitle = @"Subtitle";
    [map addAnnotation:annotation];
    [annotation release];
}

- (void)viewDidLoad {
    [super viewDidLoad];

    map = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    map.mapType = MKMapTypeSatellite;

    CLLocationCoordinate2D coords = CLLocationCoordinate2DMake(37.331689,-122.03071);

    float zoomLevel = 0.002;
    MKCoordinateRegion region = MKCoordinateRegionMake(coords,
                                                       MKCoordinateSpanMake(zoomLevel,zoomLevel));
    [map setRegion:[map regionThatFits:region] animated:YES];
}
```

code continues on next page

Code Listing 8.11 continued

```
[self.view addSubview:map];
map.delegate = self;
[self createAnnotationWithCoords:coords];
}

-(MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(id)overlay {
    MKPolylineView *plView = [[MKPolylineView alloc] initWithOverlay:overlay];
    plView.strokeColor = [UIColor redColor];
    plView.lineWidth = 5.0;

    return [plView autorelease];
}

-(void)dealloc
{
    [map release];
    [super dealloc];
}
@end
```

Code Listing 8.12 Displaying an image as an annotation.

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView viewForAnnotation:(id <MKAnnotation>)annotation {
    MKAnnotationView *aView;
    aView = (MKAnnotationView *) [map dequeueReusableAnnotationViewWithIdentifier:annotation.title];

    if (aView == nil)
        aView = [[[MKAnnotationView alloc]
                  initWithAnnotation:annotation reuseIdentifier:annotation.title] autorelease];
    else
        aView.annotation = annotation;

    [aView setImage:[UIImage imageNamed:@"iPhone.png"]];
    aView.canShowCallout = TRUE;

    return aView;
}
```



F Displaying an annotation on the map.



G A custom annotation showing a graphic.

The `mapView:viewForAnnotation:` delegate returns an `MKAnnotationView` object. In this example, you are using the `MKPinAnnotationView` subclass that, as the name suggests, displays annotations as pins F. You can, however, return your own subclass if you want a different look and feel for annotations.

You can see the simplest example of this in [Code Listing 8.12](#). You set the `image` property on the base `MKAnnotationView` class. This results in the map shown in G.

TIP Specifying a reuse queue when creating `MKAnnotationView` objects allows the mapping engine to remove annotations from the map when they move off the screen (for example, if the user zooms or scrolls the map). The annotation is taken out of the queue (also known as dequeuing) when it moves back on the screen.

TIP You can also alter the look and feel of the annotation's callout view (the view that shows up when the user touches the annotation) by overriding `viewForCalloutAccessoryPosition:` in your custom `MKAnnotationView` class.

For more information on using annotations and overlays, refer to the “Annotating Maps” section of the *Location Awareness Programming Guide*.

Adding reverse geocoding

Map Kit provides the facility to do lookups on latitude and longitude coordinates to get address information—a process known as *reverse geocoding*. You accomplish this via the **MKReverseGeocoder** class and its delegate methods.

In the previous example, you created your annotation manually, specifying the coordinates **title** and **subtitle** in the **viewDidLoad** method. You'll now change your code to perform a reverse lookup of your position instead, creating an annotation with your address in its callout view.

To add reverse geocoding:

1. Open `MappingExampleViewController.h`, and add the **MKReverseGeocodeDelegate** protocol declaration to the `@interface`:

```
@interface MappingExampleView
→ Controller : UIViewController
→ <MKMapViewDelegate,
→ MKReverseGeocodeDelegate>
```

You also create an instance variable to hold your reverse geocoder:

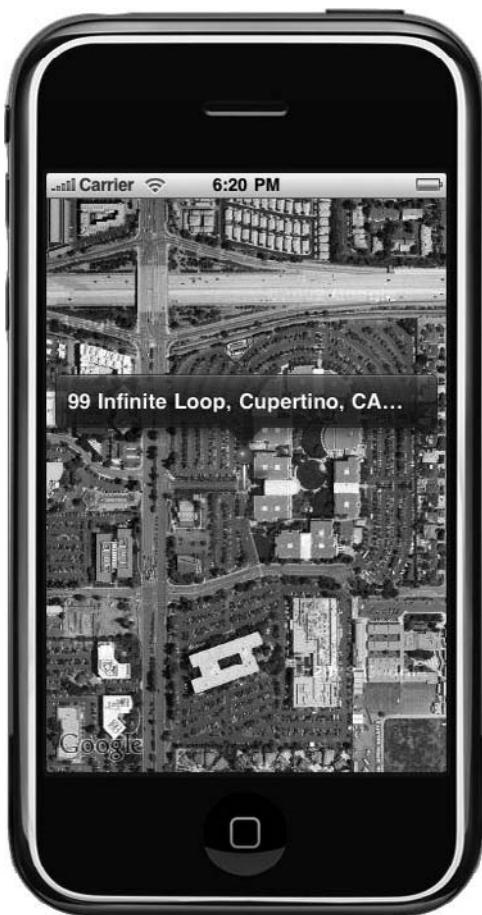
```
MKReverseGeocoder *geo;
```

Code Listing 8.13 shows the updated header file.

Code Listing 8.13 The header file updated to add reverse geocoding.

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>

@interface MappingExampleViewController : UIViewController <MKMapViewDelegate, MKReverseGeocodeDelegate >
{
    MKMapView *map;
    MKReverseGeocoder *geo;
}
@end
```



H Displaying the address in an annotation.

2. Back in `MappingExampleView` Controller.m, update the `viewDidLoad` method:

```
geo = [[MKReverseGeocoder alloc]
       initWithCoordinate:coords];
geo.delegate = self;
[geo start];
```

Here you create your geocoder instance, set the delegate, and tell it to start the lookup.

3. Finally, you need to implement the `MKReverseGeocode` delegate methods, adding an annotation if you find an address:

```
[map addAnnotation:placemark];
[geo cancel];
```

and logging a console message on failure:

```
NSLog(@"geo error: %@",error);
[geo cancel];
```

Notice that you cancel the geocoder in both situations. **Code Listing 8.14** shows the completed code. Now your application shows the address H.

TIP In the `reverseGeocoder:didFindPlaceMark:` delegate, you add an `MKPlacemark` as an annotation. This class contains properties to hold location information such as the city or state. Just like when you created your own custom annotation class in a previous example, this class implements the `MKAnnotation` protocol, so it can be placed on a map.

Code Listing 8.14 The code updated to perform a reverse-geo lookup and add the address as an annotation to the map.

```
#import "MappingExampleViewController.h"

@implementation MappingExampleViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    map = [[MKMapView alloc] initWithFrame:[self.view bounds]];
    map.mapType = MKMapTypeSatellite;
    CLLocationCoordinate2D coords = CLLocationCoordinate2DMake(37.331689,-122.03071);
    float zoomLevel = 0.002;
    MKCoordinateRegion region = MKCoordinateRegionMake(coords,
                                                       MKCoordinateSpanMake(zoomLevel,zoomLevel));
    [map setRegion:[map regionThatFits:region] animated:YES];
    [self.view addSubview:map];
    map.delegate = self;
    geo = [[MKReverseGeocoder alloc] initWithCoordinate:coords];
    geo.delegate = self;
    [geo start];
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFailWithError:(NSError *)error {
    NSLog(@"%@",@"reverse geo lookup failed with error: %@",error);
    [geo cancel];
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFindPlacemark:(MKPlacemark *)placemark {
    [map addAnnotation:placemark];
    [geo cancel];
}

- (void)dealloc
{
    [map release];
    [geo cancel];
    [geo release];
    [super dealloc];
}

@end
```

Putting It All Together

So far you've seen how to get your location, draw and plot a location on a map, and perform reverse geocoding to get the address of a location.

Now you'll combine all of these ideas into a single application that automatically updates as you walk or drive around. You'll also add some fields to show your current address, compass heading, and distance traveled **A**.

To update your application:

1. In the Groups & Files pane, expand the Targets section, right-click the application target, choose Get Info, and add CoreLocation.framework.
2. Open MappingExampleViewController.h, import the Core Location framework header, and add the **CLLocationManagerDelegate** protocol.

continues on next page



A The final completed mapping application.

3. Now add some new instance variables:

```
CLLocationManager *lm;  
UITextView *addressView;  
UITextView *distanceView;  
UITextView *headingView;  
CLLocation *startingLocation;  
CLLocation *lastPlottedLocation;  
double previousHeading;
```

The first holds the location manager, as in previous examples. Then you define a couple of views to display address distance information and the compass heading in your application. Finally, you need somewhere to hold your initial location when calculating the distance traveled, a variable to hold the last plotted location (so you can draw your route), and a variable to keep track of whether the heading has changed.

Code Listing 8.15 shows the header file.

Code Listing 8.15 The header file for the completed mapping application.

```
#import <UIKit/UIKit.h>  
#import <CoreLocation/CoreLocation.h>  
#import <MapKit/MapKit.h>  
  
@interface MappingExampleViewController : UIViewController <MKMapViewDelegate,  
MKReverseGeocoderDelegate,  
CLLocationManagerDelegate>  
{  
    MKMapView *map;  
    MKReverseGeocoder *geo;  
    CLLocationManager *lm;  
    UITextView *addressView;  
    UITextView *distanceView;  
    UITextView *headingView;  
  
    CLLocation *startingLocation;  
    CLLocation *lastPlottedLocation;  
  
    double previousHeading;  
}  
  
@property (retain) CLLocation *startingLocation;  
@property (retain) CLLocation *lastPlottedLocation;  
  
@end
```

- 4.** In the MappingExampleView Controller.m file, edit your `viewDidLoad` method to add all your new UI elements.

Set up the height of your address distance views and heading; then create a `CGRect` the same size as the main view:

```
float viewHeight = 25.0;  
CGRect viewRect = [self.view  
→ bounds];
```

You create a `CGRect` for your address view, at the bottom of your main view, and then set the color and alignment. You do the same thing for the distance and heading views, placing them at the top of the main view:

```
CGRect addressViewRect =  
→ CGRectMake(viewRect.origin.x,  
→ viewRect.size.  
→ height-viewHeight,viewRect.size.  
→ width,viewHeight);  
  
addressView = [[UITextView alloc]  
→ initWithFrame:addressViewRect];  
  
addressView.backgroundColor =  
→ [UIColor blackColor];  
  
addressView.textColor =  
→ [UIColor whiteColor];  
  
addressView.textAlignment =  
→ NSTextAlignmentCenter;
```

You need to adjust the height and y position of the map to make room for the address and distance views:

```
viewRect.size.height -=  
→ viewHeight*2;  
  
viewRect.origin.y += viewHeight;
```

The rest of the method is much the same as in previous examples.

continues on next page

5. Next, add a couple of helper methods that are called when you get a new location event:

updateMapViewWithLocation:

shouldZoom: will update your map's position.

updateGeocoderWithLocation: will create a new **MKReverseGeocoder**.

plotLocation: will plot your current location on the map.

updateHeadingView will be called whenever a new compass heading is received.

6. Finally, implement the delegates for the reverse geocoder, map overlay, and location manager. This allows you to update your map and address views whenever you receive a new location.

You also display an error dialog box if you fail to get a location. **Code Listing 8.16** shows the completed code.

Code Listing 8.16 The completed mapping application code.

```
@implementation MappingExampleViewController

@synthesize startingLocation,lastPlottedLocation;

- (void)viewDidLoad
{
    float viewHeight = 25.0;
    CGRect viewRect = [self.view bounds];

    CGRect addressViewRect = CGRectMake(viewRect.origin.x,
                                         viewRect.size.height,
                                         viewRect.size.width,
                                         viewHeight);

    addressView = [[UITextView alloc] initWithFrame:addressViewRect];
    addressView.backgroundColor = [UIColor blackColor];
    addressView.textColor = [UIColor whiteColor];
    addressView.textAlignment = NSTextAlignmentCenter;

    CGRect distanceViewRect = CGRectMake(viewRect.origin.x,
                                         viewRect.origin.y,
                                         viewRect.size.width-50.0,
                                         viewHeight);

    distanceView = [[UITextView alloc] initWithFrame:distanceViewRect];
    distanceView.backgroundColor = [UIColor blackColor];
    distanceView.textColor = [UIColor whiteColor];
    distanceView.textAlignment = NSTextAlignmentCenter;

    CGRect headingViewRect = CGRectMake(viewRect.size.width-50.0,
                                         viewRect.origin.y,
                                         50.0,
                                         viewHeight);

    headingView = [[UITextView alloc] initWithFrame:headingViewRect];
    headingView.backgroundColor = [UIColor blackColor];
    headingView.textColor = [UIColor whiteColor];
    headingView.textAlignment = NSTextAlignmentCenter;

    //adjust rect to make room for views
    viewRect.size.height -= viewHeight *2;
    viewRect.origin.y += viewHeight;

    //create map and set some properties
    map = [[MKMapView alloc] initWithFrame:viewRect];
    map.mapType = MKMapTypeSatellite;
    map.showsUserLocation = YES;
    map.delegate = self;

    //create location manager & set some properties
    lm = [[CLLocationManager alloc] init];
    lm.desiredAccuracy = kCLLocationAccuracyBest; //try to be as accurate as possible
    lm.distanceFilter = kCLDistanceFilterNone; //report all movement
    lm.delegate = self;

    //start listening
    [lm startUpdatingLocation];

    //compass?
    if (CLLocationManager.headingAvailable)
        [lm startUpdatingHeading];

    //add all the subviews to the main view
    [self.view addSubview:addressView];
    [self.view addSubview:map];
    [self.view addSubview:distanceView];
    [self.view addSubview:headingView];
}

}
```

code continues on next page

Code Listing 8.16 *continued*

```
- (void)updateMapViewWithLocation:(CLLocation *)location shouldZoom:(BOOL)doZoom
{
    //set zoom quite far so we can see updates frequently
    float zoomLevel = 0.0095;

    //if we haven't drawn map yet, need to set zoom level, otherwise just draw. this maintains any user-adjusted zoom
    if (doZoom) {
        MKCoordinateRegion region = MKCoordinateRegionMake(location.coordinate,
                                                          MKCoordinateSpanMake(zoomLevel,zoomLevel));
        [map setRegion:[map regionThatFits:region] animated:TRUE];
    }
    else
        [map setCenterCoordinate:location.coordinate animated:YES];
}

- (void)updateGeoCoderWithLocation:(CLLocation *)location
{
    //do reverse-geo lookup
    if (geo)
        [geo cancel];

    geo=[[MKReverseGeocoder alloc] initWithCoordinate:location.coordinate];
    geo.delegate=self;
    [geo start];
}

- (void)updateHeadingView
{
    NSString *headingText;
    if (previousHeading >= 45.0)
        headingText = @"N";
    if (previousHeading >= 45.0 && previousHeading < 90.0)
        headingText = @"NE";
    if (previousHeading >= 90.0 && previousHeading < 135.0)
        headingText = @"E";
    if (previousHeading >= 135.0 && previousHeading < 180.0)
        headingText = @"SE";
    if (previousHeading >= 180.0 && previousHeading < 225.0)
        headingText = @"S";
    if (previousHeading >= 225.0 && previousHeading < 270.0)
        headingText = @"SW";
    if (previousHeading >= 270.0 && previousHeading < 315.0)
        headingText = @"W";
    if (previousHeading >= 315.0)
        headingText = @"NW";

    headingView.text = headingText;
}

- (void)plotLocation:(CLLocation *)loc
{
    CLLocationCoordinate2D route[2];

    route[0] = lastPlottedLocation.coordinate;
    route[1] = loc.coordinate;

    MKPolyline *routeLine = [MKPolyline polylineWithCoordinates:route count:2];
    [map addOverlay:routeLine];
    [routeLine release];

    self.lastPlottedLocation = loc;
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFindPlacemark:(MKPlacemark *)placemark
{
    NSString *street = [placemark.addressDictionary objectForKey:@"Street"];
    NSString *city = [placemark.addressDictionary objectForKey:@"City"];
    addressView.text = [NSString stringWithFormat:@"%@, %@", street, city];
}
```

code continues on next page

Code Listing 8.16 *continued*

```
- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFailWithError:(NSError *)error
{
    [geo cancel];
}

-(MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(id)overlay
{
    MKPolylineView *plView = [[MKPolylineView alloc] initWithOverlay:overlay];
    plView.strokeColor = [UIColor redColor];
    plView.lineWidth = 3.0;

    return [plView autorelease];
}

-(void)locationManager:(CLLocationManager *)mgr didUpdateToLocation:(CLLocation *)newLoc fromLocation:(CLLocation *)oldLoc
{
    NSTimeInterval eventAge = [newLoc.timestamp timeIntervalSinceNow];

    //deal with cached locations by ignoring anything older than 5 seconds
    if (abs(eventAge) < 5)
    {
        //only look at locations that are within 10meters
        if ([newLoc horizontalAccuracy] > 0.0f && [newLoc horizontalAccuracy] <= 10.0f)
        {

            //is the first location we are recording?
            if (!startingLocation)
            {
                self.startingLocation = newLoc;
                self.lastPlottedLocation = newLoc;
                [self updateMapViewWithLocation:newLoc shouldZoom:YES];
            }
            else
                [self updateMapViewWithLocation:newLoc shouldZoom:NO];

            //update geocoder and distance view if we've changed location
            if (newLoc.coordinate.latitude != oldLoc.coordinate.latitude
                ||
                newLoc.coordinate.longitude != oldLoc.coordinate.longitude)
            {
                [self updateGeoCoderWithLocation:newLoc];
                distanceView.text = [NSString stringWithFormat:@"Distance travelled from start: %3.1fm",
                                     [newLoc distanceFromLocation:self.startingLocation]];

                [self plotLocation:newLoc];
            }
        }
    }
}

-(void)locationManager:(CLLocationManager *)manager didFailWithError:(NSError *)error
{
    [lm stopUpdatingLocation];
    if (CLLocationManager.headingAvailable)
        [lm stopUpdatingHeading];

    NSLog(@"%@",error);
}

-(void)locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading *)newHeading
{
    if (newHeading.trueHeading != previousHeading)
    {
        previousHeading = newHeading.trueHeading;
        [self updateHeadingView];
    }
}
```

code continues on next page

Code Listing 8.16 *continued*

```
- (void)dealloc
{
    [map release];
    [lm stopUpdatingLocation];
    [lm release];

    [geo release];
    [addressView release];
    [distanceView release];

    [startingLocation release];
    [lastPlottedLocation release];
}

[super dealloc];
```

9

Multimedia

The iPhone is a great portable multimedia device. It has a large, high-resolution display, a built-in camera (with video capture capabilities and multiple cameras on newer iPhones), directional audio, a powerful processor, and hardware support for playing all the common audio and video formats.

The “always-on” nature of the iPhone means that the media-rich Internet is never far away, and iTunes synchronizes with your computer so you can take your media with you. The multimedia frameworks of the iPhone SDK provide a comprehensive toolbox to use when adding these capabilities to your applications.

In this chapter, you’ll take a closer look at how to use these frameworks. You’ll learn to browse and retrieve images from the iPhone’s photo library, access and play audio from the iPod library, take pictures and video with the camera, and more.

In This Chapter

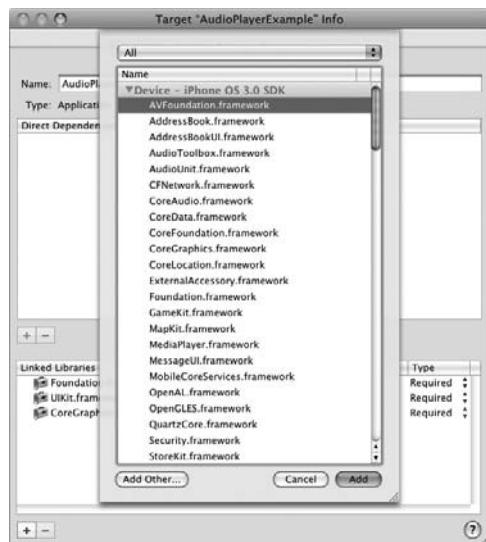
Playing Audio	350
Recording Audio	366
Using the iPhone’s Camera	371
Playing Video	381
Using the iPod Library	392

Playing Audio

For playing audio in your applications, the iPhone SDK provides the **AVAudioPlayer** class, which lets you play sound in any audio format supported by the iPhone, including AAC, AIFF, MP3, and WAV. All the functionality you would expect with an audio player is available: You can play multiple sounds simultaneously, pause the audio, jump to any position, control the volume, loop the audio, and more. You can also access properties such as the current position, the duration, and metering information.

To create an application to play audio:

1. Create a new view-based application, saving it as **AudioPlayerExample**.
2. In the Groups & Files list, expand the targets section, right-click your application target, and select **Get Info**.
3. Making sure the General tab is selected, click **Add (+)** at the bottom of the Linked Libraries list, and add **AVFoundation.framework** **A**.
4. Open the **AudioPlayerExampleViewController.h** file, add the **#import** statement to import the **AVFoundation.h** file, and create an instance variable to hold the audio player (**Code Listing 9.1**).



A Adding the **AVFoundation** framework to your application.

Code Listing 9.1 The header file of the audio player application.

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface AudioPlayerExampleViewController : UIViewController
{
    AVAudioPlayer *audioPlayer;
}

@end
```

5. Switch to the AudioPlayerExample ViewController.m file, uncomment the **viewDidLoad** method, and add the following:

```
NSString *filePath = [[[NSBundle  
→ mainBundle] resourcePath]  
→ stringByAppendingPathComponent:  
→ @"song.mp3"];  
  
NSURL *fileURL = [NSURL  
→ URLWithString:filePath];  
  
NSError *error = nil;  
  
audioPlayer = [[AVAudioPlayer  
→ alloc] initWithContentsOfURL:  
→ fileURL error:&error];  
  
if (error)  
    NSLog(@"Error: %@", error);  
else  
    audioPlayer.play;
```

This retrieves the audio file (named song.mp3 here) from the application bundle and then creates and initializes a new **AVAudioPlayer** with the audio file.

TIP Although **AVAudioPlayer** will work in most situations, it's not suitable if your application needs to have streaming audio or requires low-level control over the audio. For more information on how to work with audio, refer to the *Audio File Stream Services Reference* and *Core Audio* sections of the iPhone developer documentation.

6. Build and run the application.

The audio will play as soon as the application starts. It may be rather loud since the default value for the **volume** property is 100 percent. **Code Listing 9.2** shows the updated code.

Code Listing 9.2 The completed code for the audio player application.

```
- (void)viewDidLoad {  
  
    NSString *filePath = [[[NSBundle mainBundle] resourcePath]  
                           stringByAppendingPathComponent:@"song.mp3"];  
    NSURL *fileURL = [NSURL URLWithString:filePath];  
  
    NSError *error = nil;  
    audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL  
                                                       error:&error];  
  
    if (error)  
        NSLog(@"An error occurred: %@", error);  
    else  
        audioPlayer.play;  
}
```

Providing more control

Now you'll update this application to add Play, Pause, and Stop buttons. You'll also add a volume slider and a control to jump, or *scrub*, to any position in the audio file **B**.

To add controls to the audio player:

1. In the `AudioPlayerExampleViewController.h` header file, add the some instance variables that you'll use for your user interface:

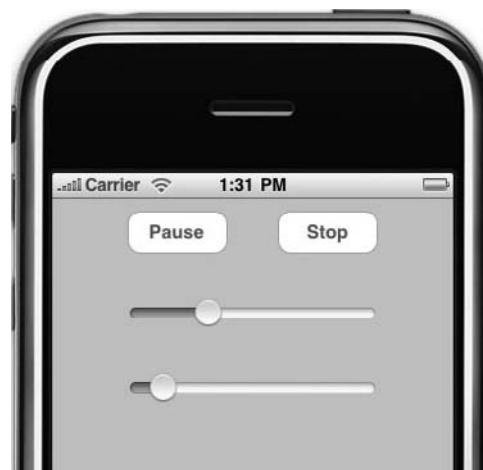
```
UIButton *playButton;  
UIButton *stopButton;  
  
UISlider *volume;  
UISlider *scrubber;  
  
UIProgressView *currentPosition;  
  
NSTimer *timer;
```

2. Switch to the `AudioPlayerExampleViewController.m` file, and in the `viewDidLoad` method, remove the call to the `play` method, since you don't want your audio to start playing until you tell it to. Set the following properties on the audio player:

```
audioPlayer.volume = 0.3;  
  
[audioPlayer prepareToPlay];
```

This sets the starting volume to be 1/3 maximum (volumes are measured as a value between 0 and 1.0). Then it calls the `prepareToPlay` method, which sets up the audio player and preloads its buffers. This is an important step to make sure your audio starts playing as soon as possible after its `play` method is called.

3. The `createControls` method is used to create the user interface. You create buttons for playing and pausing your audio, and another for stopping. You also create sliders to control the volume and to act as an audio scrubber.



B The updated application showing the new controls.

Notice how the continuous property of the volume control is set to Yes. This allows you to adjust the volume without removing your finger from the screen while the audio continues playing.

4. Next, implement the **play:** method which is called when the Play/Pause button is tapped:

```
if (! audioPlayer.playing)
{
    audioPlayer.play;
    [playButton setTitle:@"Pause"
    → forState:UIControlStateNormal];
}
else
{
    audioPlayer.pause;
    [playButton setTitle:@"Play"
    → forState:UIControlStateNormal];
}
```

You can use the **playing** property to check to see whether the audio player is playing. If so, pause the audio player, and switch the button title to Play. If not, begin playing, and change the title of the button to Pause.

5. Implement the **stop** method:

```
audioPlayer.stop;
audioPlayer.currentTime = 0;
[playButton setTitle:@"Play"
→ forState:UIControlStateNormal];
```

This stops the audio and resets the position to start. It also resets the text of the Play button in case the Stop button was tapped while the audio was paused.

continues on next page

- 6.** Implement the `changeVolume:` method that is called from your slider:

```
audioPlayer.volume = volume.value;
```

This sets the `volume` property of the audio player to the slider's current value.

- 7.** Finally, implement the `scrub:` method, which uses the `currentTime` property of the audio player to move to a different place in the audio.

- 8.** Build and run the application.

You can now play, pause, and stop the audio; adjust the volume; and jump to any position as the audio track plays. **Code Listing 9.3** shows the completed code.

Code Listing 9.3 The completed audio player code.

```
#import "AudioPlayerExampleViewController.h"

@implementation AudioPlayerExampleViewController

-(void)play:(id)sender {
    if (!audioPlayer.isPlaying) {
        audioPlayer.play;
        [playButton setTitle:@"Pause" forState:UIControlStateNormal];
    }
    else {
        audioPlayer.pause;
        [playButton setTitle:@"Play" forState:UIControlStateNormal];
    }
}

-(void)stop:(id)sender {
    audioPlayer.stop;
}

-(void)changeVolume:(id)sender {
    audioPlayer.volume = volume.value;
}

-(void)scrub:(id)sender {
    if (audioPlayer.isPlaying) {
        audioPlayer.pause;
        audioPlayer.currentTime = scrubber.value;
        audioPlayer.play;
    }
    else
        audioPlayer.currentTime = scrubber.value;
}

-(void)createControls {
```

code continues on next page

Code Listing 9.3 continued

```
playButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[playButton setFrame:CGRectMake(60,10,80,34)];
[playButton setTitle:@"Play" forState:UIControlStateNormal];
[playButton addTarget:self action:@selector(play:) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:playButton];

stopButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[stopButton setFrame:CGRectMake(180,10,80,34)];
[stopButton setTitle:@"Stop" forState:UIControlStateNormal];
[stopButton addTarget:self action:@selector(stop:) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:stopButton];

volume = [[UISlider alloc] initWithFrame:CGRectMake(60,80,200,0)];
[volume addTarget:self action:@selector(changeVolume:) forControlEvents:UIControlEventValueChanged];
volume.minimumValue = 0.0;
volume.maximumValue = 1.0;
volume.value = audioPlayer.volume;
volume.continuous = YES;
[self.view addSubview:volume];

scrubber = [[UISlider alloc] initWithFrame:CGRectMake(60,140,200,0)];
[scrubber addTarget:self action:@selector(scrub:) forControlEvents:UIControlEventValueChanged];
scrubber.minimumValue = 0.0;
scrubber.maximumValue = audioPlayer.duration;
scrubber.value = audioPlayer.currentTime;
scrubber.continuous = NO;
[self.view addSubview:scrubber];
}

-(void)viewDidLoad {
    NSString *filePath = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:@"song.mp3"];
    NSURL *fileURL = [NSURL fileURLWithPath:filePath];

    NSError *error = nil;
    audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL error:&error];

    if (error)
        NSLog(@"An error occurred: %@",error);
    else {
        audioPlayer.volume = 0.3;
        [audioPlayer prepareToPlay];
        [self createControls];
    }
}

-(void)dealloc {
    [scrubber release];
    [volume release];
    [audioPlayer release];

    [super dealloc];
}

@end
```

Responding to audio events

When you receive a call on your iPhone, it automatically mutes any audio that you're currently playing. The iPod application pauses your music, and once the call is complete, your audio continues playing right where it left off.

You can add this functionality to your own applications by adopting the **AVAudioPlayerDelegate** protocol and implementing its methods. This is also how you'll update the application to reset all the controls once the audio has finished playing.

To update the application to respond to events:

1. Open the `AudioPlayerExampleView` Controller.h file, and add the declaration to implement the **AVAudioPlayerDelegate** protocol ([Code Listing 9.4](#))

You also add a Boolean instance variable that you'll use to determine whether any audio was playing when the call came in.

Code Listing 9.4 Adding the **AVAudioPlayerDelegate** protocol declaration to the header file.

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface AudioPlayerExampleViewController : UIViewController <AVAudioPlayerDelegate>
{
    AVAudioPlayer *audioPlayer;
    UIButton *playButton;
    UIButton *stopButton;
    UISlider *volume;
    UISlider *scrubber;
    BOOL interrupted;
}
@end
```

2. Switch to the `AudioPlayerExample` `ViewController.m` file, and update the `viewDidLoad` method, setting the delegate on the audio player:

```
audioPlayer.delegate = self;
```

3. Implement three delegate methods (**Code Listing 9.5**).

The `audioPlayerBeginInterruption:` will be called if the audio is interrupted by a call. You use a Boolean variable to track whether the audio was playing.

The `audioPlayerEndInterruption:` will be called once you end the call. Here you check the Boolean set in the previous delegate and resume playing if necessary.

The `audioPlayerDidFinishPlaying:successfully:` delegate is called when the audio finishes. Here you call a method to reset all the controls (**Code Listing 9.6**).

Code Listing 9.5 Implementing the `AVAudioPlayerDelegate` methods.

```
- (void)audioPlayerBeginInterruption:(AVAudioPlayer *)player
{
    interrupted = audioPlayer.playing;
}

- (void)audioPlayerEndInterruption:(AVAudioPlayer *)player
{
    if (interrupted)
        audioPlayer.play;
}

- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player
                           successfully:(BOOL)flag
{
    [self resetControls];
}
```

Code Listing 9.6 The `resetControls` method sets all the audio controls back to their initial states.

```
- (void)resetControls
{
    audioPlayer.currentTime = 0;
    scrubber.value = 0;
    [playButton setTitle:@"Play"
                 forState:UIControlStateNormal];
}
```

Playing audio in the background

As part of the multitasking support introduced with iOS4, the iPhone is now capable of playing audio in the background. When a user exits your application, any currently playing audio will continue to play (unless, of course, another application starts with its own audio).

Note that at the time of writing, the iPhone Simulator does not currently support background audio; you will have to test your code on an actual iPhone.

To update the application to play audio in the background:

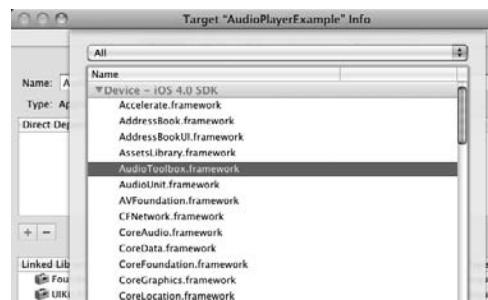
1. In the Groups & Files pane, expand the target section, right-click your application target, and select Get Info.
2. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add **AudioToolbox.framework** ①.
3. Open AudioPlayerExampleView Controller.h, and add the **#import** statement to import AudioToolbox.h (Code Listing 9.7).
4. Switch to AudioPlayerExampleView Controller.m, and add the following just before you create your audio player:

```
OSStatus status =
→ AudioSessionInitialize
→ (NULL, NULL, NULL, NULL);

UInt32 sessionCategory =
→ kAudioSessionCategory_
→ MediaPlayback;

status = AudioSessionSetProperty
→ (kAudioSessionProperty_
→ AudioCategory, sizeof
→ (sessionCategory),
→ &sessionCategory);

AudioSessionSetActive(YES);
```



① Adding the AudioToolbox framework.

Key	Value
Information Property List	(13 items)
Localization native development region	English
Required background modes	(1 item)
Main nib file base name (iPad)	<code>\$(PRODUCT_NAME)</code>
Main nib file base name (iPhone)	<code>\$(EXECUTABLE_NAME)</code>
Renders with edge antialiasing	<code>com.yourcompany</code>
Renders with group opacity	6.0
Required background modes	<code>\$(PRODUCT_NAME)</code>
Required device capabilities	APPL
Status bar is initially hidden	???
Status bar style	1.0
Supported external accessory protocols	<input checked="" type="checkbox"/>
Supported interface orientations	MainWindow
Main nib file base name	

D Updating the info.plist file.

This code is sets up an *audio session* for playing back media (as defined by `kAudioSessionCategory_MediaPlayback`). By using this category, your audio will continue to play when your application exits or when the screen locks.

The final step in getting background audio is to make a change to your application's info.plist file.

5. In the Groups & Files pane, select the Resources section, and then select `AudioPlayerExample-Info.plist`.
6. Right-click in the key column, and select Add Row. Select "Required background modes" from the list D.

continues on next page

Code Listing 9.7 The updated header file.

```
-<void>viewDidLoad {
    NSString *filePath = [[[NSBundle mainBundle] resourcePath]
        stringByAppendingPathComponent:@"song.mp3"];
    NSURL *fileURL = [NSURL fileURLWithPath:filePath];
    NSError *error = nil;

    OSStatus status = AudioSessionInitialize(NULL, NULL, NULL, NULL);
    UInt32 sessionCategory = kAudioSessionCategory_MediaPlayback;
    status = AudioSessionSetProperty (kAudioSessionProperty_AudioCategory,
        sizeof (sessionCategory),
        &sessionCategory);
    AudioSessionSetActive(YES);

    audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL
        error:&error];
    if (error)
        NSLog(@"An error occurred: %@",error);
    else
    {
        audioPlayer.volume = 0.3;
        [audioPlayer prepareToPlay];
        [self createControls];
    }
}
```

7. Expand the key you have just added, click the arrows to the right of Item 0, and select “App plays audio” **E**.

8. Build and run your application.

If you start the song playing and then press the Home button to exit your application, the audio will continue to play. **Code Listing 9.8** shows the updated **viewDidLoad** method.

▼ Required background modes		(1 item)
Item 0		App plays audio
Bundle display name		App plays audio
Executable file		App registers for location updates
Icon file		App provides Voice over IP services

E Setting the key in the info.plist file to enable background audio.

Code Listing 9.8 The updated **viewDidLoad** method for the background audio player example.

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
#import <AudioToolbox/AudioToolbox.h>

@interface AudioPlayerExampleViewController : UIViewController <AVAudioPlayerDelegate>
{
    AVAudioPlayer *audioPlayer;
    UIButton *playButton;
    UIButton *stopButton;
    UISlider *volume;
    UISlider *scrubber;
    BOOL interrupted;
}
@end
```

Controlling audio from the background

You've seen how to make your audio continue to play in the background even after your application has exited, but sometimes you might want to be able to control that audio. For example, you might have written a music player and still want to be able to change tracks and control the audio volume while running another application. Again, the multitasking features of the iOS4 SDK make it very easy for you to add this to your applications.

To update the application to control audio from the background:

1. Open AudioPlayerExampleViewController.m, and add the following line to your **viewDidLoad**: method:

```
[UIApplication sharedApplication]
→ beginReceivingRemoteControl
→ Events];
```

This tells your application that it can receive events from the external audio controls when it is running in the background.

2. Since remote audio events are sent via the responder chain, you must make sure that your view controller is capable of receiving them. This is done by implementing the following method:

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}
```

continues on next page

You'll also will need to make sure that your view controller is always the first responder after any user interaction has taken place, so in your `play:`, `changeVolume:`, and `scrub:` methods, you need to add the following:

- ```
[self becomeFirstResponder];
```
3. Finally, you implement the `remoteControlReceivedWithEvent:` method to capture remote control events:

```
switch (event.subtype)
{
 case UIEventTypeRemoteControl:
 → TogglePlayPause:
 → [self play:nil];
 break;
}
```

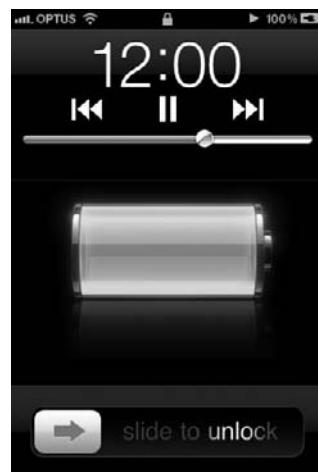
In this example, since you have only a Play/Stop button, you handle the `UIEventTypeRemoteControl` `TogglePlayPause` event; however, there are other events you can test for, such as skipping to the next chapter or seeking forward and backward through the movie. **Code Listing 9.9** shows the completed application.

#### 4. Build and run your application.

If you start playing the song and then press the Home button to exit the application, the audio will continue to play as previously. However, this time, if you double-tap the home button and swipe the multitasking area to the right, you will bring up the audio controls **F**. You can stop and start the audio by tapping the Play/Pause button. Notice also how your application icon has automatically appeared in the control—tapping it will open your application.



**F** Controlling background audio.



**G** Controlling the volume of background audio from the lock screen.

**TIP** In the example here, you used the **multitasking audio controls** to control your application's audio when in background mode. You can also control the audio from external the iPod controls. To do so, make sure your application audio is playing and then double-tap the home button from the lock screen to bring up the iPod controls. One advantage of controlling audio from here is that you also get a volume slider. **G**.

**Code Listing 9.9** The completed background audio player example.

```
#import "AudioPlayerExampleViewController.h"

@implementation AudioPlayerExampleViewController

-(void)resetControls
{
 audioPlayer.currentTime = 0;
 scrubber.value = 0;
 [playButton setTitle:@"Play"
 forState:UIControlStateNormal];
}

-(void)play:(id)sender {
 if (!audioPlayer.isPlaying) {
 audioPlayer.play;
 [playButton setTitle:@"Pause" forState:UIControlStateNormal];
 }
 else {
 audioPlayer.pause;
 [playButton setTitle:@"Play" forState:UIControlStateNormal];
 }
 [self becomeFirstResponder];
}

-(void)stop:(id)sender {
 audioPlayer.stop;
 [self resetControls];
}

-(void)changeVolume:(id)sender {
 audioPlayer.volume = volume.value;
 [self becomeFirstResponder];
}

-(void)scrub:(id)sender {
 if (audioPlayer.isPlaying) {
 audioPlayer.pause;
 audioPlayer.currentTime = scrubber.value;
 audioPlayer.play;
 }
 else
 audioPlayer.currentTime = scrubber.value;
 [self becomeFirstResponder];
}

-(void)createControls {
 //play/pause button
 playButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [playButton setFrame:CGRectMake(60,10,80,34)];
 [playButton setTitle:@"Play" forState:UIControlStateNormal];
 [playButton addTarget:self action:@selector(play:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:playButton];

 //stop button
 stopButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [stopButton setFrame:CGRectMake(180,10,80,34)];
 [stopButton setTitle:@"Stop" forState:UIControlStateNormal];
 [stopButton addTarget:self action:@selector(stop:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:stopButton];
}
```

*code continues on next page*

**Code Listing 9.9** continued

```
//volume control
volume = [[UISlider alloc] initWithFrame:CGRectMake(60,80,200,0)];
[volume addTarget:self action:@selector(changeVolume:)];
forControlEvents:UIControlEventValueChanged];
volume.minimumValue = 0.0;
volume.maximumValue = 1.0;
volume.value = audioPlayer.volume;
volume.continuous = YES;
[self.view addSubview:volume];

//scrubber control
scrubber = [[UISlider alloc] initWithFrame:CGRectMake(60,140,200,0)];
[scrubber addTarget:self action:@selector(scrub:)];
forControlEvents:UIControlEventValueChanged];
scrubber.minimumValue = 0.0;
scrubber.maximumValue = audioPlayer.duration;
scrubber.value = audioPlayer.currentTime;
scrubber.continuous = NO;
[self.view addSubview:scrubber];

}

-(void)remoteControlReceivedWithEvent:(UIEvent *)event {
switch (event.subtype) {
{
 case UIEventSubtypeRemoteControlTogglePlayPause:
 [self play:nil];
 break;

 case UIEventSubtypeRemoteControlNextTrack:
 //do nothing
 break;

 case UIEventSubtypeRemoteControlPreviousTrack:
 //do nothing
 break;
}
}

-(BOOL)canBecomeFirstResponder {
 return YES;
}

-(void)viewDidLoad {
 NSString *filePath = [[[NSBundle mainBundle] resourcePath]
 stringByAppendingPathComponent:@"song.mp3"];
 NSURL *fileURL = [NSURL fileURLWithPath:filePath];
 NSError *error = nil;

 OSSStatus status = AudioSessionInitialize(NULL, NULL, NULL, NULL);
 UInt32 sessionCategory = kAudioSessionCategory_MediaPlayback;
 status = AudioSessionSetProperty (kAudioSessionProperty_AudioCategory,
 sizeof (sessionCategory),
 &sessionCategory);
 AudioSessionSetActive(YES);

 audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL
 error:&error];
}
```

*code continues on next page*

**Code Listing 9.9** *continued*

```
if (error)
 NSLog(@"An error occurred: %@",error);
else
{
 audioPlayer.volume = 0.3;
 [audioPlayer prepareToPlay];
 [self createControls];
}

[[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
}

- (void)audioPlayerBeginInterruption:(AVAudioPlayer *)player
{
 interrupted = audioPlayer.isPlaying;
}

- (void)audioPlayerEndInterruption:(AVAudioPlayer *)player
{
 if (interrupted)
 audioPlayer.play;
}

- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player
 successfully:(BOOL)flag
{
 [self resetControls];
}

- (void)dealloc {
 [[UIApplication sharedApplication] endReceivingRemoteControlEvents];
 [scrubber release];
 [volume release];
 [audioPlayer release];
 [super dealloc];
}
@end
```

# Recording Audio

Recording audio on the iPhone is handled in much the same way as playing audio, only this time you use an instance of the **AVAudioRecorder** class. You can record audio of any length (assuming you have enough file space on the iPhone), pause and resume recording, or specify a recording of a specific length of time.

Just as with the **AVAudioPlayer** class, you can also determine the amount of time the recording has been running and access data about the audio levels (for input levels rather than output, obviously). Delegate methods handle interruptions to recording in the same way as the audio player.

## To create an application to record audio:

1. Create a new view-based application, saving it as **AudioRecorderExample**.
2. In the Groups & Files list, expand the targets section, right-click your application target, and choose **Get Info**.
3. Making sure the General tab is selected, click **Add (+)** at the bottom of the Linked Libraries list, and add **AVFoundation.framework**.
4. Open the **AudioRecorderExample.h** file, add the **#import** statement to import the **AVFoundation.h** file, and create instance variables to hold an audio recorder and an audio player (**Code Listing 9.10**).

**Code Listing 9.10** The header file for the audio recorder application.

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface AudioRecorderExampleViewController : UIViewController
{
 AVAudioPlayer *audioPlayer;
 AVAudioRecorder *audioRecorder;
}
@end
```

5. Switch to the AudioRecorder  
Example.m file, uncomment the **viewDidLoad** method, and add the following:

```
NSArray *paths = NSSearchPathFor
→ DirectoriesInDomains(NSDocument
→ Directory, NSUserDomainMask, YES);
NSString *documentsDirectoryPath =
→ [paths objectAtIndex:0];
NSString *filePath = [documents
→ DirectoryPath stringByAppending
→ PathComponent: @"myRecording.
→ caf"];
NSURL *fileURL = [NSURL file
→ URLWithPath:filePath];
NSError *error = nil;
audioRecorder = [[AVAudioRecorder
→ alloc] initWithURL:fileURL
→ settings:nil error:&error];
if (error)
 NSLog(@"error: %@",error);
else
{
 [audioRecorder prepareToRecord];
 [self createControls];
}
```

This retrieves the documents directory and then specifies a file path where you can save your recording.

Next, you create a new audio recorder object, and if there aren't any errors, you tell the audio recorder to prepare itself for recording. This ensures that recording begins as soon as possible after the **record** method is called, just as in the earlier exercise "To add controls to the audio player."

*continues on next page*

Notice that you pass **nil** for the **settings** parameter, causing the audio recorder to be created with its default settings. (This parameter is discussed in the “Controlling recording settings” sidebar.)

6. You next call the **createControls** method, which creates a user interface consisting of a Play button and a Record button.
7. The **record:** method is called when the user taps the Record button, toggling the recording on and off and updating the button title.
8. Similarly, the **play:** method checks to see whether recording is taking place, and attempts to play back the newly recorded audio.

Notice how you check for and release the audio player instance. This is necessary because you are loading the newly created recording each time you press play. **Code Listing 9.11** shows the completed code.

9. Build and run the application.

You may need to run it on an iPhone if you don’t have a microphone on your computer for the simulator to use.

**TIP** For a complete list of the settings available to audio recorders, refer to the **AVAudioRecorder Class Reference** in the iPhone developer documentation.

**Code Listing 9.11** The completed audio recorder code.

```
#import "AudioRecorderExampleViewController.h"

UIButton *recordButton;
UIButton *playButton;

@implementation AudioRecorderExampleViewController

-(void)record:(id)sender {
 if (!audioRecorder.recording) {
 audioRecorder.record;
 [recordButton setTitle:@"Stop" forState:UIControlStateNormal];
 playButton.enabled = NO;
 }
 else {
 audioRecorder.stop;
 [recordButton setTitle:@"Record" forState:UIControlStateNormal];
 playButton.enabled = YES;
 }
}
```

*code continues on next page*

**Code Listing 9.11** The completed audio recorder code.

```
- (void)play:(id)sender {
 if (!audioRecorder.recording)
 {
 if (audioPlayer)
 [audioPlayer release];

 NSError *error;
 audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:audioRecorder.url
 error:&error];
 if (error)
 NSLog(@"An error occurred: %@",error);
 else
 audioPlayer.play;
 }
}

-(void)createControls
{
 //record button
 recordButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [recordButton setFrame:CGRectMake(60,10,80,34)];
 [recordButton setTitle:@"Record" forState:UIControlStateNormal];
 [recordButton addTarget:self action:@selector(record:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:recordButton];

 //play/pause button
 playButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [playButton setFrame:CGRectMake(180,10,80,34)];
 [playButton setTitle:@"Play" forState:UIControlStateNormal];
 [playButton addTarget:self action:@selector(play:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:playButton];
}

-(void)viewDidLoad
{
 NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
 NSString *documentsDirectoryPath = [paths objectAtIndex:0];
 NSString *filePath = [documentsDirectoryPath stringByAppendingPathComponent:@"myRecording.caf"];
 NSURL *fileURL = [NSURL fileURLWithPath:filePath];

 NSError *error = nil;
 audioRecorder = [[AVAudioRecorder alloc] initWithURL:fileURL settings:nil error:&error];

 if (error)
 NSLog(@"error: %@",error);
 else {
 [audioRecorder prepareToRecord];
 [self createControls];
 }
}

-(void)dealloc {
 [audioRecorder release];
 audioRecorder = nil;

 [audioPlayer release];
 audioPlayer = nil;

 [super dealloc];
}

@end
```

## Controlling recording settings

In the “To create an application to record audio” exercise, you used the default recording settings to create the audio recorder by passing `nil` to the `settings` parameter. In most situations, this will be all you need, but audio recorders give you much more granular control over the recording, letting you set properties such as sample rate, number of channels, audio quality, bit rate, and more.

To configure audio settings, construct an **NSDictionary** of key-value pairs, and assign it to the `settings` property of your audio recorder. **Code Listing 9.12** shows the `viewDidLoad` method updated to set the audio quality, bit rate, number of channels, and sample rate of the recording.

**Code Listing 9.12** Defining some values for the recording settings property.

```
- (void)viewDidLoad
{
 NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
 NSString *documentsDirectoryPath = [paths objectAtIndex:0];
 NSString *filePath = [documentsDirectoryPath stringByAppendingPathComponent:@"myRecording.caf"];
 NSURL *fileURL = [NSURL fileURLWithPath:filePath];

 NSDictionary *audioSettings = [NSDictionary dictionaryWithObjectsAndKeys:
 [NSNumber numberWithInt:AVAudioQualityMin], AVEncoderAudioQualityKey,
 [NSNumber numberWithInt:16], AVEncoderBitRateKey,
 [NSNumber numberWithInt: 2], AVNumberOfChannelsKey,
 [NSNumber numberWithFloat:44100.0], AVSampleRateKey,
 nil];
}

NSError *error = nil;
audioRecorder = [[AVAudioRecorder alloc] initWithURL:fileURL settings:audioSettings error:&error];

if (error)
 NSLog(@"%@",error);
else {
 [audioRecorder prepareToRecord];
 [self createControls];
}
}
```

# Using the iPhone's Camera

You use the **UIImagePickerController** class to let users choose images from the iPhone's photo library and capture images or even video (if supported) from the built-in camera. **UIImagePickerController** is a view controller subclass, providing all the controls necessary for choosing and taking pictures and movies. As a developer, you simply create an instance of the class, set some properties, implement the delegate methods, and then tell the image picker to display. The control handles everything else for you.

The user selects an image or takes a picture (or movie), which is returned to the delegate. You then dismiss the image picker, returning control to your application.

You can let users edit an image or trim a movie by setting the **allowsEditing:** property to **YES**. Doing so returns an **NSDictionary** (along with the image or movie) to the delegate containing information about any edits that were made.

Setting the **sourceType** property of an image picker determines the type of interface displayed. There are three possible source types:

- **UIImagePickerControllerSourceTypePhotoLibrary**—This displays an interface that allows the user to select an image or movie from their photo library.
- **UIImagePickerControllerSourceTypeSavedPhotosAlbum**—This displays an interface that allows the user to select an image or movie from the camera roll or from the Saved Photos folder.
- **UIImagePickerControllerSourceTypeCamera**—This displays an interface that allows the user to take a picture or record a movie using the iPhone's camera.

Since the same control is used to pick images from the photo library and take pictures or record movies, you should check the source type is supported before trying to use it. You do this by using the **`isSourceTypeAvailable:`** class method, which returns a Boolean indicating whether a source type is supported. The iPod touch, for example, doesn't have a camera and would return **NO** for the source type **`UIImagePickerControllerSourceTypeCamera`**.

## To create an application with an image picker:

1. Create a new view-based application, saving it as `ImagePickerExample`.
2. Open the `ImagePickerExample.h` file, and add the **`UINavigationControllerDelegate`** and **`UIImagePickerControllerDelegate`** protocol declarations. You also create an instance variable to hold the image picker (**Code Listing 9.13**).

**Code Listing 9.13** The header file for the image picker application.

```
#import <UIKit/UIKit.h>

@interface ImagePickerExampleViewController : UIViewController <UINavigationControllerDelegate,
 UIImagePickerControllerDelegate>
{
 UIImagePickerController *imagePickerController;
}

@end
```

- 3.** Switch to the `ImagePickerExample.m` file, uncomment the `viewDidLoad` method, and add the following:

```
imagePickerController = [[UIImagePickerController alloc] init];
imagePickerController.delegate = self;
imagePickerController.allowsEditing = YES;
UIButton *photoLibraryButton =
 [UIButton buttonWithType:UIButtonTypeRoundedRect];
[photoLibraryButton setFrame:CGRectMake(100, 10, 120, 34)];
[photoLibraryButton setTitle:@"Photo Library" forState:UIControlStateNormal];
[photoLibraryButton addTarget:self action:@selector(photoLibraryButtonClick:) forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:photoLibraryButton];
```

This creates the image picker controller instance, setting the delegate and allowing users to edit any images they pick. You then create a button that will be used to launch the image picker.

- 4.** Implement the method that is called when the button is tapped:

```
[self presentModalViewController:imagePickerController animated:YES];
```

Since you are neither checking nor setting a source type, the default source type of `UIImagePickerControllerSourceTypePhotoLibrary` will be used.

*continues on next page*

5. Implement the **imagePickerController:**  
**didFinishPickingMediaWithInfo:** and  
**imagePickerControllerDidCancel:**  
delegates with the same code:

```
[picker dismissModalViewControllerAnimated:YES];
```

This hides the image picker when the user selects an image or closes the image picker by tapping the Cancel button. At this stage, you will not be doing anything with the image returned from the picker. **Code Listing 9.14** shows the completed code.

**Code Listing 9.14** The completed code for the image picker application.

```
#import "ImagePickerExampleViewController.h"

@implementation ImagePickerExampleViewController

-(void)photoLibraryButtonClick:(id)sender
{
 [self presentModalViewController:imagePicker animated:YES];
}

-(void)viewDidLoad {
 imagePicker = [[UIImagePickerController alloc] init];
 imagePicker.delegate = self;
 imagePicker.allowsEditing = YES;

 UIButton *photoLibraryButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [photoLibraryButton setFrame:CGRectMake(100, 10, 120, 34)];
 [photoLibraryButton setTitle:@"Photo Library" forState:UIControlStateNormal];
 [photoLibraryButton addTarget:self action:@selector(photoLibraryButtonClick:)
 forControlEvents:UIControlEventTouchUpInside];

 [self.view addSubview:photoLibraryButton];
}

-(void)imagePickerController:(UIImagePickerController *)picker didFinishPickingMediaWithInfo:(NSDictionary *)info
{
 NSLog(@"%@",info);
 [picker dismissModalViewControllerAnimated:YES];
}

-(void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
 [picker dismissModalViewControllerAnimated:YES];
}

-(void)dealloc {
 [imagePicker release];
 imagePicker = nil;
 [super dealloc];
}

@end
```



A Browsing the photo library.



B Capturing photos and video with the iPhone.

If you run the application in the simulator and tap the button, you should be presented with a new display A. You can browse through the photo library and edit the image before selecting.

**TIP** By setting the source type to `UIImagePickerControllerSourceTypePhotoLibrary`, you can use the `isSourceTypeAvailable` property to determine whether the user's photo library is empty (it will return NO).

**TIP** When recording movies, you are limited to a maximum of ten minutes. If the movie is longer, the user will have to trim it using the built-in editing controls before saving.

## Taking photos and video

You've seen how to select photos from the photo library. Now you'll update the application by adding a button that lets the user take a photo or, if supported by the iPhone, a video. You'll also enhance the code to check that both the photo library and the camera are available as source types and, if not, display an error message B.

## To update the application to take a photo or video:

1. Update the `viewDidLoad` method to check for the availability of both camera and video support on the iPhone:

```
if ([UIImagePickerController
→ isSourceTypeAvailable:
→ UIImagePickerControllerSource
→ TypeCamera])
{
 NSArray *availableMedia =
→ [UIImagePickerController
→ availableMediaTypesForSourceType:
→ UIImagePickerControllerSource
→ TypeCamera];
 imagePicker.mediaTypes =
→ availableMedia;
}
```

This code retrieves the array of supported media types using the `availableMediaTypesForSourceType` class method. An iPhone with video support will have two elements in the array: one for the camera and one for video. Older iPhones will have only the camera element.

You then assign this array to the `mediaTypes` property. Notice the result of writing the `availableMedia` array to the console for an iPhone with video support C.



The screenshot shows the Xcode debugger console window titled "ImagePickerControllerExample - Debugger Console". The output log shows the following text:  
2009-06-29 14:12:04.373 ImagePickerControllerExample[912:207] availableMedia: {  
 "public.image",  
 "public.movie"  
}  
Below the log, it says "GDB: Running..." and has a status bar indicating "Succeeded".

C The available camera media for an iPhone 3GS.

2. Create a second button for taking photos and videos:

```
UIButton *cameraButton =
→ [UIButton buttonWithType:UIButtonTypeRoundedRect];
[cameraButton setFrame:CGRectMake(100,60,120,34)];
[cameraButton setTitle:@"Camera"
→ forState:UIControlStateNormal];
[cameraButton addTarget:self
→ action:@selector
→ (cameraLibraryButtonClick:)
→ forControlEvents:
→ UIControlEventTouchUpInside];
[self.view addSubview:cameraButton];
```

3. Update the `photoLibraryButtonClick:` method to check that the source type is available:

```
if ([UIImagePickerController
→ isSourceTypeAvailable:
→ UIImagePickerControllerSource
→ TypePhotoLibrary])
{
 imagePickerController.sourceType =
→ UIImagePickerControllerSource
→ TypePhotoLibrary;
 [self presentModalViewController:
→ imagePickerController animated:YES];
}
else
 [self displaySourceError];
```

If the source type is not available, the preceding code will display an error message.

*continues on next page*

**4.** Implement the code for the **camera**

**LibraryButtonClick:** method to launch the image picker in camera mode :

```
if ([UIImagePickerController
 → isSourceTypeAvailable:UIImage
 → PickerControllerSourceType
 → Camera])
{
 imagePickerController.sourceType =
 → UIImagePickerControllerSource
 → TypeCamera;
 [self presentModalView
 → Controller:imagePickerController
 → animated:YES];
}
else
 [self displaySourceError];
```

Notice that you check that the camera is available before setting the source type to display the camera interface in the image picker. **Code Listing 9.15** shows the completed code.

**5.** Build and run the application.

You will need to run the application on an iPhone because the simulator doesn't support the camera as a source type.

**Code Listing 9.15** The updated camera application.

```
#import "ImagePickerExampleViewController.h"

@implementation ImagePickerExampleViewController

-(void)displaySourceError {
 UIAlertView *myAlert = [[UIAlertView alloc] initWithTitle:@"Error"
 message:@"Image source not available"
 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];
 [myAlert show];
 [myAlert release];
}
```

*code continues on next page*

**Code Listing 9.15** The updated camera application.

```
-<void>photoLibraryButtonClick:(id)sender {
 if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypePhotoLibrary]) {
 imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
 [self presentModalViewController:imagePicker animated:YES];
 }
 else
 [self displaySourceError];
}

-<void>cameraLibraryButtonClick:(id)sender {
 if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
 imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;
 [self presentModalViewController:imagePicker animated:YES];
 }
 else
 [self displaySourceError];
}

-(void)viewDidLoad {
 imagePicker = [[UIImagePickerController alloc] init];
 imagePicker.delegate = self;
 imagePicker.allowsEditing = YES;

 if ([UIImagePickerController isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
 NSArray *availableMedia = [UIImagePickerController
 availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];
 imagePicker.mediaTypes = availableMedia;
 NSLog(@"%@",availableMedia);
 }

 UIButton *photoLibraryButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [photoLibraryButton setFrame:CGRectMake(100,10,120,34)];
 [photoLibraryButton setTitle:@"Photo Library" forState:UIControlStateNormal];
 [photoLibraryButton addTarget:self action:@selector(photoLibraryButtonClick:)
 forControlEvents:UIControlEventTouchUpInside];

 [self.view addSubview:photoLibraryButton];

 UIButton *cameraButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [cameraButton setFrame:CGRectMake(100,60,120,34)];
 [cameraButton setTitle:@"Camera" forState:UIControlStateNormal];
 [cameraButton addTarget:self action:@selector(cameraLibraryButtonClick:)
 forControlEvents:UIControlEventTouchUpInside];

 [self.view addSubview:cameraButton];
}

-(void)imagePickerController:(UIImagePickerController *)picker didFinishPickingMediaWithInfo:(NSDictionary *)info {
 NSLog(@"%@",info);
 [picker dismissModalViewControllerAnimated:YES];
}

-(void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
 [picker dismissModalViewControllerAnimated:YES];
}

- (void)dealloc {
 [imagePicker release];
 imagePicker = nil;
 [super dealloc];
}

@end
```

If your iPhone has video capabilities, you should now see the video button available in the bottom-right corner of the screen **D**. By inspecting the `mediaType` key, you can determine the type of media captured.

After selecting a picture, you'll see `UIImages` for both the original image and the edited image **E**. The area cropped by the user is represented as an `NSRect`.

Now check out the same console output resulting from creating a movie **F**. This time you see the path where the movie was saved.



**D** When using the camera, the video button is available for iPhones with video support.

```
2009-06-29 14:18:06.327 ImagePickerController[945:207] info: {
 UIImagePickerControllerCropRect = NSRect: {{19, 0}, {1276, 1271}};
 UIImagePickerControllerEditedImage = <UIImage: 0x143550>;
 UIImagePickerControllerMediaType = "public.image";
 UIImagePickerControllerOriginalImage = <UIImage: 0x14ed20>;
}
```

**E** Console output after taking a picture with the iPhone camera.

```
2009-06-29 14:20:16.205 ImagePickerController[945:207] info: {
 UIImagePickerControllerMediaType = "public.movie";
 UIImagePickerControllerMediaURL = file:///localhost/private/var/mobile/Applications/A105DD21-CC89-4A0C-A435-25C6DC3E3C14/tmp/capture-T0x104520.tmp.zYgOEt/capturedvideo.MOV;
}
```

**F** Console output after recording a movie with the iPhone camera.

# Playing Video

You can add support for playing video by using the **MPMoviePlayerController** class; the process is similar to playing audio.

Movies either can be a local file or can be streamed over HTTP from a remote source. The movie controller will play all the iPhone-supported formats, including MOV, 3GP, MPV, and MP4. You can also play MP3 and AAC audio files; the movie controller will display the QuickTime logo on a white background in place of the movie.

The area in which a movie is played is determined by the **view** property of the movie controller. Just as with any other view, you can manipulate its frame and incorporate it into your view hierarchies. You can even overlay subviews on top of the movie and set custom background content via the **backgroundView** property.

By setting the **scalingMode** property of the movie, you can control how a movie that is wider or higher than the iPhone screen is displayed. This process is similar to how scaling works with the **UIImageView** class; you can have the movie either fully fill the display or scale by width or height and show a border for the other dimension.

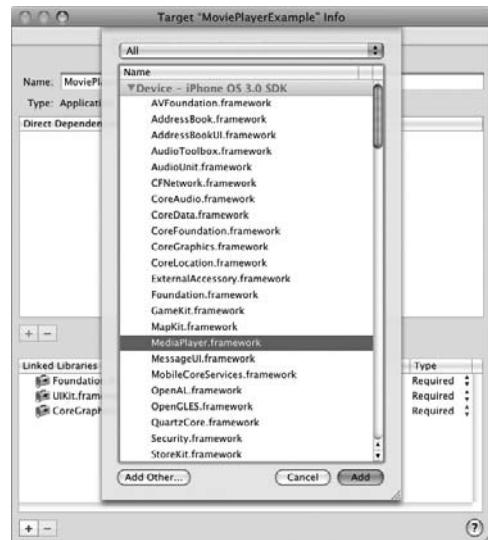
You have full control of the movie playback, including things such as starting and stopping, seeking forward and backward through the movie, and setting which controls appear onscreen to manipulate the playback of the movie. You can even choose to remove these onscreen controls entirely, which may be useful when playing an intro movie to a game.

Knowing that many developers wouldn't require this level of control, Apple created the **MPMoviePlayerViewController** class. This provides a simple view controller for displaying movies full-screen and is very easy to implement in your applications.

## To create an application to play a movie:

1. Create a new view-based application, saving it as MoviePlayerExample.
2. In the Groups & Files list, expand the targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add **MediaPlayer.framework** **A**.
4. Open the MoviePlayerExampleView Controller.h file, add the **#import** statement to import the MediaPlayer.h file, and create an instance variable to hold the movie player view controller (**Code Listing 9.16**).
5. Switch to the MoviePlayerExample ViewController.m file, uncomment the **viewDidLoad** method, and add the following:

```
UIButton *btn = [UIButton
→ buttonWithType:UIButtonTypeRoundedRect];
[btn setFrame:CGRectMake
→ (100,110,120,34)];
[btn setTitle:@"Show Movie"
→ forState:UIControlStateNormal];
[btn addTarget:self action:
→ @selector(buttonClick:
→ forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:btn];
```



**A** Adding the **MediaPlayer** framework to your application.

**Code Listing 9.16** The header file for the movie player view controller application.

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>

@interface MoviePlayerExampleViewController : UIViewController
{
 MPMoviePlayerViewController *playerViewController;
}
@end
```

Here you are simply creating a button that, when tapped, will call a method to load the movie.

6. Next implement your **buttonClick:** method:

```
NSString *filePath = [[[NSBundle
→ mainBundle] resourcePath]
→ stringByAppendingPathComponent:
→ @"movie.m4v"];

NSURL *movieURL = [NSURL
→ fileURLWithPath:filePath];
```

This code should look familiar: It's almost exactly the way you set up and play audio. You first retrieve the video file from the application bundle. (This example assumes you have added a movie named movie.m4v to your project.)

```
playerViewController =
→ [[MPMoviePlayerViewController
→ alloc]
initContentURL:movieURL];
```

You then create your movie player view controller, using the **NSURL** object you've just created.

```
[[NSNotificationCenter
→ defaultCenter] addObserver:self
→ selector:@selector
→ (moviePlayerDidFinish:
→ name:MPMoviePlayerPlayback
→ DidFinishNotification
→ object:[playerViewController
→ moviePlayer]];
```

```
[self presentMoviePlayerView
→ ControllerAnimated:playerView
→ Controller];
```

*continues on next page*

Next you set up a method that will be called when the movie finishes playing before finally presenting the movie itself. Notice you don't have to actually tell the movie to start playing—this is done automatically for you.

7. Implement the `moviePlayerDidFinish:` method:

```
MPMoviePlayerController
→ *player = [aNote object];

[[NSNotificationCenter
→ defaultCenter]
→ removeObserver:self
→ name:MPMoviePlayerPlaybackDid
→ FinishNotification
→ object:player];

[player stop];

[self dismissMoviePlayerView
→ ControllerAnimated];

[playerViewController release];
```

Here you are just cleaning up by removing yourself as an observer to the movie view controller, making sure the movie has stopped and any memory it's using is released. **Code Listing 9.17** shows the completed application.

Notice how you have a full set of onscreen controls for the movie, and you can switch between landscape and portrait modes by simply rotating the phone **B**.



**B** Playing a full-screen movie using a movie player view controller.

**Code Listing 9.17** The completed code to play a movie using a movie player view controller.

```
#import "MoviePlayerExampleViewController.h"

@implementation MoviePlayerExampleViewController

-(void)moviePlayerDidFinish:(NSNotification *)aNote {
 MPMoviePlayerController *player = [aNote object];
 [[NSNotificationCenter defaultCenter] removeObserver:self
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:player];
 [player stop];
 [self dismissMoviePlayerViewControllerAnimated];
 [playerViewController release];
}

-(void)buttonClick:(id)sender {
 NSString *filePath = [[[NSBundle mainBundle] resourcePath]
 stringByAppendingPathComponent:@"movie.m4v"];
 NSURL *movieURL = [NSURL fileURLWithPath:filePath];
 playerViewController = [[MPMoviePlayerViewController alloc] initWithContentURL:movieURL];
 [[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(moviePlayerDidFinish:)
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:[playerViewController moviePlayer]];
 [self presentMoviePlayerViewControllerAnimated:playerViewController];
}

-(void)viewDidLoad {
 [super viewDidLoad];
 UIButton *btn = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [btn setFrame:CGRectMake(100,110,120,34)];
 [btn setTitle:@"Show Movie" forState:UIControlStateNormal];
 [btn addTarget:self action:@selector(buttonClick:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:btn];
}

@end
```

## To gain more control over movie playback

You've seen how easy it is to play movies by using instances of the **MPMoviePlayerViewController**, but sometimes you may want to have more control over the appearance of your movies. As previously mentioned, you can gain a lot more control of the visual elements of your movies by using an instance of the **MPMoviePlayerController** class.

If you have a large movie or are streaming a movie across a network, it may take some time before the movie is ready to be played. Movie players handle this situation by preloading some of the movie into a buffer and then sending a notification once the movie has loaded enough to be played. By responding to this notification, you can display a progress indicator to the user that disappears only when the movie is actually ready to be played.

Now you'll see how to use these notifications to display an indicator that shows before the movie begins and then hides itself once the movie is ready to play. You'll load the movie from a network location to see how HTTP streaming works. You'll also learn how to manipulate some of the properties of the movie view to create a more custom experience.

## To update the application to use an MPMoviePlayerController:

1. Open MoviePlayerExampleView Controller.h, and create a new instance variable to hold your activity indicator (**Code Listing 9.18**)
2. Switch to MoviePlayerExampleView Controller.m, and modify the `viewDidLoad` method:

```
CGRect activityFrame =
→ CGRectMake(130,10,40,40);

activityView = [[UIActivityIndicatorView
→ alloc] initWithFrame:
→ activityFrame:
→ UIActivityIndicatorViewStyleGray];

[activityView setActivityIndicatorViewStyle:
→ UIActivityIndicatorViewStyleGray];

[self.view addSubview:
→ activityView];
```

Here you are creating and adding an activity indicator that you'll use to indicate that the movie has not loaded.

*continues on next page*

**Code Listing 9.18** The header file for the updated movie player application.

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>

@interface MoviePlayerExample2ViewController : UIViewController
{
 MPMoviePlayerController *playerController;
 UIActivityIndicatorView *activityView;
}

@end
```

3. Update the `buttonClick:` method:

```
NSURL *movieURL = [NSURL
 URLWithString:@"http://images.
 → apple.com/movies/us/hd_
 → gallery/gl1800/480p/mammoth_
 → 480pr.mov"];

playerController = [[MPMovie
 → PlayerController alloc]
 → initWithContentURL:movieURL];

playerController.should
→ Autoplay = NO;

playerController.initial
→ PlaybackTime = 5.0;

playerController.controlStyle =
→ MPMovieControlStyleEmbedded;
```

This time you are loading your movie from the Internet rather than using a local movie. Next you create your movie player in the same way as the earlier example, except now you are using an `MPMoviePlayerController` instance.

4. You then set some properties that control whether the movie starts playing automatically, determine the start position (in this example, five seconds in), and indicate the type of movie playing controls you want to show.

```
playerController.backgroundView.
→ backgroundColor = [UIColor
 → yellowColor];

CGRect playerFrame =
→ CGRectMakeInset(self.view.bounds,
 → 50.0, 50.0);

playerController.view.frame =
→ playerFrame;

[self.view addSubview:player
→ Controller.view];
```

Next you set the background color of your movie—the area of your **MPMoviePlayerController** that does not contain the playing movie. You then set the actual frame of the movie player controller and manually add it to your main view as you would any other **UIView**. Note that although the movie will not begin loading itself from the network at this point, it will not actually show until it's ready to play, either.

5. Finally, you subscribe to the same notification as earlier, in addition to a new one: **MPMoviePlayerLoadStateDidChangeNotification** is called whenever the network buffering state of the movie changes. This is indicated by enabling animation for the activity indicator.
6. Implement the **loadStateChange:** method:

```
MPMoviePlayerController *player =
→ [aNote object];
```

```
if(player.loadState ==
→ MPMovieLoadStatePlayable ||
→ MPMovieLoadStatePlaythroughOK)
[activityView stopAnimating];
else
[activityView startAnimating];
```

This method will be called whenever your movie changes its network state. By examining the **loadState** property, you can determine whether the movie is ready to be played and show or hide your activity indicator.

Code Listing 9.19 shows the completed code.

*continues on next page*

## 7. Build and run the application.

This time the movie does not load right away but instead shows an activity indicator while it is downloaded from the network. Once enough of the movie has downloaded, the activity indicator disappears, and you can tap the play button to start the movie. The activity indicator will automatically appear again if the movie has difficulty streaming from the network.

Note how the movie shows a different set of controls from the earlier example C. You can still make the movie go full-screen by tapping the resize button. Also note how the movie is locked into portrait mode and doesn't automatically switch to landscape when the iPhone is rotated.

You can read more about working with video by referring to the *MPMoviePlayerController* and *MPMoviePlayerView-Controller* class references in the iPhone developer documentation.



C Playing a movie in a custom movie player.

**Code Listing 9.19** The updated movie player code loading a movie from a network location and responding to notifications.

```
#import "MoviePlayerExample2ViewController.h"

@implementation MoviePlayerExample2ViewController

-(void)moviePlayerDidFinish:(NSNotification*)aNote {
 MPMoviePlayerController *player = [aNote object];
 [[NSNotificationCenter defaultCenter] removeObserver:self
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:player];
 [player stop];
 [activityView stopAnimating];

 [playerController.view removeFromSuperview];
 [playerController release];
}
```

code continues on next page

**Code Listing 9.19** continued

```
-loadStateChange:(NSNotification *)aNote
{
 MPMoviePlayerController *player = [aNote object];

 if(player.loadState == MPMovieLoadStatePlayable || MPMovieLoadStatePlaythroughOK)
 [activityView stopAnimating];
 else
 [activityView startAnimating];
}

-(void)buttonClick:(id)sender {

 NSURL *movieURL = [NSURL
 URLWithString:@"http://images.apple.com/movies/us/hd_gallery/g1800/480p/mammoth_480pr.mov"];

 playerController = [[MPMoviePlayerController alloc] initWithContentURL:movieURL];
 playerController.shouldAutoplay = NO;
 playerController.initialPlaybackTime = 5.0;
 playerController.controlStyle = MPMovieControlStyleEmbedded;

 playerController.backgroundView.backgroundColor = [UIColor yellowColor];

 CGRect playerFrame = CGRectMakeInset(self.view.bounds, 50.0, 50.0);
 playerController.view.frame = playerFrame;
 [self.view addSubview:playerController.view];

 [[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(moviePlayerDidFinish:)
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:playerController];

 [[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(loadStateChange:)
 name:MPMoviePlayerLoadStateDidChangeNotification
 object:playerController];
}

[activityView startAnimating];
}

-(void)viewDidLoad {

 [super viewDidLoad];

 UIButton *btn = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [btn setFrame:CGRectMake(100,110,120,34)];
 [btn setTitle:@"Show Movie" forState:UIControlStateNormal];
 [btn addTarget:self action:@selector(buttonClick:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:btn];

 CGRect activityFrame = CGRectMake(130,10,40,40);
 activityView = [[UIActivityIndicatorView alloc]
 initWithFrame:activityFrame];
 [activityView setActivityIndicatorViewStyle:UIActivityIndicatorViewStyleGray];
 [activityView setFrame:activityFrame];
 [self.view addSubview:activityView];
}

@end
```

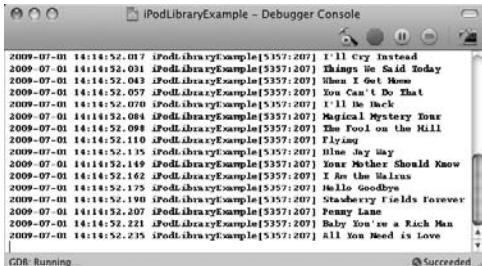
# Using the iPod Library

You've already seen how to play movies using the Media Player framework. This framework also lets you access the iPhone's iPod library from within your own applications. You can play all the library's audio content, including music, podcasts, and audio books. You can also get access to all the playlists in the iPod library, including genius, on-the-go, and smart playlists, as well as those created in iTunes. You will need to run the examples in this section on an iPhone because the simulator has no iPod library.

## Accessing media items

Every song, podcast, and audio book in the iPod library is known as a *media item*, and each media item is an instance of the **MPMediaItem** class, which represents a single piece of media. Each media item contains a unique, persistent identifier along with a set of other metadata containing information such as the song title, artist, genre, and so on.

To retrieve media items from the iPod library, you create a *media query*, specifying filter criteria based on the items you are trying to retrieve. You can optionally specify a *media grouping*, which will group and sort the returned media items by a common property such as artist, playlist, or genre. Media queries offer a number of predefined initializer methods that prepopulate the grouping type property for you.



A screenshot of a Mac OS X terminal window titled "iPodLibraryExample - Debugger Console". The window shows a list of log entries from July 1, 2009, at 14:14:52.017. The entries are as follows:

```
2009-07-01 14:14:52.017 iPodLibraryExample[5357:207] I'll Cry Instead
2009-07-01 14:14:52.031 iPodLibraryExample[5357:207] Things We Said Today
2009-07-01 14:14:52.035 iPodLibraryExample[5357:207] When I Get Home
2009-07-01 14:14:52.037 iPodLibraryExample[5357:207] I'm Gonna Make That
2009-07-01 14:14:52.070 iPodLibraryExample[5357:207] I'll Be Back
2009-07-01 14:14:52.084 iPodLibraryExample[5357:207] Magical Mystery Tour
2009-07-01 14:14:52.093 iPodLibraryExample[5357:207] The Fool on the Hill
2009-07-01 14:14:52.110 iPodLibraryExample[5357:207] Flying
2009-07-01 14:14:52.115 iPodLibraryExample[5357:207] Nine, Jay Way
2009-07-01 14:14:52.116 iPodLibraryExample[5357:207] You Should Know
2009-07-01 14:14:52.140 iPodLibraryExample[5357:207] Be the Nexus
2009-07-01 14:14:52.175 iPodLibraryExample[5357:207] Hello Goodbye
2009-07-01 14:14:52.198 iPodLibraryExample[5357:207] Strawberry Fields Forever
2009-07-01 14:14:52.207 iPodLibraryExample[5357:207] Penny Lane
2009-07-01 14:14:52.221 iPodLibraryExample[5357:207] Baby You're a Rich Man
2009-07-01 14:14:52.215 iPodLibraryExample[5357:207] All You Need is Love
```

**A** Sample console output of a media query.

## To create an application to retrieve media items from the iPod library:

1. Create a new view-based application, saving it as iPodLibraryExample.
2. In the Groups & Files list, expand the targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add **AVMediaPlayer.framework**.
4. Open the iPodLibraryExample.h file, and add the **#import** statement to import the MediaPlayer.h file.
5. Switch to the iPodLibraryExample.m file, uncomment the **viewDidLoad** method, and add the following code:

```
MPMediaQuery *query =
→ [MPMediaQueryartistsQuery];
NSArray *results = [query items];
for (MPMediaItem *m in results)
NSLog(@"%@",[m valueForProperty:
→ MPMediaItemPropertyTitle]);
```

This creates a media query for all media items, grouping by artist, and stores the media items in the variable **results**. You then loop through the array, using the **valueForProperty:** method to log the title of each media item to the console **A**.

*continues on next page*

To update the code to add a filter for a particular artist, add the following:

```
MPMediaQuery *query =
→ [MPMediaQueryartistsQuery];
MPMediaPropertyPredicate *pred =
→ [MPMediaPropertyPredicate
→ predicateWithValue:@"Dylan"
→ forProperty:MPMediaItemProperty
→ Artist
→ comparisonType:MPMediaPredicate
→ ComparisonContains];

[query addFilterPredicate:pred];
NSArray *results = [query items];
for (MPMediaItem *m in results)
NSLog(@"%@",[m valueForProperty:
→ MPMediaItemPropertyTitle]);
```

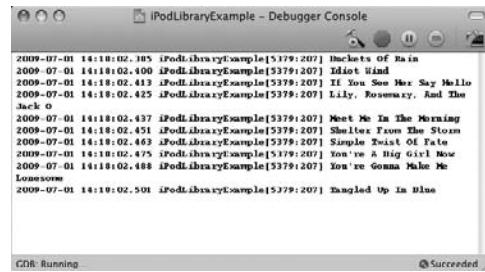
This is the same query but with a filter that returns only those media items with the word *Dylan* in their **Artist** property. You can add other filter criteria the same way. Notice the resulting console output **B**.

**TIP** Creating a media query with no filter or grouping will return all media items from the iPod library.

**TIP** Refer to the *MPMediaItem Class Reference* in the iPhone developer documentation for a complete list of metadata properties available for media items.

## Accessing media collections

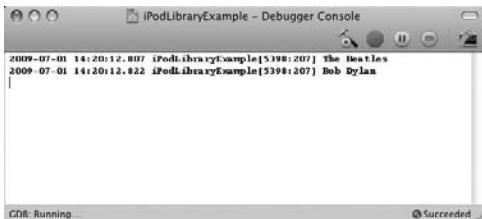
You can also use media queries to retrieve *collections* of media items grouped by a particular property, such as genre, album, artist, or playlist. This time you use the **collections** property of the media query.



The screenshot shows the Xcode debugger's "Console" tab with the title "iPodLibraryExample - Debugger Console". The log output displays a list of songs from an iPod library, filtered by the artist "Dylan". The log entries show the date (2009-07-01), time (14:18:02), file (iPodLibraryExample.m), line number (207), song title, and lyrics. The lyrics for each song are partially visible. At the bottom of the console window, there are status bars for "GDB: Running" and "Succeeded".

```
2009-07-01 14:18:02.105 iPodLibraryExample[5379:207] Hackets Of Rain
2009-07-01 14:18:02.400 iPodLibraryExample[5379:207] Idiot Wind
2009-07-01 14:18:02.413 iPodLibraryExample[5379:207] If You See Her Say Hello
2009-07-01 14:18:02.425 iPodLibraryExample[5379:207] Lily, Rosemary, And The Jack O
2009-07-01 14:18:02.437 iPodLibraryExample[5379:207] Meet Me In The Morning
2009-07-01 14:18:02.451 iPodLibraryExample[5379:207] Shelter From The Storm
2009-07-01 14:18:02.463 iPodLibraryExample[5379:207] Simple Twist Of Fate
2009-07-01 14:18:02.475 iPodLibraryExample[5379:207] You're A Big Girl Now
2009-07-01 14:18:02.488 iPodLibraryExample[5379:207] You're Gonna Make Me Lonesome
2009-07-01 14:18:02.501 iPodLibraryExample[5379:207] Tangled Up In Blue
```

**B** Sample console output after filtering the media query.

A screenshot of the Xcode debugger console window titled "iPodLibraryExample - Debugger Console". The console shows two lines of log output: "2009-07-01 14:20:12.807 iPodLibraryExample[5398:207] The Beatles" and "2009-07-01 14:20:12.822 iPodLibraryExample[5398:207] Bob Dylan". Below the console, a status bar indicates "GDB: Running..." and "Succeeded".

C Sample console output of a media query collection.

To use collections, modify the `viewDidLoad` method to look like the following:

```
MPMediaQuery *query = [MPMediaQuery
→ artistsQuery];

NSArray *results = [query
→ collections];

for (MPMediaItemCollection *m in
→ results)
{
 MPMediaItem *item =
→ [m representativeItem];

 NSLog(@"%@",[item valueForProperty:
→ MPMediaItemPropertyArtist]);
}
```

This time the array contains `MPMediaItemCollection` objects. From this you can get the *representative item*, a media item whose properties represent the common properties of all media items in the collection. Since you are retrieving by artist, you know that all media items in the collection item will have the same artist. You then log the result to the console as you did earlier C. Notice how there are only two results since you had only two artists in your media query.

You can also use collections to retrieve playlists:

```
MPMediaQuery *query = [MPMediaQuery
→ playlistsQuery];

NSArray *results = [query
→ collections];

for (MPMediaPlaylist *m in results)
 NSLog(@"%@",[m valueForProperty:
→ MPMediaPlaylistPropertyName]);
```

This time, you specify `playlistsQuery` as the initializer for the media query to return objects of type `MPMediaPlaylist` in the `collections` property, which you can loop through as before.

## Using the media picker

If you don't want or need to go through the trouble of programmatically querying for media items, you can instead use the **MPMediaPickerController** view controller class. This provides you with a graphical interface similar to the iPod's own media selection screen, allowing the user to pick media items just as they would with the iPod application. Once the user taps the Done button, the media picker controller returns a collection of media items to your application.

### To add a media picker to your application:

1. Open the iPodLibraryExample.h file, and add the declaration to implement the **AVAudioPlayerDelegate** protocol.
2. Switch to the iPodLibraryExample.m file, and edit the **viewDidLoad** method to create a button to launch the media picker:

```
UIButton *btnMedia = [UIButton
→ buttonWithType:UIButtonTypeRoundedRect];

[btnMedia setFrame:CGRectMake
→ (230.0, 410.0, 80.0, 34.0)];

[btnMedia setTitle:@"Media"
→ forState:UIControlStateNormal];

[btnMedia addTarget:self action:
→ @selector(mediaPickerAction:)
→ forControlEvents:UIControlEventTouchUpInside];

[self.view addSubview:btnMedia];
```



D The application with the media picker being used to select songs from the iPod library.

3. Implement the `mediaPickerController:` method that is called when the button is tapped:

```
MPMediaPickerController *picker =
→ [[MPMediaPickerController alloc]
→ initWithMediaTypes:MPMediaType
→ Any];

picker.delegate = self;

picker.allowsPickingMultipleItems =
→ YES;

picker.prompt = @"Please select";
[self presentModalViewController:
→ picker animated:YES];

[picker release];
```

This initializes the media picker to show all media types. You can also filter the content in the picker to allow only music, podcasts, or audio books. Setting `allowsPickingMultipleItems` lets the user select more than a single media item. You also set the title of the media picker.

4. Implement the `mediaPickerController:didPickMediaItems:` and `mediaPickerControllerDidCancel:` delegate methods:

```
[self dismissModalViewControllerAnimated:
→ YES];
```

This simply closes the media picker and logs any selected media items to the console D.

## Playing media

The **MPMusicPlayerController** class lets you play media items. Two types of music players are available:

- **The application music player**—This plays audio in your application only. When your application ends, so does the audio. Using an application music player *pauses* any audio that happens to be playing on the iPod when your application was launched.
- **The iPod music player**—This uses the iPhone’s iPod application to play the audio. When your application ends, the audio will continue to play, and the normal iPod functions control the playback. Using the iPod music player *replaces* any audio that happens to be playing on the iPod when your application was launched.

Both provide controls for playing, pausing, and stopping, as well as more powerful functions for fast-forwarding, rewinding, and skipping to the next, previous, first, and last songs. You can also access information about the song currently being played, allowing you to update your application interface accordingly.

To play audio, you pass the music player a set of one or more media items known as a *playback queue*. By using a playback queue, you can construct an ad hoc list of audio to play that doesn’t necessarily relate to any playlists or audio grouping in the iPod library.



E The completed media player application.

## To play audio from the iPod library:

1. Open the iPodLibraryExample.h file, and create an instance variable to hold the music player:

```
MPMusicPlayerController *player;
```

2. Switch to the iPodLibraryExample.m file, and edit the `viewDidLoad` method to create the music player:

```
MPMediaQuery *query =
→ [MPMediaQueryartistsQuery];
player = [MPMusicPlayerController
→ applicationMusicPlayer];
[player setQueueWithQuery:query];
player.play;
```

This code creates a media query that's grouped and sorted by artist. It then creates the application music player and passes the query to it before telling it to start playing.

Using the `setQueueWithQuery:` method is a convenient way to load a media player with media query results, but you might want to create a custom collection of music to play.

Now you'll update the application to create a simple media player, add buttons to control the previous and next songs, and add a slider to control the song volume. You'll also add some labels to display the current album artwork, artist name, song title, and track number. The media picker control will create the playlist for the application E.

## To update the application to create a media player:

1. Open the iPodLibraryExample.h file, and declare some new variables used to display the currently playing song (**Code Listing 9.20**).
2. Switch to the iPodLibraryExample.m file, and modify the **viewDidLoad**:

```
songCount = 0;
songNumber = 0;
[self createCurrentTrackDisplay];
[self createVolumeControl];
[self createPrevNextButtons];
[self createMediaPickerButton];
```

After initializing the variables you'll be using to display the current track, you call methods to build the application's UI:

The **createCurrentTrackDisplay** method adds an image view and labels to display information about the current track.

The **createVolumeControl** method adds a volume control to the interface. This uses the **MPVolumeView** class, which is a special view that is designed to control your music player's volume. You don't need to write any code other than adding this control to your interface; the slider is hooked up to your music player automatically.

**Code Listing 9.20** The media player header file.

```
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>

@interface iPodLibraryExampleViewController : UIViewController <MPMediaPickerControllerDelegate>
{
 MPMusicPlayerController *player;
 int songNumber, songCount;
 UIImageView *currentArtwork;
 UILabel *currentArtist;
 UILabel *currentSong;
 UILabel *trackCount;
}
@end
```

The `createPrevNextButtons` method creates buttons that allow you to navigate forward and backward through the selected audio items. Notice they use the same action method, `prevNextAction:`. This method uses the `tag` property of the button to determine which button has been tapped. After checking that the action can be performed, it moves to the next or previous track and updates the `songNumber` variable used to display the current track.

3. Next assign the application player to a local variable (for convenience), and then register for the `MPMusicPlayerControllerNowPlayingItemDidChangeNotification` event:

```
player = [MPMusicPlayerController
→ applicationMusicPlayer];
```

```
[[NSNotificationCenter
→ defaultCenter] addObserver:self
→ selector:@selector(songChanged:
→ name:MPMusicPlayerControllerNow
→ PlayingItemDidChangeNotification
→ object:nil];
```

```
player.beginGeneratingPlayback
→ Notifications;
```

The `songChanged:` method updates the interface each time a new media item begins playing. Calling `beginGeneratingPlaybackNotifications` tells the music player to post notifications whenever the media item changes.

*continues on next page*

4. Finally implement the code for the `mediaPickerController:didPickMediaItems:` delegate method:

```
[player setQueueWithItem
→ Collection:mediaItemCollection];
```

```
songCount = [[mediaItemCollection
→ items] count];
```

```
songNumber = 0;
```

```
if (songCount > 0)
 [player play];
```

This assigns the media collection the user has just picked to the music player using the `setQueueWithItemCollection:` method, resets the track count variables, and tells the music player to begin playing the first track. **Code Listing 9.21** shows the completed code.

**Code Listing 9.21** The completed iPod library player application.

```
#import "iPodLibraryExampleViewController.h"

@implementation iPodLibraryExampleViewController

-(void)showPicker:(id)sender {
 MPMediaPickerController *picker = [[MPMediaPickerController alloc]
 initWithMediaTypes:MPMediaTypeAny];
 picker.delegate = self;
 picker.allowsPickingMultipleItems = YES;
 picker.prompt = @"Please select";
 [self presentViewController:picker animated:YES];
 [picker release];
}

-(void)createCurrentTrackDisplay {
 currentArtwork = [[UIImageView alloc] init];
 [currentArtwork setFrame:CGRectMake(10.0,10.0,180.0,180.0)];
 currentArtwork.backgroundColor = [UIColor whiteColor];
 [self.view addSubview:currentArtwork];
 [currentArtwork release];

 currentArtist = [[UILabel alloc] init];
 [currentArtist setFrame:CGRectMake(10.0,200.0,300.0,24.0)];
 currentArtist.backgroundColor = [UIColor clearColor];
 [self.view addSubview:currentArtist];
 [currentArtist release];
```

*code continues on next page*

**Code Listing 9.21** *continued*

```
currentSong = [[UILabel alloc] init];
[currentSong setFrame:CGRectMake(10.0,230.0,300.0,24.0)];
currentSong.backgroundColor = [UIColor clearColor];
[self.view addSubview:currentSong];
[currentSong release];

trackCount = [[UILabel alloc] init];
[trackCount setFrame:CGRectMake(10.0,260.0,300.0,24.0)];
trackCount.backgroundColor = [UIColor clearColor];
[self.view addSubview:trackCount];
[trackCount release];
}

-(void)createVolumeControl {
 MPVolumeView *volumeView = [[MPVolumeView alloc] init];
 [volumeView setFrame:CGRectMake(10.0,360.0,300.0,34.0)];
 [self.view addSubview:volumeView];
 [volumeView release];
}

-(void)prevNextAction:(id)sender {
 //prev/next?
 if ([sender tag] == 0 && songNumber > 0)
 {
 player.skipToPreviousItem;
 songNumber--;
 }

 if ([sender tag] == 1 && songNumber < songCount-1)
 {
 player.skipToNextItem;
 songNumber++;
 }
}

-(void)mediaPickerController:(id)sender {
 MPMediaPickerController *picker = [[MPMediaPickerController alloc] initWithMediaTypes:MPMediaTypeAny];
 picker.delegate = self;
 picker.allowsPickingMultipleItems = YES;
 picker.prompt = @"Select tracks";
 [self presentViewController:picker animated:YES];
 [picker release];
}

-(void)createPrevNextButtons {
 UIButton *btnPrev = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [btnPrev setFrame:CGRectMake(10.0,410.0,45.0,34.0)];
 [btnPrev setTag:0];
 [btnPrev setTitle:@"<" forState:UIControlStateNormal];
 [btnPrev addTarget:self action:@selector(prevNextAction:)];
 [btnPrev forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:btnPrev];
}

UIButton *btnNext = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[btnNext setFrame:CGRectMake(60.0,410.0,45.0,34.0)];
[btnNext setTag:1];
[btnNext setTitle:@">" forState:UIControlStateNormal];
[btnNext addTarget:self action:@selector(prevNextAction:)];
[btnNext forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:btnNext];
}

-(void)createMediaPickerButton {
```

*code continues on next page*

**Code Listing 9.21** *continued*

```
UIButton *btnMedia = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[btnMedia setFrame:CGRectMake(230.0,410.0,80.0,34.0)];
[btnMedia setTitle:@"Media" forState:UIControlStateNormal];
[btnMedia addTarget:self action:@selector(mediaPickerAction:)
 forControlEvents:UIControlEventTouchUpInside];

[self.view addSubview:btnMedia];
}

-(void)mediaPicker:(MPMediaPickerController *)mediaPicker didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection
{
 [self dismissModalViewControllerAnimated:YES];

 [player setQueueWithItemCollection:mediaItemCollection];

 songCount = [[mediaItemCollection items] count];
 songNumber = 0;

 if (songCount > 0)
 [player play];
}

-(void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker
{
 [self dismissModalViewControllerAnimated:YES];
}

-(void)songChanged:(NSNotification*)aNotification {
 MPMediaItem *item = [player nowPlayingItem];
 currentArtist.text = [item valueForProperty:MPMediaItemPropertyArtist];
 currentSong.text = [item valueForProperty:MPMediaItemPropertyTitle];
 trackCount.text = [NSString stringWithFormat:@"Track: %d/%d",songNumber+1,songCount];

 MPMediaItemArtwork *artwork = [item valueForProperty:MPMediaItemPropertyArtwork];
 CGSize aSize = currentArtwork.frame.size;
 [currentArtwork setImage:[artwork imageWithSize:aSize]];
}

-(void)viewDidLoad {
 songCount = 0;
 songNumber = 0;

 [self createCurrentTrackDisplay];
 [self createVolumeControl];
 [self createPrevNextButtons];
 [self createMediaPickerButton];

 player = [MPMusicPlayerController applicationMusicPlayer];

 [[NSNotificationCenter defaultCenter] addObserver:self
 selector:@selector(songChanged:)
 name:MPMusicPlayerControllerNowPlayingItemDidChangeNotification
 object:nil];

 player.beginGeneratingPlaybackNotifications;
}

-(void)dealloc {
 [[NSNotificationCenter defaultCenter] removeObserver:self
 name:MPMusicPlayerControllerNowPlayingItemDidChangeNotification
 object:nil];
 [super dealloc];
}

@end
```

# 10

# Contacts, Calendars, E-mail, and SMS

One of the great aspects of the iPhone is how the built-in applications can share data and work with each other. For example, when making a phone call in the Phone application, you select from a list of contacts created in the Contacts application.

In this chapter, you'll learn how to use the APIs for interacting with some of the iPhone's built-in applications: the Contacts, Calendar, Mail, and SMS applications.

You'll first learn how contact information is stored and how you can go about retrieving, creating, editing, and saving contacts in your own code. You'll also see how you can replicate much of the user interface functionality of the Contacts application from within your own applications.

Next you'll learn how you work with the calendar database and how to query for specific events, as well as create your own. You'll see how easy it is to create a user interface for working with events by using the built-in view controller classes. Finally, you'll learn how to compose and send an e-mail and SMS and how the iPhone SDK makes adding this functionality to your applications almost trivial.

---

## In This Chapter

|                               |     |
|-------------------------------|-----|
| Working with the Address Book | 406 |
| Adding a User Interface       | 418 |
| The iPhone Calendar           | 428 |
| E-mail                        | 443 |
| SMS                           | 450 |

---

# Working with the Address Book

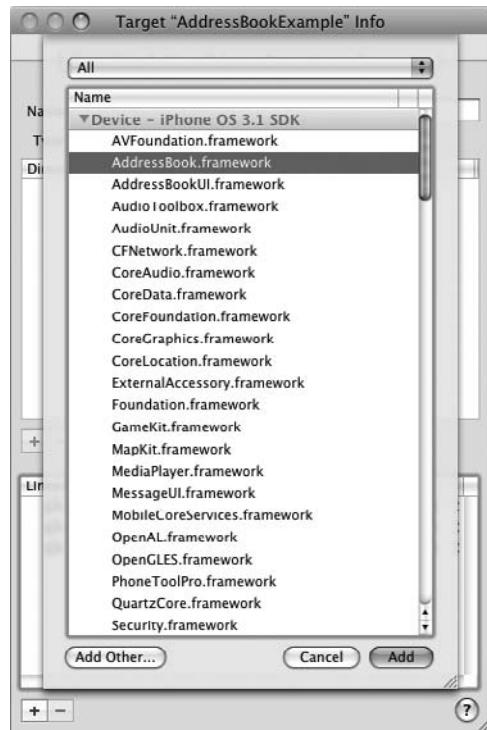
The Address Book is a system-wide database that stores all the information relating to contacts on the iPhone. Each record in the database is an instance of an **ABRecordRef**.

The Address Book can store two kinds of records:

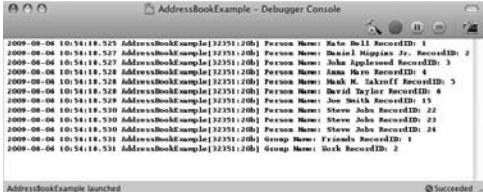
- **Person records**—Represented by the **ABPerson** type, these contain all the data relating to contacts such as names, phone numbers, and address information.
- **Group records**—Represented by the **ABGroup** type, these are used to organize contacts into groups. Unlike person records, group records have only a single property, which represents their name.

## To retrieve all records from the Address Book:

1. Create a new view-based application, saving it as AddressBookExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add AddressBook.framework A.
4. Open AddressBookExampleViewController.h, and import the AddressBook.h header:  
`#import <AddressBook/AddressBook.h>`



A Adding the Address Book framework.



The screenshot shows the Xcode debugger console window titled "AddressBookExample - Debugger Console". It displays a series of log messages in a monospaced font. The log entries are timestamped at 2009-08-04 10:54:18.525 and show various address book records being logged. The log ends with the message "AddressBookExample launched" and a status indicator "Succeeded".

```
2009-08-04 10:54:18.525 AddressBookExample[32351:20b] Person Name: Kate Bell RecordID: 1
2009-08-04 10:54:18.527 AddressBookExample[32351:20b] Person Name: Daniel Hippius Jr. RecordID: 2
2009-08-04 10:54:18.527 AddressBookExample[32351:20b] Person Name: John Appleseed RecordID: 3
2009-08-04 10:54:18.528 AddressBookExample[32351:20b] Person Name: Steve Jobs RecordID: 4
2009-08-04 10:54:18.528 AddressBookExample[32351:20b] Person Name: David Taylor RecordID: 5
2009-08-04 10:54:18.528 AddressBookExample[32351:20b] Person Name: Mark H. Tolkenoff RecordID: 6
2009-08-04 10:54:18.528 AddressBookExample[32351:20b] Person Name: Steve Jobs RecordID: 22
2009-08-04 10:54:18.529 AddressBookExample[32351:20b] Person Name: Steve Jobs RecordID: 23
2009-08-04 10:54:18.529 AddressBookExample[32351:20b] Person Name: Steve Jobs RecordID: 24
2009-08-04 10:54:18.531 AddressBookExample[32351:20b] Group Name: Friends RecordID: 1
2009-08-04 10:54:18.531 AddressBookExample[32351:20b] Group Name: Stark RecordID: 2
```

- B Logging Address Book records to the console.

5. Switch to AddressBookExampleView Controller.m, uncomment the **viewDidLoad** method, and add the following code:

```
ABAddressBookRef addressBook =
→ ABAddressBookCreate();

CFArrayRef allPeople = ABAddress
→ BookCopyArrayOfAllPeople
→ (addressBook);

CFArrayRef allGroups = ABAddress
→ BookCopyArrayOfAllGroups
→ (addressBook);

for (id person in (NSArray *)
→ allPeople)
 [self logContact:person];
for (id group in (NSArray *)
→ allGroups)
 [self logGroup:group];
CFRelease(allGroups);
CFRelease(allPeople);
CFRelease(addressBook);
```

Here you first get a reference to the Address Book using the **ABAddress BookCreate** function. Then, using the **ABAddressBook CopyArrayOfAllPeople** and **ABAddress BookCopyArrayOfAllGroups** functions, you retrieve all the contacts and groups in the Address Book. Finally, you loop through each array, sending each record to a method that logs them to the console B.

*continues on next page*

**6.** Implement the `logPerson:` method:

```
CFStringRef name = ABRecordCopy
→ CompositeName(person);

ABRecordID recId =
→ ABRecordGetRecordID(person);

NSLog(@"Person Name: %@ RecordID:
→ %d",name, recId);
```

Here you log the contact name and record ID to the console.

**7.** Similarly, implement the `logGroup:` method:

```
CFStringRef name = ABRecordCopy
→ Value(group,kABGroupName
→ Property);

ABRecordID recId =
→ ABRecordGetRecordID(group);

NSLog(@"Group Name: %@ RecordID:
→ %d",name, recId);
```

Here you log the group name and record ID to the console.

**8.** Build and run your application.

Code Listing 10.1 shows the completed code.

**TIP** Notice in the example you used the `ABRecordCopyCompositeName` function to retrieve a contact's name. Although you could have also retrieved their first and last names, this function is preferred since it takes into account not only the contact's prefix (for example, "Dr." or "Mr.") but also the iPhone's localization settings (for example, in which order to display first and last names).

**TIP** The record ID of an Address Book record is unique and will remain with the record until it is deleted. However, if you sync your contacts with the MobileMe service and the sync data is reset, then the record ID for a record may change. For this reason, Apple recommends using a combination of the record ID and the contact name if you need to store a persistent reference to a contact.

**TIP** In the previous exercise, you retrieved all the contact records in the Address Book. You could also have retrieved contacts by name by using the `ABAddressBookCopyPeopleWithName` function or retrieved an individual contact by using the `ABAddressBookGetPersonWithRecordID` function.

**Code Listing 10.1** The completed code to loop through all Address Book records and log them to the console.

```
#import "AddressBookExampleViewController.h"

@implementation AddressBookExampleViewController

- (void)logPerson:(ABRecordRef)person
{
 CFStringRef name = ABRecordCopyCompositeName(person);
 ABRecordID recId = ABRecordGetRecordID(person);
 NSLog(@"Person Name: %@ RecordID: %d",name, recId);
}

- (void)logGroup:(ABRecordRef)group
{
 CFStringRef name = ABRecordCopyValue(group,kABGroupNameProperty);
 ABRecordID recId = ABRecordGetRecordID(group);
 NSLog(@"Group Name: %@ RecordID: %d",name, recId);
}

- (void)logContactsForGroup:(ABRecordRef)group
{
 CFArrayRef membersInGroup = ABGroupCopyArrayOfAllMembers(group);
 for (id person in (NSArray *)membersInGroup)
 [self logPerson:person];
 if (membersInGroup)
 CFRelease(membersInGroup);
}

- (void)viewDidLoad {
 ABAddressBookRef addressBook = ABAddressBookCreate();
 CFArrayRef allPeople = ABAddressBookCopyArrayOfAllPeople(addressBook);
 CFArrayRef allGroups = ABAddressBookCopyArrayOfAllGroups(addressBook);

 for (id person in (NSArray *)allPeople)
 [self logPerson:person];
 for (id group in (NSArray *)allGroups)
 [self logGroup:group];
 CFRelease(allGroups);
 CFRelease(allPeople);
 CFRelease(addressBook);
}

- (void)dealloc {
 [super dealloc];
}

@end
```

## Group records

You can use group records to organize contacts into related categories. The Address Book framework provides a number of functions for working with them:

- By using the **ABAddressBookGetGroupCount** function, you can find out the number of groups in the Address Book.
- To retrieve an individual group, you can use the **ABAddressBookGetGroupWithRecordID** function.
- To get an array containing all the contacts in a group, you can use the **ABGroupCopyArrayOfAllMembers** function.
- To add and delete a contact to and from a group, use the **ABGroupAddMember** and **ABGroupRemoveMember** functions.

**Code Listing 10.2** shows some examples of working with Address Book groups.

**Code Listing 10.2** Some of the Address Book group functions.

```
- (void)groupExamples
{
 ABAddressBookRef addressBook = ABAddressBookCreate();
 CFArrayRef allGroups = ABAddressBookCopyArrayOfAllGroups(addressBook);

 //log contacts in first group
 ABRecordRef firstGroup = [(NSArray *)allGroups objectAtIndex:0];
 [self logContactsForGroup:firstGroup];

 //add a contact to the group (assume contact with recordID exists)
 ABRecordID contactRecordID = 1;
 ABRecordRef contact = ABAddressBookGetPersonWithRecordID(addressBook, contactRecordID);
 CFEErrorRef error;
 ABGroupAddMember(firstGroup, contact, &error);
 ABAddressBookSave(addressBook, &error);

 //log contacts for group
 [self logContactsForGroup:firstGroup];

 //remove first contact from the group
 CFArrayRef membersInGroup = ABGroupCopyArrayOfAllMembers(firstGroup);
 ABRecordRef contactToRemove = [(NSArray *)allGroups objectAtIndex:0];
 ABGroupRemoveMember(firstGroup, contactToRemove, &error);
 ABAddressBookSave(addressBook, &error);

 //log contact for group
 [self logContactsForGroup:firstGroup];

 CFRelease(allGroups);
 CFRelease(membersInGroup);
 CFRelease(addressBook);
}
```

---

**TABLE 10.1** Single-value person properties

| Constant                                          | Description                   |
|---------------------------------------------------|-------------------------------|
| <b>kABPersonFirstNameProperty</b>                 | First name                    |
| <b>kABPersonLastNameProperty</b>                  | Last name                     |
| <b>kABPersonMiddleNameProperty</b>                | Middle name                   |
| <b>kABPersonPrefixProperty</b>                    | Name prefix<br>(e.g., “Dr.”)  |
| <b>kABPersonSuffixProperty</b>                    | Name suffix<br>(e.g., “Jr.”)  |
| <b>kABPersonNicknameProperty</b>                  | Nickname                      |
| <b>kABPersonFirstNamePhonetic<br/>→ Property</b>  | Phonetic first<br>name        |
| <b>kABPersonLastNamePhonetic<br/>→ Property</b>   | Phonetic last<br>name         |
| <b>kABPersonMiddleNamePhonetic<br/>→ Property</b> | Phonetic<br>middle name       |
| <b>kABPersonOrganization<br/>→ Property</b>       | Organization<br>name          |
| <b>kABPersonJobTitleProperty</b>                  | Job title                     |
| <b>kABPersonDepartmentProperty</b>                | Department                    |
| <b>kABPersonEmailProperty</b>                     | E-mail<br>address             |
| <b>kABPersonBirthdayProperty</b>                  | Birthday                      |
| <b>kABPersonNoteProperty</b>                      | Notes                         |
| <b>kABPersonCreationDate<br/>→ Property</b>       | Date contact<br>created       |
| <b>kABPersonModificationDate<br/>→ Property</b>   | Date contact<br>last modified |

## Person records

Person records represent contacts in the Address Book. Each contact has a number of *properties* to hold attributes such as their name, phone number, and address information. There are two kinds of properties: single-value properties and multivalue properties.

*Single-value* properties are used for simple information such as the contact’s name. To retrieve a single-value property, you use the **ABRecordCopyValue** function.

For example, to get a user’s first and last names, you would use this:

```
CFStringRef firstName =
 → ABRecordCopyValue(person,
 → kABPersonFirstNameProperty);

CFStringRef lastName =
 → ABRecordCopyValue(person,
 → kABPersonLastNameProperty);
```

Notice the second parameter contains a constant representing the property name. **Table 10.1** lists the available properties.

*Multivalue* properties are used to manage lists of values. A contact would generally have more than one phone number (for example, home, work, and cell phone numbers) and possibly multiple addresses (for work and home).

Retrieving multivalue properties is similar to retrieving single-value properties. You again use the **ABRecordCopyValue** function, only this time an **ABMultiValueRef** type is returned. You can then create an array containing a dictionary of values for each element by using the **ABMultiValue CopyArrayOfAllValues** function:

```
ABMultiValueRef addrValues =
→ ABRecordCopyValue(person,
→ kABPersonAddressProperty);

NSArray *arrAddr = [NSArray
→ arrayWithArray:(id)
→ ABMultiValueCopyArray
→ OfAllValues(addrValues)];
```

You then loop through each of these elements to get the values:

```
for (NSDictionary *addr in
→ arrAddresses)

{
 NSString *street =
→ [addr objectForKey:(NSString *)
→ kABPersonAddressStreetKey];

 NSString *city =
→ [addr objectForKey:(NSString *)
→ kABPersonAddressCityKey];

 NSString *state =
→ [addr objectForKey:(NSString *)
→ kABPersonAddressStateKey];
}
```

Now you'll look at an example of how to create a new contact and how to set values for both single and multivalue properties.

## To create a new contact:

1. Create a new view-based application, saving it as ContactExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add AddressBook.framework.
4. Open ContactExampleViewController.h, and import the AddressBook.h header:

```
#import <AddressBook/AddressBook.h>
```

5. Switch to ContactExampleViewController.m, uncomment the `viewDidLoad` method, and add the following code:

```
[self createPerson];
ABAddressBookRef addressBook =
→ ABAddressBookCreate();
CFArrayRef contacts = ABAddress
→ BookCopyPeopleWithName
→ (addressBook,CFSTR("Steve Jobs"));
ABRecordRef person = [(NSArray *
→ contacts objectAtIndex:0];
[self logAddressesForPerson:
→ person];
```

You first call a method to create a sample contact. To test whether the contact was added successfully, you then query the Address Book for contacts matching the name of the newly added one and call a method to log the address to the console.

*continues on next page*

**6.** Implement the `createPerson` method.

You first create a new person record and see some single-value properties:

```
ABRecordRef person =
```

```
→ ABPersonCreate();
```

```
CFErrorRef error = NULL;
```

```
ABRecordSetValue(person,
```

```
→ kABPersonFirstNameProperty,
```

```
→ @"Steve", &error);
```

```
ABRecordSetValue(person,
```

```
→ kABPersonLastNameProperty,
```

```
→ @"Jobs", &error);
```

Any errors in setting a property will be stored in the error variable; in this example, you don't bother checking for errors.

**7.** Create a multivalue property representing the phone numbers, and set values for the main and mobile phone numbers:

```
ABMultiValueRef phoneRef =
```

```
→ ABMultiValueCreateMutable
```

```
→ (kABMultiStringPropertyType);
```

```
ABMultiValueAddValueAndLabel
```

```
→ (phoneRef,@"555-123-432",
```

```
→ kABPersonPhoneMainLabel,NULL);
```

```
ABMultiValueAddValueAndLabel
```

```
→ (phoneRef,@"554-987-321",
```

```
→ kABPersonPhoneMobileLabel,NULL);
```

```
ABRecordSetValue(person,
```

```
→ kABPersonPhoneProperty,
```

```
→ phoneRef,&error);
```

8. Create another multivalue property, this time representing an address:

```
ABMutableMultiValueRef
→ addressRef = ABMultiValueCreate
→ Mutable(kABMultiDictionary
→ PropertyType);

NSMutableDictionary *dict =
→ [[NSMutableDictionary alloc]
→ initWithObjectsAndKeys:
→ @"1 Infinite Loop",
→ kABPersonAddressStreetKey,
→ @"Cupertino",
→ kABPersonAddressCityKey,
→ @"California",
→ kABPersonAddressStateKey,
→ @"95014",kABPersonAddressZIPKey,
→ @"USA",kABPersonAddressCountry
→ Key,nil];

ABMultiValueAddValueAndLabel
→ (addressRef,dict,kABHomeLabel,
→ NULL);

ABRecordSetValue(person,
→ kABPersonAddressProperty,
→ addressRef,&error);
```

Notice that this time you use the **kABMultiDictionaryPropertyType** type when creating the property. You then create a dictionary of key-value pairs representing the address; add this as the home address, and then save the address.

*continues on next page*

9. Add the newly created contact to the Address Book:

```
ABAddressBookRef addressBook =
→ ABAddressBookCreate();

ABAddressBookAddRecord
→ (addressBook, person, &error);

ABAddressBookSave(addressBook,
→ &error);
```

Notice the **ABAddressSave** function; this must be called in order to update the Address Book correctly. **Code Listing 10.3** shows the completed code.

10. Build and run the application.

You should see the address of the newly added contact logged to the console. If you open the Contacts application, you should see your newly added contact C.



C You can view the newly added contact in the Contacts application.

#### Code Listing 10.3 The completed code to add a new contact.

```
#import "ContactExampleViewController.h"

@implementation ContactExampleViewController

- (void)createPerson {

 ABRecordRef person = ABPersonCreate();
 CFErrorRef error = NULL;

 //single-value properties
 ABRecordSetValue(person, kABPersonFirstNameProperty, @"Steve", &error);
 ABRecordSetValue(person, kABPersonLastNameProperty, @"Jobs", &error);

 //multi-value: phone
 ABMultiValueRef phoneRef = ABMultiValueCreateMutable(kABMultiStringPropertyType);
 ABMultiValueAddValueAndLabel(phoneRef, @"555-123-432", kABPersonPhoneMainLabel, NULL);
 ABMultiValueAddValueAndLabel(phoneRef, @"554-987-321", kABPersonPhoneMobileLabel, NULL);
 ABRecordSetValue(person, kABPersonPhoneProperty, phoneRef, &error);
 CFRelease(phoneRef);

 //mutable multi-value: address
 ABMutableMultiValueRef addressRef = ABMultiValueCreateMutable(kABMultiDictionaryPropertyType);
 NSMutableDictionary *dict = [[NSMutableDictionary alloc] initWithObjectsAndKeys:
 @"1 Infinite Loop",kABPersonAddressStreetKey,
 @"Cupertino",kABPersonAddressCityKey,
 @"California",kABPersonAddressStateKey,
 @"95014",kABPersonAddressZIPKey,
 @"USA",kABPersonAddressCountryKey, nil];
 ABMultiValueAddValueAndLabel(addressRef,dict,kABHomeLabel,NULL);
 ABRecordSetValue(person, kABPersonAddressProperty, addressRef, &error);
 CFRelease(addressRef);
```

code continues on next page

### Code Listing 10.3 continued

```
ABAddressBookRef addressBook = ABAddressBookCreate();
ABAddressBookAddRecord(addressBook, person, &error);
ABAddressBookSave(addressBook, &error);

if (error != NULL)
 NSLog(@"%@",error);

CFRelease(person);
CFRelease(addressBook);
}

- (void)logAddressesForPerson:(ABRecordRef)person {

ABMultiValueRef addressValues = ABRecordCopyValue(person,kABPersonAddressProperty);
NSArray *arrAddresses = [NSArray arrayWithArray:(id)ABMultiValueCopyArrayOfAllValues(addressValues)];

for (NSDictionary *addr in arrAddresses) {
 NSLog(@"Street: %@",[addr objectForKey:(NSString *)kABPersonAddressStreetKey]);
 NSLog(@"City: %@",[addr objectForKey:(NSString *)kABPersonAddressCityKey]);
 NSLog(@"State: %@",[addr objectForKey:(NSString *)kABPersonAddressStateKey]);
 NSLog(@"ZIP: %@",[addr objectForKey:(NSString *)kABPersonAddressZIPKey]);
 NSLog(@"Country: %@",[addr objectForKey:(NSString *)kABPersonAddressCountryKey]);
}

CFRelease(addressValues);
}

- (void)viewDidLoad {

[self createPerson];

ABAddressBookRef addressBook = ABAddressBookCreate();
CFArrayRef contacts = ABAddressBookCopyPeopleWithName(addressBook,CFSTR("Steve Jobs"));
ABRecordRef person = [(NSArray *)contacts objectAtIndex:0];

[self logAddressesForPerson:person];

CFRelease(contacts);
CFRelease(addressBook);
}

- (void)dealloc {
 [super dealloc];
}

@end
```

# Adding a User Interface

So far, you've seen how to programmatically work with the Address Book framework and how it takes quite a lot of code to perform even a relatively simple task such as adding a new contact. You can imagine that replicating an interface for something like the Contacts application would be a lot of work.

Luckily, Apple thought about this and provides a second framework—the Address Book UI framework—that enables you to use all the interface elements in the Contacts application from within your own iPhone applications.

## Picking people

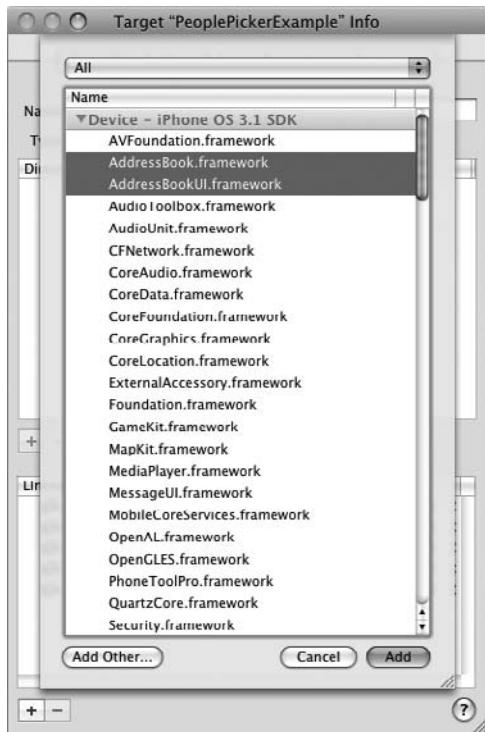
The first element of the Address Book UI framework you will look at is the people picker, represented by the `ABPeoplePickerNavigationController` class, which allows you to browse the Address Book in the same way you do in the Contacts application **A**.

### To create an application with a people picker:

1. Create a new view-based application, saving it as `PeoplePickerExample`.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add both `AddressBook.framework` and `AddressBookUI.framework` **B**.



**A** The Contacts application displaying a people picker.



- ③ Adding both Address Book frameworks to your project.

4. Open PeoplePickerExampleView Controller.h, import the AddressBookUI.h header, and add the **ABPAPeople PickerNavigationControllerDelegate** protocol declaration. You also create an instance variable to hold the people picker and create a button to launch the picker (**Code Listing 10.4**).

5. Switch to PeoplePickerExampleView Controller.m, uncomment the **viewDidLoad** method, and add the following code:

```
[self createUI];
picker = [[ABPeoplePickerController alloc]
 → init];
picker.peoplePickerController =
→ self;
```

Here you first call a method to create the user interface (a single button) and then create the people picker, setting its delegate to **self**.

6. Implement the **peoplePickerController:shouldContinueAfter SelectingPerson:** delegate method, which is called when a person is selected in the people picker:

```
[peoplePicker dismissModalViewControllerAnimated:YES];
[self logPerson:person];
```

*continues on next page*

#### Code Listing 10.4 The header file for the people picker application.

```
#import <UIKit/UIKit.h>
#import <AddressBookUI/AddressBookUI.h>

@interface PeoplePickerExampleViewController : UIViewController <ABPeoplePickerControllerDelegate>
{
 ABPeoplePickerController *picker;
 UIButton *showPicker;
}
@end
```

Here you simply close the people picker and call a method to log the selected person to the console.

## 7. Build and run the application.

You should be able to select a person and see their name and record ID being logged to the console. **Code Listing 10.5** shows the completed code.

### Code Listing 10.5 The completed people picker application.

```
#import "PeoplePickerExampleViewController.h"

@implementation PeoplePickerExampleViewController

- (void)logPerson:(ABRecordRef)person
{
 CFStringRef name = ABRecordCopyCompositeName(person);
 ABRecordID recId = ABRecordGetRecordID(person);
 NSLog(@"Person Name: %@ RecordID: %d",name, recId);
}

- (void)showPeoplePicker:(id)sender {
 [self presentModalViewController:picker animated:YES];
}

- (void)createUI {
 showPicker = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [showPicker addTarget:self action:@selector(showPeoplePicker:)];
 [showPicker forControlEvents:UIControlEventTouchUpInside];
 [showPicker setFrame:CGRectMake(190,10,120,38)];
 [showPicker setTitle:@"People Picker" forState:UIControlStateNormal];
 [self.view addSubview:showPicker];
}

- (void)viewDidLoad {
 [self createUI];

 picker = [[ABPeoplePickerNavigationController alloc] init];
 picker.peoplePickerDelegate = self;
}

- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)peoplePicker
shouldContinueAfterSelectingPerson:(ABRecordRef)person {
 [peoplePicker dismissModalViewControllerAnimated:YES];
 [self logPerson:person];
 return NO;
}

- (void)dealloc {
 [picker release];
 [super dealloc];
}

@end
```

## Editing people

See how much easier it is to use a people picker rather than selecting contacts through code? Now you'll see how you can display a contact's details when the person is selected in the people picker, rather than simply closing the picker. You'll also see how to use another class from the Address Book UI framework to visually edit your contact.

### To update the application to allow editing of people:

1. Open PeoplePickerExampleView Controller.h, and add some new instance variables:

```
UILabel *labelContactName;
UILabel *labelAddress;
UIButton *editContact;
ABPersonViewController
→ *editContactViewController;
UINavigationController *navEdit;
```

You add some controls you'll use to display and edit the selected contact. The **ABPersonViewController** will provide the user interface for editing a person. The navigation controller is necessary for correctly displaying the person view controller (**Code Listing 10.6**).

*continues on next page*

**Code Listing 10.6** The updated header file for the people picker.

```
#import <UIKit/UIKit.h>
#import <AddressBookUI/AddressBookUI.h>

@interface PeoplePickerExampleViewController : UIViewController <ABPeoplePickerNavigationControllerDelegate>
{
 ABPeoplePickerNavigationController *picker;
 UIButton *showPicker;

 ABPersonViewController *editContactViewController;
 UINavigationController *navEdit;
 UILabel *labelContactName;
 UILabel *labelAddress;
 UIButton *editContact;
}
@end
```

**2.** Switch to PeoplePickerExample

Controller.m, and update the

**viewDidLoad** method:

```
editContactViewController =
→ [[ABPersonViewController alloc]
→ init];
editContactViewController.
→ allowsEditing = YES;
UIBarButtonItem *closeButton =
→ [[UIBarButtonItem alloc]
→ initWithTitle:@"Close"
→ style:UIBarButtonItemStylePlain
→ target:self action:@selector
→ (closeEdit:)];
[[editContactViewController
→ navigationItem] setLeftBarButtonItem
→ Item:closeButton];
[closeButton release];
navEdit = [[UINavigationController
→ alloc] initWithRootView
→ Controller:editContactView
→ Controller];
```

You first create a new **ABPersonView Controller**, setting the **allowsEditing** property to true since you want to be able to edit the contact. You next create a bar button and add it to the person view controller's navigation toolbar. This is necessary since this control does not actually provide any means of closing itself. Finally, you create the navigation controller required for correctly displaying the person view controller.

3. Update the **createUI** method. In addition to adding some labels to display the selected contact's name and address, you create a button you will use to edit them:

```
editContact = [UIButton buttonWithType:UIButtonTypeCustom];
[editContact addTarget:self
 action:@selector(editContact:)
 forControlEvents:UIControlEventTouchUpInside];
[editContact setFrame:CGRectMake(10,10,60,60)];
[editContact setEnabled:NO];
```

Notice how the button is of type **UIButtonTypeCustom**. This is because you will use the contact image as the background for the button. The button is initially set to disabled since you first need to select a contact with the people picker before you can edit it.

4. Implement the delegate methods used by the people picker. You first update the **peoplePickerController:shouldContinueAfterSelectingPerson:** delegate method you used earlier. By returning **YES**, the people picker will show the details of a person instead of closing.

*continues on next page*

5. Implement the `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:` delegate:

```
[self updateNameLabelForPerson:
→ person];

[self updateImageViewForPerson:
→ person];

[self updateAddressLabelForPerson:
→ person];

editContactViewController.
→ displayedPerson = person;

[editContact setEnabled:YES];

[peoplePicker dismissModalView
→ ControllerAnimated:YES];
```

Here you call some methods to update the user interface with the details of the selected person. You also tell the contact view controller which person has been selected and enable the edit contact button (since you've now selected a contact) before closing the people picker ④.

6. Implement the `editContact:` method that's called when you tap the contact's image:

```
[self presentModalViewController:
→ navEdit animated:YES];
```

Similar to when using the people picker, you tell the navigation controller to display, which in turn shows the person view controller and allows you to edit the contact.



④ Displaying the contact information.



D Editing contact details using an **ABPersonViewController**.

7. Implement the **closeEdit:** method, which is called when the user finishes editing:

```
ABRecordRef person =
→ editContactViewController.
→ displayedPerson;

[self updateNameLabelForPerson:
→ person];

[self updateImageButtonForPerson:
→ person];

[self updateAddressLabelForPerson:
→ person];

[navEdit dismissModalViewControllerAnimated:
→ Animated:YES];
```

As when closing the people picker, you update the user interface to reflect any edits the user may have just made, before closing the navigation controller.

8. Build and run the application.

Notice how the people picker now drills down to a contact's details. Once the people picker has closed, tapping the contact's image allows you to edit their details D. **Code Listing 10.7** shows the updated code.

**TIP** Just as the **ABPersonViewController** provides a user interface to view and edit a contact, the **ABNewPersonViewController** class allows you to add new contacts.

**TIP** For more information on working with the Address Book and Address Book UI frameworks, refer to the *Address Book Programming Guide for iPhone OS* in the developer documentation.

**Code Listing 10.7** The people picker code updated to allow editing of the selected contact.

```
#import "PeoplePickerExampleViewController.h"

@implementation PeoplePickerExampleViewController

- (void)updateNameLabelForPerson:(ABRecordRef)person
{
 CFStringRef name = ABRecordCopyCompositeName(person);
 labelContactName.text = [NSString stringWithFormat:@"%@",name];
}

- (void)updateImageButtonForPerson:(ABRecordRef)person
{
 CFDataRef imgData = ABPersonCopyImageData(person);
 UIImage *contactImage = [UIImage imageNamed:@"noImage.png"];

 if (imgData != nil) {
 contactImage = [UIImage imageWithData:(NSData *)imgData];
 CFRelease(imgData);
 }
 [editContact setBackgroundImage:contactImage forState:UIControlStateNormal];
}

- (void)updateAddressLabelForPerson:(ABRecordRef)person {
 ABMultiValueRef addressValues = ABRecordCopyValue(person,kABPersonAddressProperty);
 NSArray *arrAddresses = [NSArray arrayWithArray:(id)ABMultiValueCopyArrayOfAllValues(addressValues)];

 if ([arrAddresses count]) {
 NSDictionary *dictAddress = [arrAddresses objectAtIndex:0];
 NSString *street = [dictAddress objectForKey:(NSString *)kABPersonAddressStreetKey];
 NSString *city = [dictAddress objectForKey:(NSString *)kABPersonAddressCityKey];
 NSString *state = [dictAddress objectForKey:(NSString *)kABPersonAddressStateKey];
 NSString *zip = [dictAddress objectForKey:(NSString *)kABPersonAddressZIPKey];
 NSString *country = [dictAddress objectForKey:(NSString *)kABPersonAddressCountryKey];
 labelAddress.text = [NSString stringWithFormat:@"%@\n%@ %@\n%@",street,city,state,zip,country];
 }
}

- (void)editContact:(id)sender {
 [self presentModalViewController:navEdit animated:YES];
}

- (void)showPeoplePicker:(id)sender {
 [self presentModalViewController:picker animated:YES];
}

- (void)closeEdit:(id)sender {
 ABRecordRef person = editContactViewController.displayedPerson;
 [self updateNameLabelForPerson:person];
 [self updateImageButtonForPerson:person];
 [self updateAddressLabelForPerson:person];

 [navEdit dismissModalViewControllerAnimated:YES];
}

- (void)createUI {
 showPicker = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 [showPicker addTarget:self action:@selector(showPeoplePicker:) forControlEvents:UIControlEventTouchUpInside];
 [showPicker setFrame:CGRectMake(198,10,120,38)];
 [showPicker setTitle:@"People Picker" forState:UIControlStateNormal];
 [self.view addSubview:showPicker];

 labelContactName = [[UILabel alloc] initWithFrame:CGRectMake(10,80,300,28)];
 labelContactName.backgroundColor = [UIColor clearColor];
 [self.view addSubview:labelContactName];
}
```

*code continues on next page*

### Code Listing 10.7 continued

```
labelAddress = [[UILabel alloc] initWithFrame:CGRectMake(10,110,300,70)];
labelAddress.numberOfLines = 3;
labelAddress.backgroundColor = [UIColor clearColor];
[self.view addSubview:labelAddress];

editContact = [UIButton buttonWithType:UIButtonTypeCustom];
[editContact addTarget:self action:@selector(editContact:) forControlEvents:UIControlEventTouchUpInside];
[editContact setFrame:CGRectMake(10,10,60,60)];
[editContact setEnabled:YES];
[self.view addSubview:editContact];
}

-(void)viewDidLoad {
 [self createUI];

picker = [[ABPeoplePickerNavigationController alloc] init];
picker.peoplePickerDelegate = self;

editContactViewController = [[ABPersonViewController alloc] init];
editContactViewController.allowsEditing = YES;
UIBarButtonItem *closeButton = [[UIBarButtonItem alloc]
 initWithTitle:@"Close"
 style:UIBarButtonItemStylePlain
 target:self action:@selector(closeEdit:)];
[[editContactViewController navigationItem] setLeftBarButtonItem:closeButton];
[closeButton release];
navEdit = [[UINavigationController alloc] initWithRootViewController:editContactViewController];
}

-(BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)peoplePicker
shouldContinueAfterSelectingPerson:(ABRecordRef)person {
 return YES;
}

-(BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController *)peoplePicker
shouldContinueAfterSelectingPerson:(ABRecordRef)person
 property:(ABPropertyID)property
 identifier:(ABMultiValueIdentifier)identifier {
 [self updateNameLabelForPerson:person];
 [self updateImageButtonForPerson:person];
 [self updateAddressLabelForPerson:person];

 editContactViewController.displayedPerson = person;
 [editContact setEnabled:YES];

 [peoplePicker dismissModalViewControllerAnimated:YES];
 return NO;
}

-(void)peoplePickerNavigationControllerDidCancel:(ABPeoplePickerNavigationController *)peoplePicker {
 [peoplePicker dismissModalViewControllerAnimated:YES];
}

-(void)dealloc {
 [picker release];
 [editContactViewController release];
 [navEdit release];
 [labelContactName release];
 [labelAddress release];
 [super dealloc];
}
@end
```

# The iPhone Calendar

Items in the iPhone calendar application are known as *events* and are represented by instances of the **EKEvent** class. You access data in the calendar database by using an **EKEventStore** object.

## To retrieve events from the default calendar:

1. Create an **EKEventStore** object, which is used to access the calendar database:

```
EKEventStore *eventStore =
→ [[EKEventStore alloc] init];

NSDate *startDate = [NSDate date];

NSDateComponents *comp =
→ [[NSDateComponents alloc] init];

[comp setMonth:1];

NSCalendar *cal =
→ [[NSCalendar alloc] initWith
→ CalendarIdentifier:NSGregorian
→ Calendar];

NSDate *endDate = [cal dateBy
→ AddingComponents:comp toDate:
→ startDate options:0];
```

Next you create two date objects: one representing today and another for a month from today.

2. Create a predicate from these dates, and pass it to your event store object, returning an array of matching **EKEvent** objects:

```
NSPredicate *predicate =
→ [eventStore predicateForEvents
→ WithStartDate:startDate endDate:
→ endDate calendars:nil];

NSArray *matchingEvents =
→ [eventStore eventsMatching
→ Predicate:predicate];
```

**Code Listing 10.8** shows an example of retrieving events from the default calendar for the next month.

**TIP** To work with events, you will need to add the Event Kit framework into your application and import the EventKitUI/EventKitUI.h header file.

**TIP** Notice how you pass `nil` in the `calendars` property when creating the predicate in the previous example. This tells the event store that you want to use the default calendar for your search. In this chapter, you will be working with the default calendar, but remember that it's possible to have multiple calendars. You can retrieve these calendars as an array by using the `calendars` property of an `EKEventStore` object.

**Code Listing 10.8** Querying the calendar event store.

```
- (void)logCalendarEventsForNextMonth
{
 EKEventStore *eventStore = [[EKEventStore alloc] init];

 NSDate *startDate = [NSDate date];

 //create a date representing 1 month from now
 NSDateComponents *comp = [[NSDateComponents alloc] init];
 [comp setMonth:+1];
 NSCalendar *cal = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
 NSDate *endDate = [cal dateByAddingComponents:comp toDate:startDate options:0];

 //create a predicate for date range
 NSPredicate *predicate = [eventStore predicateForEventsWithStartDate:startDate endDate:endDate calendars:nil];
 NSArray *matchingEvents = [eventStore eventsMatchingPredicate:predicate];

 for (EKEvent *e in matchingEvents)
 {
 NSLog(@"%@", e.eventIdentifier);
 NSLog(@"%@", e.title);
 NSLog(@"%@", e.startDate);
 NSLog(@"%@", e.endDate);
 NSLog(@"%@", e.location);
 NSLog(@"%@", e.notes);
 }

 //cleanup
 [cal release];
 [comp release];
 [eventStore release];
}
```

## Events

As previously mentioned, events are stored in the calendar database as **EKEvent** objects. Events have many properties. Some of the more commonly used include the following:

- **eventIdentifier**—A unique identifier for the event
- **title**—An **NSString** representing the title of the event
- **startDate/endDate**—The start and end date of the event
- **alarms**—An array of **EKAlarm** objects, each one representing an alarm for the event
- **calendar**—The calendar to which the event belongs
- **location**—An **NSString** representing any location associated with the event
- **notes**—An **NSString** representing any notes associated with the event
- **recurrenceRule**—An **EKRecurrenceRule** object that describes the recurrence pattern for the event

Events can be added, updated, and removed from the calendar database via an **EKEventStore** object.

## To create an event for a birthday:

1. Create an event store you will use to add the event to the calendar database:

```
EKEventStore *eventStore =
→ [[EKEventStore alloc] init];

NSDateComponents *comp =
→ [[NSDateComponents alloc] init];
[comp setDay:1];
[comp setMonth:2];
[comp setYear:2010];

NSCalendar *cal =
→ [[NSCalendar alloc] initWith
→ CalendarIdentifier:NSGregorian
→ Calendar];

NSDate *eventDate =
→ [cal dateFromComponents:comp];
```

You also create an **NSDate** object representing the event date (in this case, Feb 1st, 2010—the author's birthday!).

2. Next create a recurrence rule to determine the frequency of the event. Since this event is a birthday, you set the frequency to recur yearly:

```
EKRecurrenceRule *recurrence =
→ [[EKRecurrenceRule alloc]
initRecurrenceWithFrequency:
→ EKRecurrenceFrequencyYearly
→ interval:1end:[EKRecurrenceEnd
→ recurrenceEndWithEndDate:
→ [NSDate distantFuture]]];
```

3. Create an **EKRecurrenceRule** object.

Note how by using **[NSData distantFuture]** for the **end** property of the recurrence, you effectively say it recurs forever.

*continues on next page*

4. Set an alarm by creating an `EKAlarm` object and setting it to occur one day before the event:

```
double alarmAmountInSeconds =
 → 60.0 * 60.0 * 24.0;
EKAlarm *alarm = [EKAlarm
 → alarmWithRelativeOffset:(-1.0 *
 → alarmAmountInSeconds)];
```

5. Next create the actual event, and set some of its properties:

```
EKEvent *myEvent =
 → [EKEvent eventWithEventStore:
 → eventStore];
myEvent.calendar = eventStore.
 → defaultCalendarForNewEvents;
myEvent.title = @"My Birthday";
myEvent.startDate = eventDate;
myEvent.endDate = eventDate;
myEvent.allDay = TRUE;
myEvent.recurrenceRule =
 → recurrance;
myEvent.alarms = [NSArray
 → arrayWithObject:alarm];
myEvent.notes = @"Remember to buy
 → a present!";
```

Note how you add it to the default calendar of the event store.

6. Finally, add the event to the event store and log its identifier to the console:

```
NSError *err = nil;
BOOL result = [eventStore
 → saveEvent:myEvent span:
 → EKSpanFutureEvents error:&err];
```

Code Listing 10.9 show an example of creating an event for a birthday.

**Code Listing 10.9** Creating an event.

```
- (void)createBirthdayEvent
{
 EKEventStore *eventStore = [[EKEventStore alloc] init];

 //create date representing the event
 NSDateComponents *comp = [[NSDateComponents alloc] init];
 [comp setDay:1];
 [comp setMonth:2];
 [comp setYear:2010];
 NSCalendar *cal = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
 NSDate *eventDate = [cal dateFromComponents:comp];

 //create recurrence rule
 EKRecurrenceRule *recurrence = [[EKRecurrenceRule alloc]
 initRecurrenceWithFrequency:EKRecurrenceFrequencyYearly
 interval:1
 end:[EKRecurrenceEnd recurrenceEndWithEndDate:[NSDate distantFuture]]];

 //create alarm 1 day before event
 double alarmAmountInSeconds = 60.0 * 60.0 * 24.0;
 EKAlarm *alarm = [EKAlarm alarmWithRelativeOffset:(-1.0 * alarmAmountInSeconds)];

 //create event
 EKEvent *myEvent = [EKEvent eventWithEventStore:eventStore];
 myEvent.calendar = eventStore.defaultCalendarForNewEvents;
 myEvent.title = @"My Birthday";
 myEvent.startDate = eventDate;
 myEvent.endDate = eventDate;
 myEvent.allDay = TRUE;
 myEvent.recurrenceRule = recurrence;
 myEvent.alarms = [NSArray arrayWithObject:alarm];
 myEvent.notes = @"Remember to buy a present!";

 //add to event store
 NSError *err = nil;
 BOOL result = [eventStore saveEvent:myEvent span:EKSpanFutureEvents error:&err];

 //log result
 if (!result)
 NSLog(@"error saving event: %@",err);
 else
 NSLog(@"event (%@) saved",myEvent.eventIdentifier);

 //cleanup
 [recurrence release];
 [cal release];
 [comp release];
 [eventStore release];
}
```

## Viewing event details

Just as with the Address Book UI frameworks previously discussed, Apple provides the Event Kit UI framework for visually managing calendar events with a user interface similar to the built-in Calendar application.

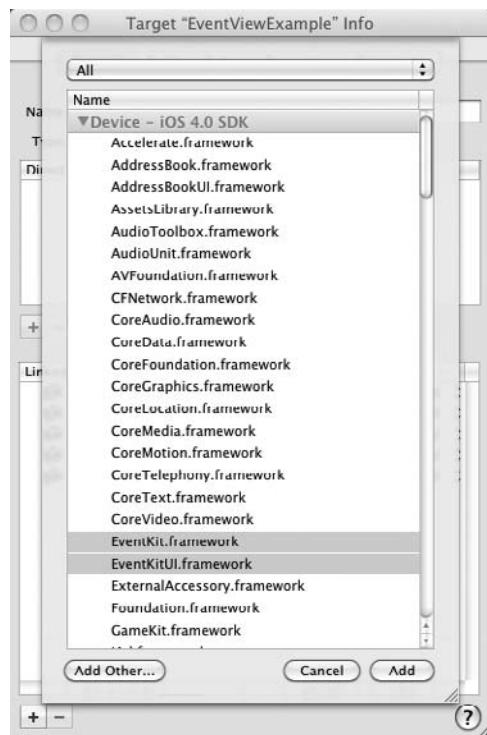
The Event Kit UI framework offers two main classes:

- **EKEventViewController**—Displays details about the calendar event
- **EKEventEditViewController**—Is used to add and edit existing calendar events

Next you'll see how to go about viewing details of a calendar event using the **EKEventViewController** class.

### To create an application to view calendar events:

1. Create a new navigation-based application, saving it as EventViewExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add both EventKit.framework and EventKitUI.framework **A**.
4. Open RootViewController.h, and import the Event Kit headers. Here you also create two properties: an array to hold the calendar events and an **EKEventStore** object that will be used to access the calendar database (**Code Listing 10.10**).



**A** Adding the Event Kit frameworks.

**Code Listing 10.10** The header file for the view event details example.

```
#import <UIKit/UIKit.h>
#import <EventKit/EventKit.h>
#import <EventKitUI/EventKitUI.h>

@interface RootViewController : UITableViewController
{
 NSArray *eventArray;
 EKEventStore *eventStore;
}

@property (nonatomic, retain) NSArray *eventArray;
@property (nonatomic, retain) EKEventStore *eventStore;
@end
```

5. Switch to RootViewController.m, uncomment the viewDidLoad method, and add the following code:

```
self.title = @"Events";
self.navigationController.
→ delegate = self;
self.eventStore = [[EKEventStore
→ alloc] init];
```

Since you are working with a navigation-based application, the first thing you do is set the title before creating the calendar store object that you'll be using to access the calendar database.

Finally, you call the method **refreshSampleData**, which queries the data store for events occurring in the next month, before forcing the table view to reload.

6. Implement the **tableView** delegate methods. **tableView:numberOfRowsInSection:** simply returns the number of elements in the array. To return the actual data for your **tableView**, you implement the **tableView:cellForRowAtIndexPath:** method:

```
EKEvent *event = [self.eventArray
→ objectAtIndex:indexPath.row];
cell.textLabel.text = event.title;
```

After extracting the event from the events array using the current **tableView** row, you use the title of the event as the title of cell.

*continues on next page*

7. Finally, implement the `tableView: didSelectRowAtIndexPath:IndexPath` method, which will be called when you select a row:

```
EKEvent *event = [self.eventArray
→ objectAtIndex:indexPath.row];
[self viewEvent:event];
```

Again, you query the event array to find the event the user has selected and pass this to the `viewEvent` method:

```
EKEventViewController
→ *editController = [[EKEventView
→ Controller alloc] init];
editController.event = event;
[self.navigationController
→ pushViewController:
→ editController animated:YES];
[editController release];
```

Here you create an `EKEventView Controller` object, set its `event` property to the value selected, and push it onto the navigation controllers stack.

8. Build and run your application.

You should see a list of events. Selecting one will display the details for the event ❸. Note that you might have to adjust the predicate defined in the `refreshSampleData` method if you don't have any calendar events for the next month.

**Code Listing 10.11** shows the completed code.



❸ Viewing details of an event.

**Code Listing 10.11** Viewing an event using an `EKEventViewController`.

```
#import "RootViewController.h"

@implementation RootViewController

@synthesize eventArray, eventStore;

- (void)refreshSampleData {

 NSDate *startDate = [NSDate date];
```

code continues on next page

**Code Listing 10.11** continued

```
//create a date representing 1 month from now
NSDateComponents *comp = [[NSDateComponents alloc] init];
[comp setMonth:1];
NSCalendar *cal = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *endDate = [cal dateByAddingComponents:comp toDate:[NSDate date] options:0];
[comp release];
[cal release];

//create a predicate for date range
NSPredicate *predicate = [eventStore predicateForEventsWithStartDate:[NSDate date] endDate:endDate calendars:nil];
NSArray *matchingEvents = [eventStore eventsMatchingPredicate:predicate];

//save in property and tell tableview to refresh itself
self.eventArray = [[NSArray alloc] initWithArray:matchingEvents];
[self.tableView reloadData];
}

- (void)viewEvent:(EKEvent *)event
{
 EKEventViewController *editController = [[EKEventViewController alloc] init];
 editController.event = event;
 [self.navigationController pushViewController:editController animated:YES];
 [editController release];
}

- (void)viewDidLoad {
 self.title = @"Events";
 self.eventStore = [[EKEventStore alloc] init];
 [self refreshSampleData];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
 return [self.eventArray count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
 static NSString *CellIdentifier = @"Cell";

 UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
 if (cell == nil)
 cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:CellIdentifier] autorelease];

 cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

 EKEvent *event = [self.eventArray objectAtIndex:indexPath.row];
 cell.textLabel.text = event.title;

 return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
 EKEvent *event = [self.eventArray objectAtIndex:indexPath.row];
 [self viewEvent:event];
}

- (void)dealloc {
 [eventArray release];
 [eventStore release];
 [super dealloc];
}
```

## Editing events

As mentioned, the Event Kit UI framework offers a second class (**EKEventEditView Controller**) that you can use to add and edit calendar events. Now you'll update the application to add this additional functionality.

### To update the application to add and edit calendar events:

1. Open RootViewController.h, and add protocol declarations for **UINavigationControllerDelegate** and **EKEventEditViewDelegate** (Code Listing 10.12), which will allow you implement the methods required for editing of calendar events.
2. Switch to RootViewController.m, and update the **viewDidLoad** method:

```
UIBarButtonItem * addButton =
→ [[UIBarButtonItem alloc]
→ initWithBarButtonSystemItem:
→ UIBarButtonSystemItemAdd target:
→ self action:@selector
→ (addEvent:)];
self.navigationItem.rightBarButtonItem =
→ButtonItem = addButton;
[addButton release];
```

**Code Listing 10.12** The updated header file.

```
#import <UIKit/UIKit.h>
#import <EventKit/EventKit.h>
#import <EventKitUI/EventKitUI.h>

@interface RootViewController : UITableViewController <UINavigationControllerDelegate,
EKEventEditViewDelegate>
{
 NSArray *eventArray;
 EKEventStore *eventStore;
}
@property (nonatomic, retain) NSArray *eventArray;
@property (nonatomic, retain) EKEventStore *eventStore;
@end
```

First you tell the navigation controller that you want the current class to be the delegate. This is necessary since you need to refresh the table view when you edit or delete events.

You then create a navigation bar button that you will use to add new events. Notice that the call to `refreshSampleData` has been removed because you are now performing this in the `navigationController:willShowController:animated` method, which will be called when the navigation controller first loads.

3. Create the `addEvent:` method that is called when you press the navigation bar button:

```
EKEventEditViewController
→ *addController = [[EKEventEdit
→ ViewController alloc] init];
addController.eventStore =
→ self.eventStore;
addController.editViewDelegate =
→ self;
[self presentModalView
→ Controller:addController
→ animated:YES];
[addController release];
```

Here you create an `EKEventEditViewController` object, which is used to add the event. Notice how you set its `eventStore` to the same event store being currently used by the other events. You also set the delegate, which is how you will be informed of events being added.

*continues on next page*

4. Define the `eventEditViewController:didCompleteWithAction:` method, which is called when you have added a new event, and press Save:

```
if (action == EKEventEditView
→ ActionSaved)
{
 [controller.eventStore saveEvent:
→ controller.event span:
→ EKSpanThisEvent error:nil];
 [self refreshSampleData];
}
[controller dismissModalView
→ ControllerAnimated:YES];
```

You first check the `action` parameter to make sure the user is saving the new event and then tell the event store to save the new event. Notice how you also refresh the sample data to force the table view to redraw itself correctly. Finally, you make sure the `EKEventEditViewController` is properly dismissed.

5. Add the following to the `viewEvent` method:

```
editController.allowsEditing = YES;
```

Setting this property tells your `EKEventViewController` that you want to be able to edit events.

6. Implement the `navigationController:willShowViewController:animated:` delegate that is used to refresh the table whenever any data changes:

```
if (viewController == self)
 [self refreshSampleData];
```

Here you do a quick check to make sure that you are showing the correct view controller. If so, you refresh the event data, causing the table view to refresh.



① Editing an event.

## 7. Build and run the application.

When you view an existing event, you should now see an edit button in the navigation bar. Tap it to edit the event details . **Code Listing 10.13** shows the completed code.

**Code Listing 10.13** The completed code for adding and editing calendar events.

```
#import "RootViewController.h"

@implementation RootViewController

@synthesize eventArray, eventStore;

- (void)refreshSampleData {
 NSDate *startDate = [NSDate date];

 //create a date representing 1 month from now
 NSDateComponents *comp = [[NSDateComponents alloc] init];
 [comp setMonth:1];
 NSCalendar *cal = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
 NSDate *endDate = [cal dateByAddingComponents:comp toDate:startDate options:0];
 [comp release];
 [cal release];

 //create a predicate for date range
 NSPredicate *predicate = [eventStore predicateForEventsWithStartDate:startDate endDate:endDate calendars:nil];
 NSArray *matchingEvents = [eventStore eventsMatchingPredicate:predicate];

 //save in property and tell tableview to refresh itself
 self.eventArray = [[NSArray alloc] initWithArray:matchingEvents];
 [self.tableView reloadData];
}

- (void)viewEvent:(EKEvent *)event
{
 EKEventViewController *editController = [[EKEventViewController alloc] init];
 editController.event = event;
 editController.allowsEditing = YES;
 [self.navigationController pushViewController:editController animated:YES];
 [editController release];
}

- (void)addEvent:(id)sender
{
 EKEventEditViewController *addController = [[EKEventEditViewController alloc] init];
 addController.eventStore = self.eventStore;
 addController.editViewDelegate = self;

 [self presentViewController:addController animated:YES];
 [addController release];
}

- (void)viewDidLoad {
 self.title = @"Events";
 self.eventStore = [[EKEventStore alloc] init];
 self.navigationController.delegate = self;
}
```

*code continues on next page*

**Code Listing 10.13** *continued*

```
UIBarButtonItem *addButton = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
 target:self
 action:@selector(addEvent:)];
self.navigationItem.rightBarButtonItem = addButton;
[addButton release];
}

-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
 return [self.eventArray count];
}

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
 static NSString *CellIdentifier = @"Cell";
 UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
 if (cell == nil)
 cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:CellIdentifier] autorelease];
 cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
 EKEvent *event = [self.eventArray objectAtIndex:indexPath.row];
 cell.textLabel.text = event.title;
 return cell;
}

-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
 EKEvent *event = [self.eventArray objectAtIndex:indexPath.row];
 [self viewEvent:event];
}

-(void)navigationController:(UINavigationController *)navigationController
willShowViewController:(UIViewController *)viewController
animated:(BOOL)animated {
 if (viewController == self)
 [self refreshSampleData];
}

-(void)eventEditViewController:(EKEventEditViewController *)controller
didCompleteWithAction:(EKEventEditViewAction)action {
 if (action == EKEventEditViewActionSaved)
 {
 [controller.eventStore saveEvent:controller.event span:EKSpanThisEvent error:nil];
 [self refreshSampleData];
 }
 [controller dismissModalViewControllerAnimated:YES];
}

-(void)dealloc {
 [eventArray release];
 [eventStore release];
 [super dealloc];
}
```

# E-mail

Although the iPhone SDK does not provide any means to *receive* e-mail, it does offer an API for *sending* e-mail. You can send e-mail from your applications in two ways. The first is to create a specially formatted URL that causes the Mail application to be launched:

```
NSString *mailString = @"mailto:
→ steve@apple.com?subject=test&body=
→ some content";

NSURL *mailURL = [NSURL URLWithString:
→ String:mailString];

[[UIApplication sharedApplication]
→ openURL:mailURL];
```

This technique has several drawbacks:

- There is a limit to the length of the string that can be used via a `mailto:` URL, meaning your e-mail messages will need to be quite short.
- You can't attach files to the e-mail.
- Your application will be closed, and the Mail application will be opened. Once the e-mail has been sent, the user is left in the Mail application—not exactly the perfect user experience!

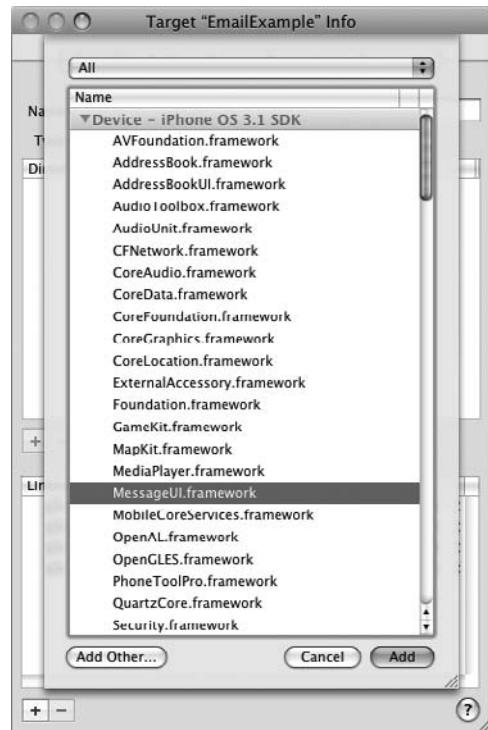
The second, preferred, option is to use the **MessageUI** framework, which provides a standard interface for composing and sending e-mails from within your applications via the **MFMailComposeViewController** class.

E-mails composed using this class are placed in the outbox of the Mail application so that even if the user doesn't have network access, the message can be delivered at a later date. Unfortunately, this also means that the class doesn't provide any notification about whether the e-mail is actually sent.

(A third option is to use a third-party e-mail library such as the **skpsmtpmessage** class, available at <http://code.google.com/p/skpsmtpmessage>.)

## To create an application to compose and send e-mail:

1. Create a new view-based application, saving it as EmailExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.
3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add MessageUI.framework **A**.
4. Open EmailExampleViewController.h, import the MFMailComposeViewController.h header, and add the **MFMailComposeViewControllerDelegate** protocol declaration (**Code Listing 10.14**).



**A** Adding the **MessageUI** framework.

**Code Listing 10.14** The header file for the e-mail application.

```
#import <UIKit/UIKit.h>
#import <MessageUI/MFMailComposeViewController.h>

@interface EmailExampleViewController : UIViewController <MFMailComposeViewControllerDelegate>
{
}

@end
```

5. Switch to EmailExampleViewController.m, uncomment the `viewDidLoad` method, and add the following code:

```
UIButton *compose = [UIButton
→ buttonWithType:UIButtonType
→ RoundedRect];

CGRect composeFrame =
→ CGRectMake(80,10,160,38);

[compose setFrame:composeFrame];
[compose setTitle:@"Compose Mail"
→ forState:UIControlStateNormal];
[compose addTarget:self action:
→ @selector(composeClick:)
→ forControlEvents:UIControlEventTouchUpInside
→ TouchUpInside];

[self.view addSubview:compose];
```

Here you are adding a button that calls the `composeClick:` method when tapped.

6. Implement the `composeClick:` method:

```
if ([MFMailComposeViewController
→ canSendMail])
```

The first thing you do is call the `canSendMail` class method to make sure you are capable of sending e-mail from your application. If the user has not configured their iPhone with an e-mail account, this method will return `NO`.

```
MFMailComposeViewController
→ *mailController = [[MFMail
→ ComposeViewController alloc]
→ init];

mailController.mailCompose
→ Delegate = self;
```

Next you create the mail compose view controller that will be used to present an interface to your user for composing their e-mail.

*continues on next page*

```
NSArray *to = [[NSArray alloc]
→ initWithObjects:@"steve@apple
→ .com",@"bill@microsoft.com",nil];
[mailController setToRecipients:
→ to];
[to release];
```

Then you create an array of e-mail addresses to send the e-mail to and assign to the `mailController` using the `setToRecipients:` method. You can also set the CC and BCC recipients in the same way by using the `setCcRecipients:` and `setBccRecipients:` methods.

```
NSString *subject = @"some
→ subject";
[mailController setSubject:
→ subject];

NSString *body = @"An
→ important email.";
[mailController setMessageBody:
→ body isHTML:YES];
```

You then set the subject and the body of the e-mail. Notice how you can use the `isHTML` parameter to specify that the body contents are HTML.

```
NSString *filePath = [[NSBundle
→ mainBundle] pathForResource:
→ @"apple" ofType:@"png"];
NSData *fileData = [NSData
→ dataWithContentsOfFile:
→ filePath];
[mailController addAttachment
→ Data:fileData mimeType:@"image/
→ png" fileName:@"apple"];
```



- ③ Composing an e-mail in your application.

You next attach a file to e-mail; in this example, it's an image called apple.png located in the resources folder of the application bundle.

```
[self presentModalViewController:
→ mailController animated:YES];
```

Lastly, you show the compose interface ④. The user is able to edit any of the mail settings (and of course, you need not have applied any settings before showing the compose interface). Notice how the signature is automatically appended to the mail body.

7. Implement the `mailComposeController:didFinishWithResult:error:` delegate method:

```
switch (result)
{
case MFMailComposeResultCancelled:
 break;
case MFMailComposeResultSaved:
 NSLog(@"mail was saved");
 break;

case MFMailComposeResultSent:
 NSLog(@"mail was sent");
 break;

case MFMailComposeResultFailed:
 NSLog(@"Error: %@",error);
 break;
}
[self dismissModalViewControllerAnimated:
→ YES];
```

*continues on next page*

This method will be called if either the Cancel or Send button is tapped in the compose interface. You can check the result parameter to see what the user did. If the e-mail can't be sent right away, it will be saved into the user's outbox. You can use the error parameter to check for any errors. Don't forget to hide the compose interface in this method.

## 8. Build and run your application.

Since the simulator does not have any e-mail capabilities, you will have to run the application on an iPhone to test it. **Code Listing 10.15** shows the completed code.

**Code Listing 10.15** The completed code for the e-mail application.

```
#import "EmailExampleViewController.h"

@implementation EmailExampleViewController

- (void)composeClick:(id)sender {
 if ([MFMailComposeViewController canSendMail]) {
 MFMailComposeViewController *mailController = [[MFMailComposeViewController alloc] init];
 mailController.mailComposeDelegate = self;

 NSArray *to = [[NSArray alloc]
 initWithObjects:@"steve@apple.com",@"bill@microsoft.com",nil];
 [mailController setToRecipients:to];
 [to release];

 NSString *subject = @"some subject";
 [mailController setSubject:subject];

 NSString *body = @"An important email.";
 [mailController setMessageBody:body isHTML:YES];

 NSString *filePath = [[NSBundle mainBundle]
 pathForResource:@"apple"
 ofType:@"png"];
 NSData *fileData = [NSData dataWithContentsOfFile:filePath];
 [mailController addAttachmentData:fileData
 mimeType:@"image/png"
 fileName:@"apple"];

 [self presentModalViewController:mailController animated:YES];
 [mailController release];
 }
 else {

```

*code continues on next page*

**Code Listing 10.15** *continued*

```
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error"
 message:@"Can't send email!"
 delegate:self
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];
[alert show];
[alert release];
}
}

-(void)viewDidLoad {
UIButton *compose = [UIButton buttonWithType:UIButtonTypeRoundedRect];
CGRect composeFrame = CGRectMake(80,10,160,38);
[compose setFrame:composeFrame];
[compose setTitle:@"Compose Mail" forState:UIControlStateNormal];
[compose addTarget:self
 action:@selector(composeClick:)
 forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:compose];
}

-(void)mailComposeController:(MFMailComposeViewController*)controller
 didFinishWithResult:(MFMailComposeResult)result
 error:(NSError*)error {
switch (result) {
{
 case MFMailComposeResultCancelled:
 break;

 case MFMailComposeResultSaved:
 NSLog(@"mail was saved");
 break;

 case MFMailComposeResultSent:
 NSLog(@"mail was sent");
 break;

 case MFMailComposeResultFailed:
 NSLog(@"Error: %@",error);
 break;
}
}

[self dismissModalViewControllerAnimated:YES];
}

-(void)dealloc {
 [super dealloc];
}

@end
```

# SMS

As with sending e-mails, there are two methods you can use to send an SMS. The first is to create a specially formatted URL that causes the SMS application to be launched:

```
NSString *smsString = @"sms:5551234";
→ NSURL *smsURL = [NSURL URLWithString:smsString];
→ String: smsString];
[[UIApplication sharedApplication]
→ openURL:smsURL];
```

But as with sending e-mails, this results in a fairly poor user experience, since the user leaves your application and is forced into the SMS application.

A better technique is to use an **MFMessageComposeViewController** object, which works in a very similar manner to the previous e-mail example.

Just like sending an e-mail, you specify an array of recipients and the body of the SMS. However, unlike sending e-mail, you cannot programmatically add attachments to the SMS. And of course, you don't need to specify a subject.

## To create an application to compose and send an SMS:

1. Create a new view-based application, saving it as SMSExample.
2. In the Groups & Files pane, expand the Targets section, right-click your application target, and select Get Info.

3. Making sure the General tab is selected, click Add (+) at the bottom of the Linked Libraries list, and add MessageUI.framework.
4. Open SMSExampleViewController.h, import the MFMessageComposeViewController.h header, and add the **MFMessageComposeViewControllerDelegate** protocol declaration (**Code Listing 10.16**).
5. Switch to SMSExampleViewController.m, uncomment the **viewDidLoad** method, and add the following code:

```
UIButton *compose = [UIButton
→ buttonWithType:UIButtonTypeRoundedRect];
CGRect composeFrame =
→ CGRectMake(80,10,160,38);
[compose setFrame:composeFrame];
[compose setTitle:@"Compose SMS"
→ forState:UIControlStateNormal];
[compose addTarget:self action:
→ @selector(composeClick:)
→ forControlEvents:UIControlEventTouchUpInside];
[self.view addSubview:compose];
```

Here you are adding a button that calls the **composeClick:** method when tapped.

*continues on next page*

**Code Listing 10.16** The header file for the SMS application.

```
#import <UIKit/UIKit.h>
#import <MessageUI/MFMessageComposeViewController.h>

@interface SMSExampleViewController : UIViewController <MFMessageComposeViewControllerDelegate>
{
}
@end
```

6. Implement the `composeClick:` method:

```
if ([MFMessageComposeView
→ ControllercanSendText])
```

Just as in the example earlier, where you composed and sent an email, you first call the `canSendText` class method to make sure your device is capable of sending SMS.

```
MFMessageComposeViewController
→ *messageController =
→ [[MFMessageComposeViewController
→ alloc] init];

messageController.messageCompose
→ Delegate = self;
```

Next you create the SMS compose view controller that will be used to present an interface to your user for composing their SMS and set the delegate to the current class.

```
[messageController setBody:
→ @"Hello world!"];
```

Then you create the body of the SMS. Note how, unlike the earlier e-mail example, you have not specified the recipients of the SMS because you are leaving this up to the user to do in the SMS compose window. (Of course, you could have set this property by creating an array of phone numbers and using the `recipient` property.)

```
[self presentModalViewController:
→ messageController animated:YES];
```

Finally, you tell the view controller to show the SMS compose interface A. Again as in the earlier example, the user is able to edit any of the SMS settings.



A Composing an SMS in your application.

7. Implement the `messageComposeViewController:didFinishWithResult:` delegate method:

```
switch (result)
{
 case MessageComposeResultCancelled:
 break;
 case MessageComposeResultSent:
 NSLog(@"SMS was sent");
 break;

 case MessageComposeResultFailed:
 NSLog(@"SMS Failed to send");
 break;
}
```

[self dismissModalViewControllerAnimated:YES];

This method will be called if either the Cancel or Send button is tapped in the compose interface. You can check the result parameter to see what the user did.

8. Build and run your application.

Since the simulator does not have any SMS capabilities, you will have to run the application on an iPhone to test it. **Code Listing 10.17** shows the completed code.

**Code Listing 10.17** The completed code for the SMS application.

```
#import "SMSExampleViewController.h"

@implementation SMSExampleViewController

- (void)composeClick:(id)sender {
 if ([MFMessageComposeViewController canSendText]) {
 MFMessageComposeViewController *messageController = [[MFMessageComposeViewController alloc] init];
 messageController.messageComposeDelegate = self;

 [messageController setBody:@"Hello world!"];
 [self presentModalViewController:messageController animated:YES];
 [messageController release];
 } else {
 UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error"
 message:@"Can't send SMS!"
 delegate:self
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil];
 [alert show];
 [alert release];
 }
}

- (void)viewDidLoad {
 UIButton *compose = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 CGRect composeFrame = CGRectMake(80,10,160,38);
 [compose setFrame:composeFrame];
 [compose setTitle:@"Compose SMS" forState:UIControlStateNormal];
 [compose addTarget:self
 action:@selector(composeClick:)
 forControlEvents:UIControlEventTouchUpInside];
 [self.view addSubview:compose];
}

- (void)messageComposeViewController:(MFMessageComposeViewController *)controller
 didFinishWithResult:(MessageComposeResult)result {
 switch (result) {
 case MessageComposeResultCancelled:
 break;

 case MessageComposeResultSent:
 NSLog(@"message was sent");
 break;

 case MessageComposeResultFailed:
 NSLog(@"SMS Failed to send");
 break;
 }

 [self dismissModalViewControllerAnimated:YES];
}

- (void)dealloc {
 [super dealloc];
}

@end
```

# 11

# Multitasking

One of the most exciting new features in iOS4 is the introduction of multitasking. Previously, the iPhone was capable of running only one application at a time, but now your application can continue to run in the background when you close it and switch to another.

In this chapter, you'll get a brief overview of some of the multitasking features of iOS 4 and how they work.

---

## In This Chapter

|                                   |     |
|-----------------------------------|-----|
| What Is Multitasking?             | 456 |
| Responding to Local Notifications | 466 |

---

# What Is Multitasking?

Quite simply, *multitasking* is the ability to do more than one thing at the same time. On your desktop computer, for example, you might be running an e-mail client and a Web browser while simultaneously listening to music and downloading a file over FTP. Because your desktop operating system has been designed with multitasking at its core, all of these applications can essentially run at the same time with only minimal impact to the general system performance and each other.

Multitasking on the iPhone, however, is not quite the same as on the desktop. Although powerful, the iPhone has limited memory, CPU, and other resources. Running multiple applications at once would quickly use up these system resources, resulting in applications becoming slow and unstable. Worst of all, your battery life could be significantly impacted.

Apple has gotten around this problem by implementing multitasking in a special way. When you exit your application, instead of quitting, it is suspended and enters a special *background mode*. When your application resumes, you return immediately to where you were when you left, with all your application's data and state preserved.

At any time, double-tapping the Home button causes the recently used apps to slide up from the bottom of the screen **A**. This dock shows icons of all the applications currently in background mode. Tapping an icon will instantly resume that application. (You can, of course, also resume your application by tapping its desktop icon.)



**A** Applications running in the background are shown in the recently used apps area.

| Key                                      | Value                               |
|------------------------------------------|-------------------------------------|
| ▼ Information Property List              |                                     |
| Application does not run in background   | <input checked="" type="checkbox"/> |
| Application does not run in background   | English                             |
| Application requires iPhone environment  | \$PRODUCT_NAME                      |
| Application supports iTunes file sharing | \$(EXECUTABLE_NAME)                 |
| Application uses Wi-Fi                   |                                     |
| Bundle creator OS Type code              | com.yourcompany.\$P                 |
| Bundle display name                      | 6.0                                 |

B You can prevent your application from multitasking using `<app>-Info.plist`.

**TIP** Apple recommends that all developers implement multitasking in their applications. In fact, it's enabled by default. To prevent your application from multitasking, you must set the **Application does not run in background** key in the `<app>-Info.plist` file of your application B. For more information, see the "Understanding application settings" section in Chapter 3, "Common Tasks."

**TIP** Some older iOS devices don't support multitasking. You can use the **multitasking Supported** property of the `UIDevice` class to determine whether multitasking is available.

## Entering and exiting background mode

If the operating system determines that it needs more memory, it may choose to terminate any background applications. That's why it's important to understand the various phases an application goes through when entering and exiting background mode.

When your application switches to background mode, it goes through three distinct states:

- **Active**—Your application is running. This is where you initiate the switch to background mode (usually by exiting the application).
- **Inactive**—Your application is no longer running and is transitioning to the background or suspended state.
- **Background**—Your application is now either suspended or running one or more background services.

There are some important application delegate methods to be aware of when working with multitasking:

- **applicationDidEnterBackground:** is called when your application becomes inactive and begins to enter background mode. This is a good place for you to save any state information (remember, the user or system may still terminate your application once it's in background mode). It's important to note that you only have approximately five seconds in which to perform any actions here before your application is fully suspended.
- **applicationWillTerminate:** is called if your application is in the background but is *not* suspended, such as if it is monitoring location updates or using background audio. It's important to note that if an application is in the background in *suspended* state and the user (or system) terminates the application, you will have no way of handling this.
- **applicationWillEnterForeground:** is called when your application comes out of background mode and begins to transition into becoming the running application. Your application is still inactive at this time, so this is a good place for you to reverse any changes you might have made when the application entered background mode.
- **applicationDidBecomeActive:** is called when your application becomes active as the current running application.

## Multitasking services

For the majority of applications, simply pausing when entering background mode is enough. However, some applications might need to continue to perform tasks while in background mode. For example, if you were listening to some audio and then switched to check your e-mail, you'd probably still want to be able to hear the audio.

For these situations, iOS 4 provides the following services.

### Task completion

Imagine you have written an application that uploads files to a server. If the user puts your application into background mode, you would want to be able to finish any uploads that were taking place. For this scenario, you can use the task completion service.

Task completion provides you with a small amount of time (about ten minutes) to complete any long-running tasks before your application is suspended.

Long-running tasks are typically started from your application delegate's **applicationDidEnterBackground:** method and must be wrapped in calls to **beginBackgroundTaskWithExpirationHandler:** and **endBackgroundTask.**

For more information on task completion, see "Initiating Background Tasks" in the *iOS Application Programming Guide*.

### Background location

If you are writing an application that needs to monitor a user's location even when running in the background (such as a GPS mapping application), you can use the background location service.

This service is enabled by setting the **Required Background Modes** key in the <app>-Info.plist file C. For more information, see the “Understanding application settings” section in Chapter 3.

Within your application, you can choose to monitor the user’s location in two ways:

- **Standard location changes**—This provides frequently updated information of a user’s location; however, it is not recommended for most applications running in the background since it uses a lot of battery power.
- **Significant location changes**—This uses much less power. When the user’s position changes significantly (determined by the cellular radio of the device), your application will be briefly woken from background mode and given the new location information. Even better, if your application has been terminated, it is relaunched. The location event information will be passed in the **UIApplicationLaunchOptions LocationKey** key of the options dictionary sent to the **application:DidFinishLaunchingWithOptions:** method.

For more information on launch events, see the “Application Startup and Configuration” section in Chapter 3.

Background location tracking can have some privacy concerns because the user may not be aware an application is accessing their location. iOS 4 addresses these concerns in several ways:

- Any application that wants to use location services must first request permission from the user D.
- When an application is accessing location services, an indicator appears in the status bar E.

| Key                                    | Value                               |
|----------------------------------------|-------------------------------------|
| Information Property List              | (13 items)                          |
| Required background modes              | (1 item)                            |
| Item 0                                 | App registers for location updates  |
| Localization native development region | App plays audio                     |
| Bundle display name                    | App registers for location updates  |
| Executable file                        | App provides Voice over IP services |
| Icon file                              |                                     |

C Enabling background location services.



D Users must give permission for applications to use location services.



E An icon in the top right of the toolbar shows that the current application is using location services.



F Applications using location services.



G The background audio controls.

- The user can disable location services on a per-application basis via the system settings. Additionally, an icon appears next to any application that has used location services within the last 24 hours F.

For more information on using location services, see the “About Core Location” section in Chapter 8, “Location and Mapping.”

### Background audio

This service allows your application to continue to play audio while it is in background mode. A set of audio controls is available by launching the task-switching dock and swiping to the right G.

For an example of implementing background audio in your applications, see Chapter 9, “Multimedia.”

### Voice over IP (VoIP)

This service allows VoIP-based applications such as Skype to continue to receive and make calls when running from the background.

### Local notifications

While push notifications were supported in previous versions, iOS 4 lets your application create *local* notifications. A notification can be scheduled to occur at any time in the future, and your application doesn’t even have to be running for the notification to occur.

## To create an application that uses local notifications:

1. Create a new view-based application, saving it as LocalNotificationExample.
2. Open LocalNotificationExampleView Controller.m, uncomment the `viewDidLoad` method, and add the following code:

```
UIButton *btn = [UIButton
→ buttonWithType:UIButtonType
→ RoundedRect];

CGRect buttonRect =
→ CGRectMake(75,50,150,35);

[btn setFrame:buttonRect];

[btn setTitle:@"Create
→ Notification" forState:
→ UIControlStateNormal];

[btn addTarget:self action:
→ @selector(createNotification:)
→ forControlEvents:UIControlEventTouchUpInside
→ TouchDown];

[self.view addSubview:btn];
```

Here you are simply creating a button and adding it to your view.

3. Implement the `createNotification:` method:

```
NSTimeInterval ss = 10;
NSDate *notificationDate =
→ [NSDate dateWithTimeInterval
→ SinceNow:ss];
```

First you create a date on which your notification should occur; in this example, set it to ten seconds from now.

```
UILocalNotification *note =
→ [[UILocalNotification alloc]
→ init];

note.fireDate = notificationDate;
note.timeZone = [NSTimeZone
→ defaultTimeZone];
```

Next you create your notification, set its `fireDate` (the date on which it will occur), and set the time zone.

```
note.alertBody = @"The
→ Notification Body";
note.alertAction = @"View";
note.soundName = UILocal
→ NotificationDefaultSoundName;
note.applicationIconBadgeNumber
→ = 1;
```

Next you set the text that will show in the notification pop-up as well as the label for the button that will launch your application. You also set a sound that will play when the notification appears as well as the badge number of your application icon.

```
NSDictionary *dict =
→ [NSDictionary dictionary
→ WithObject:@"Joe Smith"
→ forKey:@"username"];

note.userInfo = dict;
```

*continues on next page*

You can pass additional information to your application by using the `userInfo` property. Here, you simply pass a string representing a username.

```
[[UIApplication sharedApplication]
→ scheduleLocalNotification:note];
```

Finally, you call `schedule` the notification with the system.

#### 4. Build and run the application.

Tapping the button will cause a notification to be created. If you exit your application and wait, you should see the notification occur  . **Code Listing 11.1** shows the completed application.



 A local notification. Notice the application icon also has a badge.

**Code Listing 11.1** Creating local notifications.

```
#import "LocalNotificationExampleViewController.h"

int badgeNumber;

@implementation LocalNotificationExampleViewController

- (void)createNotification:(id)sender
{
 NSTimeInterval ss = 10;
 NSDate *notificationDate = [NSDate dateWithTimeIntervalSinceNow:ss];

 UILocalNotification *note = [[UILocalNotification alloc] init];
 note.fireDate = notificationDate;
 note.timeZone = [NSTimeZone defaultTimeZone];

 note.alertBody = @“The Notification Body”;
 note.alertAction = @“View”;
 note.soundName = UILocalNotificationDefaultSoundName;

 badgeNumber++;
 note.applicationIconBadgeNumber = badgeNumber;

 NSDictionary *dict = [NSDictionary dictionaryWithObject:@“Joe Smith” forKey:@“username”];
 note.userInfo = dict;

 [[UIApplication sharedApplication] scheduleLocalNotification:note];

 [note release];
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 badgeNumber = 0;

 UIButton *btn = [UIButton buttonWithType:UIButtonTypeRoundedRect];
 CGRect buttonRect = CGRectMake(75,50,150,35);
 [btn setFrame:buttonRect];

 [btn setTitle:@“Create Notification” forState:UIControlStateNormal];
 [btn addTarget:self action:@selector(createNotification:) forControlEvents:UIControlEventTouchDown];

 [self.view addSubview:btn];
}

- (void)dealloc
{
 [super dealloc];
}

@end
```

# Responding to Local Notifications

You've seen how easy to create a local notification, but that's only one half of the puzzle. You also want to be able to respond to a local notification, possibly to capture any information passed in the `userInfo` dictionary.

This is all done by using a couple of the `UIApplication` delegate methods. Next you'll see how to implement this.

## To update your application to respond to local notifications:

1. Open `LocalNotificationExampleApp` `Delegate.m`, and uncomment the `application:didFinishLaunchingWithOptions:` method:

```
UILocalNotification *note =
→ [launchOp objectForKey:
→ UIApplicationLaunchOptionsLocal
→ NotificationKey];
if (note)
 NSLog(@"Notification info:
→ %@",note.userInfo);
```

Here you check the `launchOptions` dictionary for the key `UIApplicationLaunchOptionsLocalNotificationKey`, which will exist only if a local notification has been received. If it exists, you log the `userInfo` dictionary of the local notification to the console.

- 2.** Implement the **applicationDidBecomeActive:** delegate method:

```
application.applicationIconBadge
→ Number = 0;
```

This simply ensures your application badge number disappears (recall you set it when you created your local notification).

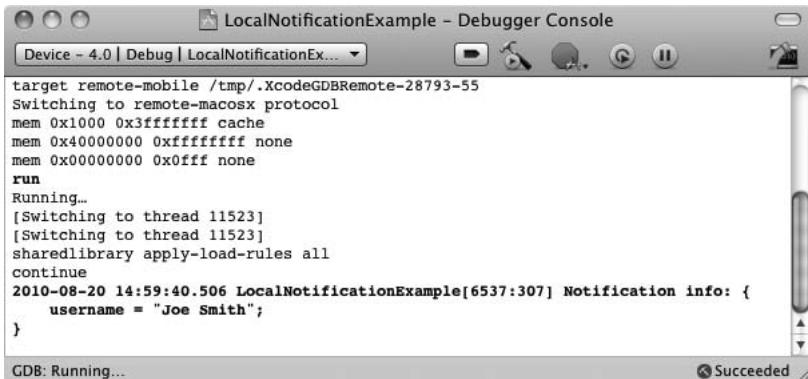
- 3.** Finally, to handle notifications that occur when your application is either running or in background mode, implement the **application:didReceiveLocalNotification:** delegate method.

**Listing 11.2** shows the completed code.

- 4.** Build and run your application.

This time when you create the notification, don't exit the application. If you wait ten seconds, you should see your notification information being sent to the console **A**.

For more information on multitasking, see "Executing Code in the Background" in the *iOS Application Programming Guide*.



The screenshot shows the Xcode Debugger Console window titled "LocalNotificationExample - Debugger Console". The title bar includes "Device - 4.0 | Debug | LocalNotificationEx..." and various control icons. The main pane displays the following log output:

```
target remote-mobile /tmp/.XcodeGDBRemote-28793-55
Switching to remote-macosx protocol
mem 0x1000 0x3fffffff cache
mem 0x40000000 0xffffffff none
mem 0x00000000 0x0fff none
run
Running...
[Switching to thread 11523]
[Switching to thread 11523]
sharedlibrary apply-load-rules all
continue
2010-08-20 14:59:40.506 LocalNotificationExample[6537:307] Notification info: {
 username = "Joe Smith";
}
GDB: Running...
Succeeded
```

**A** Logging the contents of the **userInfo** dictionary.

**Code Listing 11.2** Responding to local notifications.

```
#import "LocalNotificationExampleAppDelegate.h"
#import "LocalNotificationExampleViewController.h"

@implementation LocalNotificationExampleAppDelegate

@synthesize window;
@synthesize viewController;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
 UILocalNotification *note = [launchOptions objectForKey:UIApplicationLaunchOptionsLocalNotificationKey];
 if (note)
 NSLog(@"%@",note.userInfo);
 [window addSubview:viewController.view];
 [window makeKeyAndVisible];
 return YES;
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
 application.applicationIconBadgeNumber = 0;
}

- (void)application:(UIApplication *)application didReceiveLocalNotification:(UILocalNotification *)note {
 NSLog(@"%@",note.userInfo);
}

- (void)dealloc {
 [viewController release];
 [window release];
 [super dealloc];
}

@end
```

# Index

## Symbols

- : (colon), using with methods, 7
- { } (braces), using in header (.h) file, 3
- + (plus) sign, prefixing class methods with, 7
- [ ] (square brackets), using with methods, 5

## A

- ABGroup** type, using with Address Book, 406
- ABPerson** type, using with Address Book, 406
- accelerometer
  - creating tilt-sensitive applications, 307–310
  - detecting shakes, 298–299
  - determining orientation, 299, 301–302
  - redrawing for orientation changes, 303
  - responding to, 307
  - updating for autorotation, 304–306
- accelerometer data, smoothing out, 307
- accessibility
  - enabling via VoiceOver, 98–99
  - overview of, 98
- accessibility attributes
  - Frame, 101
  - Hint, 101
  - Label, 101
  - Traits, 101
  - Value, 101
- Accessibility inspector, using with IB (Interface Builder), 102
- accessible applications, creating, 99–100
- action sheets
  - versus alert views, 145
  - changing appearance of, 145
  - creating, 144–145

## actions

- confirming, 144–145
- creating in IB (Interface Builder)
  - ma manually, 72–75
- activity indicator view, creating, 140–141
- Address Book
  - creating contacts, 413–417
  - creating multivalue properties, 414–415
  - getting reference to, 407
  - group records, 410
  - group records stored in, 406
  - kABPerson\*** properties, 411
  - logging records to console, 407, 409
  - person records, 411–412
  - person records stored in, 406
  - record ID of records, 408
  - retrieving contact records in, 406–408
  - retrieving multivalue properties, 412
  - retrieving records from, 406–409
  - setting values for addresses, 415
  - setting values for phone numbers, 414
- Address Book UI framework
  - adding contacts, 424
  - adding labels for contact names, 423
  - adding to projects, 419
  - displaying contact information, 424
  - editing people, 421–427
  - people picker, 418–420
  - tapping contact images, 424
- AddressBookExampleViewController.h file, 406
- addresses, displaying in map annotations, 339
- alarms, setting in calendar, 432
- alert messages, displaying, 142–143
- alert views versus action sheets, 145

**alloc** method  
calling, 13  
using to create objects, 10

annotation callouts, altering, 337

annotation class, creating for maps, 334

annotations, displaying addresses in, 339

<app>Info.plist file  
contents of, 44–45  
settings, 87–88

Apple Developer Connection Web site, 41, 307

application delegate, using, 84–86

application icons  
key for, 88  
removing “gloss effect” from, 88

application name, displaying, 88

application settings  
adding controls, 92–93  
application preferences, 90  
creating settings pages, 91  
user preferences, 87–90

application setup, setting unique identifier for, 88

applications. *See also* interapp communication;  
projects; settings page  
adding classes to, 57  
adding strings file to, 95–96  
adding views to, 113  
building in Xcode IDE, 58  
creating settings page for, 91  
with custom URL scheme, 104–105  
force-quitting, 90  
launching from other applications, 103  
localizing, 95–97  
making accessible, 99–100  
peer-to-peer, 271–273  
peer-to-peer chat, 273–278  
running in Xcode IDE, 58  
setting preferences for, 90  
setting version numbers for, 88  
sharing information between, 105, 109–110  
shipping with default settings files, 239–243  
terminating in background, 88  
updating in IB (Interface Builder), 70–72  
view-based, 312

arrays  
accessing objects in, 25  
creating, 24  
creating for parsed data, 256  
getting lengths of, 24  
looping back through values of, 25  
mutable, 26  
sorting for strings, 25  
using **@selector** keyword with, 25  
using with drill-down details application, 220  
using with table views, 204–205  
verifying objects in, 25

**articles** array, using with parsed RSS feed, 255

asynchronous connections  
updating applications for, 251–253  
using, 251

audio. *See also* background audio  
controlling from background, 361–365  
creating Play button for, 368  
playing, 398  
playing from iPod library, 399  
playing in background, 358–360  
recording, 366–370

audio controls, accessing, 362

audio events, responding to, 356

audio player application  
adding controls to, 352–356  
checking playing status of, 352  
completing, 354–355  
creating, 350–351  
creating user interface, 352  
implementing delegate methods, 357  
implementing **play:** method, 352  
implementing **scrub:** method, 354  
implementing **stop** method, 352  
preloading buffers, 352  
resetting audio controls, 357  
responding to events, 356–357  
setting volume for, 352–353  
setting **volume** property, 354

audio recorders  
passing **nil** to **settings** parameter, 368, 370  
settings for, 368

audio session, setting up, 359  
audio settings, configuring, 368  
`AudioPlayerExampleViewController.h` file, 350, 352, 356, 358  
`AudioRecorderExample.h` file, 368  
autorelease pools, using in memory management, 11–13  
**autoresizingMask** values, described, 118  
autorotation  
    defined, 303  
    using, 304–306  
**AVAudioPlayer** class  
    behavior of, 351  
    explained, 350  
**AVAudioPlayerDelegate** protocol  
    adopting, 356  
    implementing, 396  
    implementing methods for, 357

## B

background audio, controlling volume of, 362.  
    See *also* audio  
background audio service, 461  
background color  
    changing for orientation, 302  
    setting for movies, 389  
background location service, 459–461. See *also* location manager  
background mode, entering and exiting, 457–458  
battery power, preserving for location manager, 315, 319  
birthdays, creating events for, 431–433  
bookmarks, using in Xcode IDE, 53  
border styles, using with text fields, 156  
bounds, using with views, 113–114  
braces ({ }), using in header (.h) file, 3  
breakpoints  
    adding in Xcode IDE, 52  
    removing in Xcode IDE, 52  
brightness slider, 175  
Build Results window, displaying in Xcode IDE, 58

buttons  
    adding for alert views, 142–143  
    adding to applications, 170–171  
    adding to main view, 184  
    adding to toolbars, 153–156  
checkbox, 172  
    creating for custom cells, 227  
    creating for taking photos and videos, 377  
    predefined, 153  
    specifying images for, 171  
    using target-action pattern with, 37

## C

calendar event store, querying, 429  
calendar events  
    editing, 438–442  
    viewing, 434–437  
calendars, using with dates, 22. See *also* iPhone calendar  
**calendars** property, passing `nil` in, 429  
camel case notation, variation of, 5  
camera application, 378–379  
camera mode, launching image picker in, 378  
camera support, checking for, 376  
capitalization, setting for keyboards, 158  
categories, using as alternative to subclassing, 37–38  
cells, customizing in table views, 224–232  
**CGRect**  
    converting to `NSString`, 114  
    creating for frame of view, 113  
    creating for subview, 117  
    uses of, 286  
checkbox buttons, creating, 172  
class files, adding in Xcode IDE, 57  
class methods  
    autoreleased objects returned by, 6  
    defining, 6  
    prefixing with plus (+) sign, 6  
    using, 6  
classes. See *also* subclassing  
    adding to applications, 57  
arrays, 24–26  
    class methods provided by, 6

classes (*continued*)  
dates and times, 20–23  
dictionaries, 27–29  
header (.h) file, 3  
implementation (.m) file, 4  
methods, 5–7  
with methods for handling files, 234–235  
notifications, 30–31  
saving in Xcode IDE, 57  
strings, 14–20  
timers, 32–34  
clicks, capturing in web views, 169  
**CLLocation** events  
generating, 312  
properties of, 318  
**CLLocationManager** class, delegate methods, 314  
**CLPreviewController** object, using, 244  
Cocoa, defined, 1  
Cocoa Touch Class, adding, 57  
Cocoa Touch framework group, described, 2  
code  
collapsed in Xcode IDE, 52  
commenting in Xcode IDE, 53  
displaying in functional groups, 52  
hiding in Xcode IDE, 52  
uncommenting in Xcode IDE, 53  
code completion, using in Xcode IDE, 53  
code-signing identities, resource for, 79  
colon (:), using with methods, 7  
Command key. See keyboard shortcuts  
compass, accessing in Core Location framework, 323–324  
contact information  
displaying, 424  
editing in Address Book, 425  
contact names, adding labels for, 423  
contact records, retrieving in Address Book, 406–408  
`ContactExampleViewController.h` file, 413  
contacts  
adding to Address Book, 424  
creating for Address Book, 413–417  
grouping in Address Book, 410

Contacts application  
displaying people picker in, 418  
viewing contacts in, 416  
Control key. See keyboard shortcuts  
controls  
buttons, 170–171  
defined, 111  
segmented, 177–179  
sliders, 175–176  
switches, 172–173  
using **UIControl** class with, 170  
**copy** method, calling, 13  
Core Location framework. See *also* location manager  
accessing compass, 323–324  
adding logging of data to screen, 316–317  
adding timeouts, 318–323  
adding to projects, 327  
**CLLocation** events, 312  
**CLLocationManager** class, 312  
decreasing **desiredAccuracy** level, 323  
handling location updates, 314–315  
increasing accuracy, 317–318  
power used by, 315  
testing outside simulator, 315–316  
Core OS framework group, described, 2  
Core Services framework group, described, 2  
`CoreLocationExampleViewController.h` file, 312, 316, 318

## D

data detectors, using with text views, 161  
date and time intervals, calculating, 20–21  
date objects, creating for calendar, 428  
date picker, creating, 151  
dates  
calculating seconds between, 20–21  
comparing, 20–21  
creating, 20  
getting day, month, and year from, 21  
localizing, 94  
setting styles for, 23

Debugger Console  
  displaying location information in, 314  
  logging tap counts to, 286  
  logging touch locations to, 294  
defaults system, reading and writing to, 88  
Delegate design pattern, using, 36  
delegate methods  
  for **CLLocationManager** class, 314  
  defining for picker view, 148  
  implementing for alert view, 143, 162–163  
  implementing for page control, 133  
  implementing for zooming, 131  
  for multitasking, 458  
  using to manage heading updates, 324  
for web pages, 165  
design patterns  
  categories, 37–38  
  Delegate, 36  
  MVC (Model View Controller), 35  
  singletons, 39  
  Target-Action, 37  
`DetailViewController.h` file, 218  
developer, registering with Apple as, 41  
dictionaries  
  accessing objects in, 28  
  creating, 27  
  mutable, 29  
  populating from files, 28  
  verifying number of elements in, 28  
directories  
  **Documents**, 237  
  **Library**, 238  
  **tmp**, 237  
documentation viewer, launching for iOS, 78  
documents, previewing, 244  
**Documents** directory, explained, 237  
double-tap support, adding, 287  
drill-down details application, main header file for, 219

**E**  
editor pane in Xcode IDE, using, 50–51  
**EKEventViewController**, using, 436–437  
e-mail  
  attaching files to, 447  
  composing and sending, 443–449  
  showing compose interface for, 447  
e-mail libraries, using, 444  
`EmailExampleViewController.h` file, 444  
event store  
  adding events to, 432  
  creating for birthdays, 431–433

**F**  
file sharing, enabling, 238  
file system, overview of, 236  
files  
  adding to projects, 51  
  attaching to e-mail, 447  
  classes related to, 234  
  copying, 236  
  creating in Xcode IDE, 57  
  editing in Xcode Organizer, 79  
  finding quickly in projects, 51  
  opening from keyboard, 51  
  reading in application bundles, 238  
`FlipModalExampleViewController.h` file, 189  
focus ribbon in Xcode IDE, explained, 52  
font size, specifying for labels, 136  
fonts, list of, 137  
force-quitting applications, 90  
Frame accessibility attribute, 101  
frames, representing views as, 112  
frameworks  
  adding to projects, 2  
  defined, 2  
  referencing in code, 2  
French localization, displaying, 97  
French strings file, adding to application, 95–96

## G

Game Kit API, using for peer-to-peer applications, 271–273  
Get Info in Xcode IDE, keyboard shortcut, 46–47  
getter methods, generating, 9  
GetWebContentViewController.h file, 248, 251  
graphics  
  adding for load screen, 79  
  adding to tabs, 196  
  using in image views, 126  
groups in Xcode  
  creating, 45  
  static, 44  
gutter in Xcode IDE, explained, 52

## H

header (.h) file  
  **@end** directive, 3  
  **#import** directive, 3  
  **@interface** line, 3  
  use of braces ({} ) in, 3  
heading updates, checking support for, 324  
helloWorld target in Xcode, features of, 46–47  
helloWorldViewController.h file, 72  
helloWorldViewController.xib file  
  construction of, 76–77  
  opening in IB, 65  
Hint accessibility attribute, setting, 100–101  
home directory, contents of, 237  
HTTP authentication, responding to, 266  
hyperlinks  
  handling in web views, 168–169  
  opening in Safari application, 169

## I

IB (Interface Builder)  
  Accessibility inspector, 102  
  actions, 64  
  configuring slider in, 74  
  creating actions manually, 72–75  
  creating interfaces, 69–70  
  creating outlets manually, 72–75  
  displaying class's actions, 70

displaying class's outlets, 70  
document window, 65–66  
features of, 64–65  
File's Owner object, 65  
First Responder object, 66  
increasing width of time label, 73  
inspector window, 67–69  
laying out applications, 69  
Library window, 67  
outlets, 64  
updating applications, 70–72  
using to set styles for table views, 207  
View object, 66  
XIB files, 65  
image picker  
  closing, 374  
  creating application with, 372–375  
  hiding, 374  
  launching in camera mode, 378  
  setting **sourceType** property of, 371–372  
image views. *See also* views  
  controlling scrolling behavior of, 127  
  creating, 126  
  resizing, 126  
  using with scroll views, 129–130  
ImagePickerExample.h file, 372  
images  
  animating, 127  
  animating over, 128  
  choosing from photo library, 371  
  displaying as annotations, 336–337  
  getting paths of, 238  
  panning around, 130  
  specifying for buttons, 171  
  using scroll view with, 129–130  
  zooming in and out of, 130  
implementation (.m) file  
  **@end** directive, 4  
  **@implementation** line, 4  
  **#import** directive, 4  
  **@synthesize** directive, 4  
interapp communication, using **openURL:** method in, 103. *See also* applications  
interface, redrawing when rotating, 303

Interface Builder (IB)  
Accessibility inspector, 102  
actions, 64  
configuring slider in, 74  
creating actions manually, 72–75  
creating interfaces, 69–70  
creating outlets manually, 72–75  
displaying class’s actions, 70  
displaying class’s outlets, 70  
document window, 65–66  
features of, 64–65  
File’s Owner object, 65  
First Responder object, 66  
increasing width of time label, 73  
inspector window, 67–69  
laying out applications, 69  
Library window, 67  
outlets, 64  
updating applications, 70–72  
using to set styles for table views, 207  
View object, 66  
XIB files, 65  
Internet connections, testing for, 252  
iOS SDK (software development kit)  
documentation, 78  
downloading, 41  
frameworks, 2  
iPhone, displaying logging information on, 317  
iPhone calendar. *See also* calendars  
accessing database for, 428  
adding events, 430  
**alarms** event, 430  
**calendar** event, 430  
creating date objects, 428  
creating events for birthdays, 431–433  
editing events, 438–442  
Event Kit UI classes, 434  
**eventIdentifier** event, 430  
**location** event, 430  
**notes** event, 430  
**recurrenceRule** event, 430  
retrieving events from, 428–429  
setting alarms, 432  
**startDate/endDate** event, 430

**title** event, 430  
viewing calendar events, 434–437  
viewing details of events, 436–437  
viewing event details, 434  
iPhone screen, adding logging location data to, 316–317  
iPhone Simulator  
adding photos to, 63  
backing up data on, 63  
features of, 61–63  
limitations of, 61–62  
removing applications from, 63  
resetting, 63  
iPhone vs. iPhone Simulator, building for, 58  
iPhones, identifying via UDID, 79  
iPod library  
accessing media items, 392  
playing audio from, 399  
selecting songs from, 397  
iPodLibraryExample.h file, 396, 399–400  
iPods, identifying via UDID, 79  
iTunes file sharing, support for, 88

## J

JavaScript, executing in web views, 167

## K

keyboard shortcuts  
bookmarks in Xcode IDE, 53  
building applications in Xcode IDE, 58  
code management in Xcode IDE, 52  
documentation viewer, 78  
finding text in Xcode IDE, 53  
force-quitting applications, 90  
Get Info in Xcode IDE, 46–47  
help in Xcode IDE, 53  
jump-to-definition in Xcode IDE, 53  
opening files in windows (Xcode), 51  
Project Find window in Xcode IDE, 53  
Redo in Xcode IDE, 60  
Single File Find in Xcode IDE, 53  
Undo in Xcode IDE, 60  
Xcode IDE, 60

- keyboards
- hiding, 161
  - scrolling interface in response to, 162–163
  - setting capitalization for, 158
  - using, 157–159
- keyboardType** property
- disabling, 158
  - setting, 157
- L**
- Label accessibility attribute, 101
- label text
- adding shadows to, 136
  - aligning, 138
  - displaying lines of, 138
- labels
- controlling wrapping of, 138
  - creating and setting properties of, 136, 138
  - setting line counts for, 138
  - specifying font sizes for, 136
- landscape versus portrait orientation, 186, 303, 305–306. *See also* orientation
- language codes, resource for, 97
- launch image, specifying name of, 88
- launchOptions** values, displaying, 107
- layoutSubviews** method, using with table views, 225
- Library** directory, explained, 238
- links
- handling in web views, 168–169
  - opening in Safari application, 169
- load screen, adding graphic for, 79
- load time, speeding perception of, 79
- loadView** method, explained, 183
- local notifications. *See also* notifications
- creating application for, 462–465
  - responding to, 466–468
  - service, 461
- localization
- dates, 94
  - numbers, 94
  - overview of, 94–95
  - support for, 88
- localized applications, creating, 95–97
- location aware applications, creating, 312–313
- location events
- checking ages of, 320
  - filtering, 317–318
- location manager. *See also* background location service; Core Location framework
- adding timeouts to, 318–323
  - generating, 319
- heading Available** class method, 324
- location search, results of, 323
- location updates, handling, 314–315
- locations
- setting and showing on maps, 328
  - showing in maps, 328
- logging information, displaying on iPhone, 317
- long-touch support, adding, 288–291
- low-memory conditions, handling, 193. *See also* memory management
- M**
- .m (implementation) file
- @end** directive, 4
  - @implementation** line, 4
  - #import** directive, 4
  - @synthesize** directive, 4
- Mail application, launching, 103, 443
- MainWindow.xib, objects for, 76–77
- map center coordinate, creating variable for, 328
- Map Kit framework, adding to projects, 325
- map overlays, creating, 329–330
- mapping application
- adding helper methods, 344
  - adding instance variables to, 342
  - completing, 345–348
  - creating **CGRect** for address view, 343
  - header file for, 342
  - implementing delegates for, 344
- MappingExampleViewController.h file, 325, 330, 338, 341
- maps
- adding annotations to, 333–337
  - adding reverse geocoding to, 338–340

adding to applications, 325–327  
defining regions on, 330  
displaying, 329  
displaying addresses in annotations, 339  
drawing routes on, 330–332  
drawing shapes on, 329–330  
removing annotations from, 337  
showing locations on, 327–328  
zooming into, 328

Maps application, launching and searching, 104

**mapType** property, using to display maps, 329

media collections, accessing, 394–395

Media framework group, described, 2

media items

- accessing in iPod library, 392–393
- metadata properties for, 394
- playing, 398

media picker

- adding, 396–397
- closing, 397

media player, creating, 400–404

media query, console output of, 393–394

**MediaPlayer** framework, adding, 382

memory management. See *also* low-memory conditions

- autorelease pools, 11–13
- referencing counting, 10

**MessageUI** framework, using, 443–444

messaging methods, 5

method calls

- nesting, 7
- performing steps in, 8

methods

- calling, 5–6
- initializer, 8
- passing values into, 5
- phrases used with, 5
- syntax for, 6
- use of square brackets ([ ]) with, 5
- using colon (:) with, 7
- writing, 7

**MKCircle** class, using with maps, 330

**MKMapView**, example of, 326

**MKPolygon** class, using with maps, 330

modal views, displaying, 189–193

Model View Controller (MVC) design pattern, 35

**motion\*** methods, using to detect shakes, 298

movement detection. See accelerometer

movie playback, controlling, 386

movie player, customizing, 387–390

movie player video controller

- completed code, 385
- header file for, 382
- using, 384

movie recording, time limitation of, 375

**MoviePlayerExampleViewController.h** file, 382, 387

movies. *See also* videos

- loading from network locations, 390–391
- setting background color of, 389
- showing activity indicator for, 390

**MPMediaItem** class, explained, 392

**MPMediaPickerController** view controller class, 396

**MPMoviePlayerController:**, using, 387–391

**MPMusicPlayerController** class, 398

multitasking

- delegate methods, 458
- overview of, 456
- preventing, 457
- verifying capability for, 457

multitasking services

- background audio, 461
- background location, 459–461
- local notifications, 461
- task completion, 459
- VoIP (Voice over IP), 461

multi-touch gestures. *See also* touch-based applications

- pinch, 292–294
- rotate, 292–294
- supporting, 292–294
- zoom, 292–294

**MultiTouchExampleViewController.h** file, 292, 295

music players, types of, 398

MVC (Model View Controller) design pattern, 35

**MyCustomCell.h** file, 224

## N

**Name** property, using with settings page, 93  
navigation bar in Xcode IDE, using, 55–56  
navigation controllers, using with table views, 217  
network locations, loading movies from, 390–391  
networking  
  creating peer-to-peer applications, 271–273  
  creating peer-to-peer chat applications, 273–278  
  parsing RSS feeds, 255–261  
  parsing XML, 254  
  responding to HTTP authentication, 266  
  retrieving content from web pages, 248  
  retrieving stock quotes from web pages, 248–251  
  searching Wikipedia, 262–265  
  sending data to web pages, 262  
  updating status on Twitter, 266–271  
  using asynchronous connections, 251–253  
**NIB (NeXT Interface Builder)**, 65  
**NIB Name** property, using with  
  MainWindow.xib, 76  
notifications. See *also* local notifications  
  described, 30  
  registering objects as observers for, 30  
  using, 30–31  
**NSCalendar** class, using, 21  
**NSData** class, using with files, 234  
**NSDate** class, using, 20  
**NSDate** objects, creating, 21  
**NSDateFormatter** class, using, 22–23, 94  
**NSDictionary**  
  loading and saving as file, 239–243  
  using, 27  
  using with files, 234  
**NSFileManager** class, explained, 236  
**NSHomeDirectory()** function, using, 237  
**NSLog()** statements  
  using with asynchronous connections, 252–253  
  using with stock quotes, 250  
**NSMutableArray** class, using, 26

**NSMutableDictionary** class, using, 29  
**NSNotification** object, using, 30  
**NSNumberFormatter** class, using, 94  
**NSString** class  
  containing numbers, 15  
  converting **CGRect** to, 114  
  creating, 14  
  file functions, 18  
  format specifiers, 14  
  immutable quality of, 14  
  initializer methods in, 8  
  **stringWithContentsOfURL:** method, 248  
  using to read and write to URL, 18  
  using with files, 234  
**NSTimer** class, using, 32  
**NSURLConnection** class, using, 251–252  
**NSUserDefaults** class, using, 87, 90  
**NSXMLParser** class, using, 254, 256  
numbers, localizing, 94

## O

Objective-C  
  classes, 3–4  
  creating objects, 7–8  
  defined, 1  
  methods, 5–7  
  properties, 8–9  
objects  
  calling **release** method for, 10–11  
  creating, 7–8  
on/off controls, creating, 172–173  
**openURL:** method, using with **UIApplication** class, 103  
Option key. See keyboard shortcuts  
orientation. See *also* landscape mode versus  
  portrait orientation; view controllers  
  autorotating, 303  
  detecting, 299, 301–302  
  determining for shakes, 299–301  
  Portrait versus Landscape, 186  
  responding to changes in, 184–188  
  specifying for applications, 88  
  tracking changes in, 187  
orientation changes, redrawing interface for, 303

OrientationExampleViewController.h file, 304  
outlets, creating in IB (Interface Builder)  
    manually, 72–75  
Overview toolbar, using in Xcode IDE, 59

## P

P2PExampleViewController.h file, 274  
page control, creating for scroll view, 132–135  
parser delegate methods, implementing, 256  
**parser** variable, using with RSS feed, 255  
passwords, saving in settings file, 239–242  
pasteboard, using, 109–110  
paths  
    getting array of filenames for, 236  
    retrieving for applications, 237  
patterns. *See* design patterns  
PDF viewer, creating, 244–247  
peer picker controller, 272  
peer picker delegate method, implementing, 275  
peer-to-peer applications, creating, 271–273  
peer-to-peer chat application, creating, 273–278  
people picker  
    adding to Address Book, 418–420  
    editing in Address Book, 421–427  
    updates for editing contacts, 426–427  
PeoplePickerExampleViewController.h, 419, 421  
phone numbers  
    dialing, 104  
    setting values in Address Book, 414  
photo library  
    choosing images from, 371  
    determining empty status of, 375  
photos  
    adding to iPhone Simulator, 63  
    taking, 375–380  
picker views. *See also* views  
    creating, 146–148  
    enhancing, 148–150  
pictures. *See* images; photos  
pinch gestures, adding, 295–297  
Play button, creating for audio, 368  
playback queue, using with audio, 398

playlists, retrieving, 395  
plus (+) sign, prefixing class methods with, 6  
portrait versus landscape orientation, 186, 303, 305–306. *See also* orientation  
PostTweetViewController.h file, 266  
PostWebContentController.h file, 262  
predicate, creating for search text, 214  
preferences  
    application, 90  
    user, 87, 89–90  
PreferenceSpecifiers key, using, 91  
progress views, creating, 139–140  
Project Find history, accessing in Xcode IDE, 53  
projects. *See also* applications  
    adding files to, 51  
    adding frameworks to, 2  
    creating in Xcode IDE, 43  
properties  
    defining, 9  
    using getter methods with, 8  
    using setter methods with, 8  
property list files, selecting in Xcode IDE, 50  
provisioning profiles, use of, 79, 82

## Q

QuartzCore framework, adding to long-touch project, 289  
Quick Look framework  
    adding, 244  
    resource for, 246

## R

Record button, creating for audio, 368  
recording settings, controlling, 370  
rectangles. *See* **CGRect**  
referencing counting, using in memory management, 10  
**release** method, calling for objects, 10–11  
responder objects, defined, 280  
**retain** method, calling, 13  
Return key  
    changing text on, 158  
    hiding text field for, 158  
reverse geocoding, adding to maps, 338–340

RootViewController.h file, 201, 204, 210, 218, 225, 434  
rotate gestures, adding, 295–297  
rotating  
    iPhones, 303  
    views, 296–297  
rotation transforms, applying to views, 124–125  
**roundedCornerView** class, creating, 120–122  
routes, drawing on maps, 330–332  
rows  
    grouping into sections and styles, 204  
    indicating for custom cells, 230  
RSS feeds  
    format of XML records for, 254  
parsing, 255–261

## S

Safari application, opening links in, 169  
sandbox, defined, 233  
**saveClick:** method, implementing, 241–242  
scale transforms, using with views, 123–125  
scope highlighting effect, using in Xcode IDE, 52  
screen, adding logging location data to, 316–317  
scroll views  
    adding zoom to, 131  
    paging content of, 131  
    using to zoom in and out of images, 130  
    using with images, 129–130  
search text, creating predicate for, 214  
segmented controls  
    creating, 177–178  
    properties, 179  
    removing, 178  
    styles, 179  
setter methods, generating, 9  
Settings example, header for, 240  
settings file  
    saving password in, 239–242  
    saving username in, 239–242  
settings page  
    adding controls to, 92–93  
    creating for applications, 91  
    setting **Name** property for, 93

setting **Title** property for, 93  
**setView:** method, using with table views, 224  
ShakeExampleViewController.m file, 299  
shakes  
    detecting, 298  
    determining orientation for, 299–301  
    supporting, 299  
Shift key. See keyboard shortcuts  
simulator. See iPhone Simulator  
single-tap support, adding, 287  
singletons, using, 39  
**skpsmtppmessage** class, availability of, 444  
slider, configuring in IB (Interface Builder), 74  
sliders  
    features of, 175  
    implementing, 175–176  
smart groups, creating in Xcode IDE, 48  
SMS (Short Message Service)  
    composing and sending, 450–454  
    creating body of, 452  
SMS application, launching, 103, 450  
SMSExampleViewController.h file, 451  
songs, selecting from iPod library, 397  
splash screens, adding to applications, 86  
square brackets ([]), using with methods, 5  
**startWiggle:** method, using with long-touch support, 289  
status bar  
    choosing display styles for, 88  
    launching applications without, 88  
    leaving visible, 88  
stock quotes, retrieving from web pages, 248–251  
**stopWiggle:** method, using with long-touch support, 289  
string methods, common uses of, 19  
strings  
    combining, 17  
    comparing, 15  
    converting case of, 15  
    creating arrays with substrings, 17  
    creating substrings, 16  
    getting lengths of, 15  
**NSString** class, 14–19

performing case-sensitive comparisons, 15  
replacing substrings in, 17  
searching for substrings in, 17  
trimming characters from, 16  
verifying substrings in, 17  
strings file, adding to applications, 95–96  
**stringWithContentsOfURL:** method, using, 248  
subclassing, alternative to, 37–38. See *also* classes  
substrings. See strings  
switches  
    altering appearance of, 173–174  
    creating, 172–173  
    creating for custom cells, 227  
synchronous connection, explained, 250  
system items  
    availability of, 153  
    types of, 154

## T

tab bar items  
    hiding, 217  
    updating applications for use of, 196–199  
tab view controllers, implementation files for, 198–199  
tab views, using 194–199. See *also* views  
table views. See *also* views  
    cells in, 200  
    creating applications with, 201–203  
    creating arrays for sections of, 204–205  
    creating predicate for search text, 214  
    creating with drill-down behavior, 218–222  
    customizing cells in, 224–232  
    drilling down in, 217  
    editable for searching, 210–216  
    editing, 210  
    elements of, 200  
    grouped, 204–209  
    grouping rows in, 204  
    implementing data sources for, 200  
    implementing delegates for, 200  
    searching, 210  
    setting styles in IB (Interface Builder), 207  
    styles for cells, 223

suppressing delete button, 214  
**UITableView** class, 200  
**UITableViewCell** class, 200  
using, 200  
using navigation controllers with, 217  
using sections with, 204–209  
tabs  
    adding graphics to, 196  
    adding titles to, 196  
    creating applications with, 194–195  
    disabling, 197  
    limiting number of, 195  
    selecting in code, 195  
tap counts, logging to Debugger Console, 286  
tap delay, setting, 288–289  
**tapCount** property, using with **touch** objects, 285  
tappable links, converting data into, 161  
tapping support  
    adding, 285–286  
    single and double, 287–288  
target-action pattern  
    defined, 111  
    using, 37  
targets in Xcode IDE  
    cleaning, 58  
    helloWorld, 46–47  
    using, 46–47  
task completion service, 459  
templates, choosing in Xcode IDE, 42–43  
text fields  
    border styles for, 156  
    creating, 156–158  
    hiding for Return key, 158  
    removing text from, 157  
    resizing automatically, 157  
    restricting content entered into, 159–160  
text views  
    using, 160  
    using data detectors with, 161  
thumb image, setting for **UISlider**, 175  
TiltingExampleViewController.h file, 307  
tilt-sensitive applications, creating, 307–310

time. See date and time intervals; World Clock application  
time picker, creating, 151  
timers  
    creating, 32, 34  
    stopping, 33  
**Title** property, using with settings page, 93  
titles, adding to tabs, 196  
**tmp** directory, explained, 237  
toolbar items, creating, 154–155  
toolbars  
    adding buttons to, 153–156  
    **autoresizingMask** property of, 152  
    creating, 152–153  
    sizing, 154  
    updating in Xcode IDE, 49  
touch locations, logging to Debugger Console, 294  
touch-based applications. *See also* multi-touch gestures  
    adding long-touch support, 288–291  
    creating, 281–283  
    header file of, 281  
    updated header file, 283  
    updating, 283–285  
touch-based events, methods for, 280  
**TouchExampleViewController.h** file, 281, 283, 287, 289  
Traits accessibility attribute, 101  
**tweetClick:** method, implementing, 267  
Twitter, updating status on, 266–271

## U

**UDID** (unique device identifier), 79  
**UIApplication** class, using **openURL:** method with, 103  
**UIApplicationDelegate** class, using, 84  
**UIBarButtonItem** class, using, 153–156  
**UIButton** class, using, 170–171  
**UIControl** class, using, 170  
**UIDatePicker** class, using, 151  
**UIDevice** class, using, 86  
**UIDevice** singleton, using with shakes, 299  
**UIImage** class, using with files, 234

**UIImagePickerController** class, using, 371  
**UIImageView** class, using, 126–127  
**UILabel** class  
    instances of, 136  
    retrieving for switches, 173–174  
**UIPageControl**, using with scroll views, 131  
**UISegmentedControl** class, described, 177  
**UISlider**, setting thumb image of, 175  
**UISwitch** class, using, 172–173  
**UITabBarController** class, using, 194  
**UITableView** class, explained, 200  
**UITextField** class, using, 156  
**UITextView** class  
    using, 160  
    using with stock quotes, 249  
**UIToolbar** class, using, 152–153  
**UITouch** object, explained, 280  
**UIView** class  
    creating subclasses for, 118  
    using, 112  
**UIViewController** class, described, 182  
Unicode Consortium homepage Web site, 97  
unique device identifier (UDID), 79  
URL identifiers, support for, 88  
URL scheme  
    responding to being launched via, 106–108  
    using in interapp communication, 103–105  
user preferences  
    setting, 87, 89  
    storage of, 87  
username, saving in settings file, 239–242

## V

Value accessibility attribute, 101  
version number, setting for applications, 88  
vibrate feature, adding, 143  
video button, availability of, 380  
videos. *See also* movies  
    playing, 381–385  
    taking, 375–380  
view controllers. *See also* orientation  
    displaying modal views, 189–192  
    main views, 182  
    responsibilities of, 182

**UIViewController** class, 182  
view-based applications, creating, 312  
**viewDidAppear** method, explained, 183  
**viewDidDisappear** method, explained, 184  
**viewDidLoad** method  
    explained, 183–184  
    implementing, 241  
**viewDidUnload** method, explained, 183  
views. *See also* image views; picker views; tab views; table views; web views  
activity indicator, 140–141  
adding subviews to, 113, 117–118  
adding to applications, 113  
alert, 142–143, 145  
animating, 289  
animating properties of, 115–117  
applying rotation transform to, 124–125  
applying scale transform to, 124–125  
autosizing, 117–119  
bounds, 113–114  
creating custom rounded-corner, 120–122  
custom drawing, 118  
defined, 111  
locating origins of, 112  
nesting, 112  
overview of, 112  
presenting, 183–184  
presenting modally, 191  
replacing with custom classes, 122  
representing as frames, 112  
resizing for animations, 116  
resizing via scale transform, 123  
rotating, 296–297  
rounded-corner, 120–123  
specifying origins and sizes of, 112  
text, 160  
**viewWillAppear** method, explained, 183  
**viewWillDisappear** method, explained, 184  
VoiceOver  
    enabling over iPhone, 98–99  
    improving descriptions used by, 100–101  
VoIP (Voice over IP) service, 461  
volume, controlling for background audio, 362

## W

Web pages  
    delegate methods for, 165  
    displaying in applications, 164–166  
    retrieving, 251  
    retrieving content from, 248  
    retrieving stock quotes from, 248–251  
    sending data to, 262  
    using Backward buttons in, 166  
    using Forward buttons in, 166  
Web sites  
    Apple Developer Connection, 41, 307  
    code-signing identities, 79  
    documentation for iOS, 78  
    iOS SDK, 41  
    language codes, 97  
    registering as Apple developer, 41  
    **skpsmtppmessage** class, 444  
    Unicode Consortium homepage, 97  
web views. *See also* views  
    capturing clicks in, 169  
    executing JavaScript in, 167  
    handling hyperlinks, 168–169  
    implementing, 166  
    loading local content, 168–169  
Wi-Fi, setting property for, 88  
Wikipedia, searching, 262–265  
World Clock application, 194

## X

Xcode IDE  
    Action toolbar feature, 48  
    adding classes to applications, 57  
    <app>-Info.plist properties, 44–45  
    bookmarks, 53  
    Bookmarks pop-up menu, 56  
    Bookmarks smart group, 45  
    Breakpoints pop-up menu, 56  
    Build and Go toolbar feature, 48  
    building and running applications, 58  
    building for iPhone vs. Iphone Simulator, 58  
    choosing project templates in, 42–43  
    Class hierarchy pop-up menu, 56

- Xcode IDE (*continued*)  
cleaning targets, 59  
code completion, 53  
collapsed code in, 52  
commenting code, 53  
Counterpart pop-up menu, 56  
creating files, 57  
creating groups in, 45  
creating projects in, 43  
creating smart groups, 48  
Debug configuration, 47  
default smart groups in, 45  
details pane, 49–50  
displaying breakpoints in, 52  
displaying errors in, 52  
displaying line numbers in, 52  
displaying warnings in, 52  
editor pane, 50–51  
Errors and Warnings smart group, 45  
Executables smart group, 45  
Find Results smart group, 45  
find-and-replace operations, 53  
Groups & Files pane, 44–45, 47  
gutter and focus ribbon, 52  
help feature, 53  
hiding code in, 52  
Included files pop-up menu, 56  
Info toolbar feature, 48  
jump-to-definition, 53  
keyboard shortcuts, 60  
Lock pop-up menu, 56  
navigation bar, 55–56  
opening files in windows, 51  
overview of, 41–42  
Overview toolbar, 59  
Overview toolbar feature, 48  
Project Find history, 53  
Project Find window, 53  
Redo keyboard shortcut, 60  
Release configuration, 47  
saving classes, 57  
saving projects in, 43  
SCM (source-code management) smart group, 45  
scope highlighting effect, 52  
Search toolbar feature, 48  
selecting property list files, 50  
Single File Find dialog box, 53  
static groups, 44  
targets, 46–47  
Targets smart group, 45  
Tasks toolbar feature, 48  
toolbar features, 48  
uncommenting code, 53  
Undo keyboard shortcut, 60  
updating toolbar in, 49  
Xcode Organizer  
Archived Applications feature, 81  
Developer Profile feature, 82  
Device Logs feature, 82  
Devices section, 80  
editing files in, 80  
iPhone development area, 81–82  
Provisioning Profiles feature, 82  
Screenshots feature, 82  
Share Application feature, 81  
Sharing section, 81  
Software Images feature, 82  
Submit to iTunesConnect feature, 82  
Validate Application feature, 81  
XIB files  
changing view mode of, 66  
displaying two of, 76–77  
using with IB (Interface Builder), 65  
XML, parsing, 254  
XMLEExampleViewController.h file, 255
- ## Z
- zoom gestures, adding, 295–297  
zooming  
adding to scroll view, 131  
enabling, 131  
into maps, 328  
in and out of images, 130

*This page intentionally left blank*