iPad Programming Guide

General

2010-04-13

Contents

Introduction **Introduction 9**

Prerequisites 9
Organization of This Document 9
See Also 10

Chapter 1 **About iPad Development 11**

What is iPad All About? 11
Development Fundamentals 11
    Core Architecture 11
    View Controllers 12
    Graphics and Multimedia 13
    Event Handling 13
    Device Integration Support 13
What's New for iPad Devices? 14
    More Room for Your Stuff 14
    New Elements to Distinguish Your User Interface 14
    Enhanced Support for Text Input and Display 15
    Support for External Displays and Projectors 16
    Formalized Support for Handling Documents and Files 16
    PDF Generation 17

Chapter 2 **Starting Your Project 19**

Creating a Universal Application 19
    Configuring Your Xcode Project 19
    Updating Your Info.plist Settings 20
    Updating Your Views and View Controllers 21
    Adding Runtime Checks for Newer Symbols 21
    Using Runtime Checks to Create Conditional Code Paths 22
    Updating Your Resource Files 22
Using a Single Xcode Project to Build Two Applications 23
Starting from Scratch 24
Important Porting Tip for Using the Media Player Framework 24

Chapter 3 **The Core Application Design 27**

iPad Application Architecture 27
The Application Bundle 29
    New Keys for the Application's Info.plist File 29
    Providing Launch Images for Different Orientations 30

Providing Application Icons for iPad 31
Document Support on iPad Devices 31
Previewing and Opening Files 31
Registering the File Types Your Application Supports 33
Opening Supported File Types 34

Chapter 4 **Views and View Controllers 35**

Designing for Multiple Orientations 35
Creating a Split View Interface 36
Adding a Split View Controller in Interface Builder 38
Creating a Split View Controller Programmatically 38
Supporting Orientation Changes in a Split View 39
Using Popovers to Display Content 39
Creating and Presenting a Popover 42
Implementing a Popover Delegate 43
Tips for Managing Popovers in Your Application 43
Configuring the Presentation Style for Modal Views 44
Making Better Use of Toolbars 46

Chapter 5 **Gesture Recognizers 47**

Gesture Recognizers Simplify Event Handling 47
Recognized Gestures 47
Gestures Recognizers Are Attached to a View 48
Gestures Trigger Action Messages 49
Discrete Gestures and Continuous Gestures 49
Implementing Gesture Recognition 50
Preparing a Gesture Recognizer 50
Responding to Gestures 51
Interacting with Other Gesture Recognizers 52
Requiring a Gesture Recognizer to Fail 52
Preventing Gesture Recognizers from Analyzing Touches 53
Permitting Simultaneous Gesture Recognition 53
Regulating the Delivery of Touches to Views 54
Default Touch-Event Delivery 54
Affecting the Delivery of Touches to Views 55
Creating Custom Gesture Recognizers 55
State Transitions 56
Implementing a Custom Gesture Recognizer 57

Chapter 6 **Graphics and Drawing 61**

Drawing Shapes Using Bezier Paths 61
Bezier Path Basics 61
Adding Lines and Polygons to Your Path 62

Adding Arcs to Your Path 62
Adding Curves to Your Path 63
Creating Oval and Rectangular Paths 64
Modifying the Path Using Core Graphics Functions 65
Rendering the Contents of a Bezier Path Object 66
Doing Hit-Detection on a Path 67
Generating PDF Content 68
Creating and Configuring the PDF Context 69
Drawing PDF Pages 70
Creating Links Within Your PDF Content 72

Chapter 7 **Custom Text Processing and Input 75**

Input Views and Input Accessory Views 75
Simple Text Input 76
Communicating with the Text Input System 77
Overview of the Client Side of Text Input 78
Text Positions and Text Ranges 79
Tasks of a UITextInput Object 79
Tokenizers 80
Facilities for Text Drawing and Text Processing 80
Core Text 80
UIStringDrawing and CATextLayer 82
Core Graphics Text Drawing 83
Foundation-Level Regular Expressions 83
ICU Regular-Expression Support 84
Spell Checking and Word Completion 85
Custom Edit Menu Items 86

Document Revision History 89

Figures, Tables, and Listings

Chapter 3 **The Core Application Design 27**

Figure 3-1	Key objects in an iPad application	27
Table 3-1	The role of objects in an application	28
Table 3-2	New `Info.plist` keys in iOS 3.2	29
Listing 3-1	Document type information for a custom file format	34

Chapter 4 **Views and View Controllers 35**

Figure 4-1	A split view interface	37
Figure 4-2	Using a popover to display a master pane	41
Figure 4-3	Modal presentation styles	45
Listing 4-1	Creating a split view controller programmatically	39
Listing 4-2	Presenting a popover	42

Chapter 5 **Gesture Recognizers 47**

Figure 5-1	Path of touch objects when gesture recognizer is attached to a view	48
Figure 5-2	Discrete versus continuous gestures	49
Figure 5-3	Possible state transitions for gesture recognizers	56
Table 5-1	Gestures recognized by the gesture-recognizer classes of the UIKit framework	47
Listing 5-1	Creating and initializing discrete and continuous gesture recognizers	50
Listing 5-2	Handling pinch, pan, and double-tap gestures	51
Listing 5-3	Implementation of a "checkmark" gesture recognizer.	58
Listing 5-4	Resetting a gesture recognizer	59

Chapter 6 **Graphics and Drawing 61**

Figure 6-1	An arc in the default coordinate system	63
Figure 6-2	Curve segments in a path	64
Figure 6-3	Workflow for creating a PDF document	68
Figure 6-4	Creating a link destination and jump point	73
Listing 6-1	Creating a pentagon shape	62
Listing 6-2	Creating a new arc path	63
Listing 6-3	Assigning a new `CGPathRef` to a `UIBezierPath` object	65
Listing 6-4	Mixing Core Graphics and `UIBezierPath` calls	65
Listing 6-5	Drawing a path in a view	66
Listing 6-6	Testing points against a path object	67
Listing 6-7	Creating a new PDF file	69
Listing 6-8	Drawing page-based content	71

Chapter 7 **Custom Text Processing and Input 75**

Figure 7-1	Paths of communication with the text input system	78
Figure 7-2	Architecture of the Core Text layout engine	81
Figure 7-3	An editing menu with a custom menu item	87
Table 7-1	ICU files included in iOS 3.2	84
Listing 7-1	Creating an input accessory view programmatically	76
Listing 7-2	Implementing simple text entry	77
Listing 7-3	Finding a substring using a regular expression	83
Listing 7-4	Spell-checking a document	85
Listing 7-5	Presenting a list of word completions for the current partial string	86
Listing 7-6	Implementing a Change Color menu item	87

Introduction

The introduction of iPad creates new opportunities for application development using iOS. Because it runs iOS, an iPad is capable of running all of the same applications already being written for iPhone and iPod touch. However, the larger screen size of iPad also means that there are now new opportunities for you to create applications that go beyond what you might have done previously.

This document introduces the new features available for iPad and shows you how to use those features in your applications. However, just because a feature is available does not mean that you have to use it. As a result, this document also provides guidance about when and how you might want to use any new features in order to help you create compelling applications for your users.

Prerequisites

Before reading this document, you should already be familiar with the development process for iPhone applications. The process for developing iPad applications and iPhone applications is very similar and so should be considered a starting point. If you need information about the architecture or development process for iPhone applications (and iPad applications by extension), see *iOS Application Programming Guide*.

Organization of This Document

This document contains the following chapters:

- "About iPad Development" (page 11) provides an introduction to the platform, including information about new features you can include in your iPad applications.
- "Starting Your Project" (page 19) explains the options for porting iPhone applications and shows you how to set up your Xcode projects to support iPad development.
- "The Core Application Design" (page 27) describes the basic application architecture for iPad along with information about how you use some new core features.
- "Views and View Controllers" (page 35) describes the new interface elements for the platform and provides examples of how you use them.
- "Gesture Recognizers" (page 47) describes how to use the new gesture-recognizer technology to process touch events and trigger actions.
- "Graphics and Drawing" (page 61) describes how to use the new drawing-related technologies.
- "Custom Text Processing and Input" (page 75) describes new text-related features and explains how you can better incorporate text into your application.

See Also

To develop iPad applications, you use many of the same techniques and processes that you use to develop iPhone applications. If you are unfamiliar with the design process for iPhone applications, you should refer to the following documents for more information:

- For information about the general architecture of an iPad application, see *iOS Application Programming Guide*.
- For information about view controllers and the crucial role they play in implementing your application infrastructure, see *View Controller Programming Guide for iOS*.
- For information about the human interface guidelines you should follow when implementing your iPad application, see *iPad Human Interface Guidelines*.

About iPad Development

This chapter provides an introduction to the iPad family of devices, orienting you to the basic features available on the devices and what it takes to develop applications for them. If you have already written an iPhone application, writing an iPad application will feel very familiar. Most of the basic features and behaviors are the same. However, iOS 3.2 includes features specific to iPad devices that you will want to use in your applications.

What is iPad All About?

With iPad devices, you now have an opportunity to create Multi-Touch applications on a larger display than previously available. The 1024 x 768 pixel screen provides much more room to display content, or provide greater detail for your existing content. And the addition of new interface elements in iOS 3.2 enable an entirely new breed of applications.

The size and capabilities of iPad mean that it is now possible to create a new class of applications for a portable device. The increased screen size gives you the space you need to present almost any kind of content. The Multi-Touch interface and support for physical keyboards enables diverse modes of interaction, ranging from simple gesture-driven interactions to content creation and substantial text input.

The increased screen size also makes it possible to create a new class of immersive applications that replicate real-world objects in a digital form. For example, the Contacts and Calendar applications on iPad look more like the paper-based address book and calendar you might have on your desk at home. These digital metaphors for real-life objects provide a more natural and familiar experience for the user and can make your applications more compelling to use. But because they are digital, you can go beyond the limitations of the physical objects themselves and create applications that enable greater productivity and convenience.

Development Fundamentals

If you are already familiar with the process for creating iPhone applications, then the process for creating iPad applications will feel very familiar. For the most part, the high-level process is the same. All iPhone and iPad devices run iOS and use the same underlying technologies and design techniques. Where the two devices differ most are in screen size, which in turn may affect the type of interface you create for each. Of course, there are also some other subtle differences between the two, and so the following sections provide an overview of some key system features for iPad devices along with information about places where those features differ from iPhone devices.

Core Architecture

With only minor exceptions, the core architecture of iPad applications is the same as it is for iPhone applications. At the system level:

- Only one application runs at a time and that application's window fills the entire screen.
- Applications are expected to launch and exit quickly.
- For security purposes, each application executes inside a sandbox environment. The sandbox includes space for application-specific files and preferences, which are backed up to the user's computer. Interactions with other applications on a device are through system-provided interfaces only.
- Each application runs in its own virtual memory space but the amount of usable virtual memory is constrained by the amount of physical memory. In other words, memory is not paged to and from the disk.
- Custom plug-ins and frameworks are not supported.

Inside an application, the following behaviors apply:

- (**New**) An application's interface should support all landscape and portrait orientations. This behavior differs slightly from the iPhone, where running in both portrait and landscape modes is not required. For more information, see "Designing for Multiple Orientations" (page 35).
- Applications are written in Objective-C primarily but C and C++ may be used as well.
- All of the classes available for use in iPhone applications are also available in iPad applications. (Classes introduced in iOS 3.2 are not available for use in iPhone applications.)
- Memory is managed using a retain/release model.
- Applications may spawn additional threads as needed. However, view-based operations and many graphics operations must always be performed on the application's main thread.

All of the fundamental design patterns that you are already familiar with for iPhone applications also apply to iPad applications. Patterns such as delegation and protocols, Model-View-Controller, target-action, notifications, and declared properties are all commonly used in iPad applications.

If you are unfamiliar with the basics of developing iPhone applications, you should read *iOS Application Programming Guide* before continuing. For additional information about the fundamental design patterns used in all Cocoa Touch applications, see *Cocoa Fundamentals Guide*

View Controllers

Just as they are for iPhone applications, view controllers are a crucial piece of infrastructure for managing and presenting the user interface of your iPad application. A view controller is responsible for a single view. Most of the time, a view controller's view is expected to fill the entire span of the application window. In some cases, though, a view controller may be embedded inside another view controller (known as a container view controller) and presented along with other content. Navigation and tab bar controllers are examples of container view controllers. They present a mixture of custom views and views from their embedded view controllers to implement complex navigation interfaces.

In iPad applications, navigation and tab bar controllers are still supported and perfectly acceptable to use but their importance in creating polished interfaces is somewhat diminished. For simpler data sets, you may be able to replace your navigation and tab bar controllers with a new type of view controller called a split view controller. Even for more complex data sets, navigation and tab bar controllers often play only a secondary role in your user interface, providing lower-level navigation support only.

For specific information about new view controller-related behaviors in iOS 3.2, see "Views and View Controllers" (page 35).

Graphics and Multimedia

All of the graphics and media technologies you use in your iPhone applications are also available to iPad applications. This includes native 2D drawing technologies such as Core Graphics, UIKit, and Core Animation. You can also use OpenGL ES 2.0 or OpenGL ES 1.1 for drawing 2D and 3D content.

Using OpenGL ES on iPad is identical to using OpenGL ES on other iOS devices. An iPad is a PowerVR SGX device and supports the same basic capabilities as other SGX devices. However, because the processor, memory architecture, and screen dimensions are different for iPad, you should always test your code on an iPad device before shipping to ensure performance meets your requirements.

All of the same audio technologies you have used in iOS previously are also available in your iPad applications. You can use technologies such as Core Audio, AV Foundation, and OpenAL to play high-quality audio through the built-in speaker or headphone jack. You can also play tracks from the user's iPod library using the classes of the Media Player framework.

If you want to incorporate video playback into your application, you use the classes in the Media Player framework. In iOS 3.2, the interface for playing back video has changed significantly, providing much more flexibility. Rather than always playing in full-screen mode, you now receive a view that you can incorporate into your user interface at any size. There is also more direct programmatic control over playback, including the ability to seek forwards and backwards in the track, set the start and stop points of the track, and even generate thumbnail images of video frames.

For information on how to port existing Media Player code to use the new interfaces, see "Important Porting Tip for Using the Media Player Framework" (page 24). For more information on the hardware capabilities of OpenGL ES, along with how to use it in iOS applications, see *OpenGL ES Programming Guide for iOS*.

Event Handling

The Multi-Touch technology is fundamental to both iPhone and iPad applications. Like iPhone applications, the event-handling model for iPad applications is based on receiving one or more touch events in the views of your application. Your views are then responsible for translating those touch events into actions that modify or manipulate your application's content.

Although the process for receiving and handling touch events is unchanged for iPad applications, iOS 3.2 now provides support for detecting gestures in a uniform manner. Gesture recognizers simplify the interface for detecting swipe, pinch, and rotation gestures, among others, and using those gestures to trigger additional behavior. You can also extend the basic set of gesture recognizer classes to add support for custom gestures your application uses.

For more information about how to use gesture recognizers, see "Gesture Recognizers" (page 47).

Device Integration Support

Many of the distinguishing features of iPhone are also available on iPad. Specifically, you can incorporate support for the following features into your iPad applications:

- Accelerometers
- Core Location
- Maps (using the MapKit framework)

- Preferences (either in app or presented from the Settings application).
- Address Book contacts
- External hardware accessories
- Peer-to-peer Bluetooth connectivity (using the Game Kit framework)

Although iPad devices do not include a camera, you can still use them to access the user's photos. The image picker interface supports selecting images from the photo library already on the device.

What's New for iPad Devices?

Although there are many similarities between iPhone and iPad applications, there are new features available for iPad devices that make it possible to create dramatically different types of applications too. These new features may warrant a rethinking of your existing iPhone applications during the porting process. The advantage of using these new features is that your application will look more at home on an iPad device.

More Room for Your Stuff

The biggest change between an iPhone application and an iPad application is the amount of screen space available for presenting content. The screen size of an iPad device measures 1024 by 768 pixels. How you adapt your application to support this larger screen will depend largely on the current implementation of your existing iPhone application.

For immersive applications such as games where the application's content already fills the screen, scaling your application is a good strategy. When scaling a game, you can use the extra pixels to increase the amount of detail for your game environment and the objects within it. With extra space available, you should also consider adding new controls or status displays to the game environment. If you factor your code properly, you might be able to use the same code for both types of device and simply increase the amount of detail when rendering on iPad.

For productivity applications that use standard system controls to present information, you are almost certainly going to want to replace your existing views with new ones designed to take advantage of iPad devices. Use this opportunity to rethink your design. For example, if your application uses a navigation controller to help the user navigate a large data set, you might be able to take advantage of some of the new user interface elements to present that data more efficiently.

New Elements to Distinguish Your User Interface

To support the increased screen space and new capabilities offered by iPad, iOS 3.2 includes some new classes and interfaces:

- **Split views** are a way to present two custom views side-by-side. They are a good supplement for navigation-based interfaces and other types of master-detail interfaces.
- **Popovers** layer content temporarily on top of your existing views. You can use them to implement tool palettes, options menus, and present other kinds of information without distracting the user from the main content of your application.

- Modally presented controllers now support a configurable **presentation style**, which determines whether all or only part of the window is covered by the modal view.
- Toolbars can now be positioned at the top and bottom of a view. The increased screen size also makes it possible to include more items on a toolbar.
- Responder objects now support **custom input views**. A custom input view is a view that slides up from the bottom of the screen when the object becomes the first responder. Previously, only text fields and text views supported an input view (the keyboard) and that view was not changeable. Now, you can associate an input view with any custom views you create. For information about specifying a custom input view, see "Input Views and Input Accessory Views" (page 75).
- Responders can also have a custom **input accessory view**. An input accessory view attaches itself to the top of a responder's input view and slides in with the input view when the object becomes first responder. The most common use for this feature is to attach custom toolbars or other views to the top of the keyboard. For information about specifying a custom input accessory view, see "Input Views and Input Accessory Views" (page 75).

As you think about the interface for your iPad application, consider incorporating the new elements whenever appropriate. Several of these elements offer a more natural way to present your content. For example, split views are often a good replacement (or supplement) to a navigation interface. Others allow you to take advantage of new features and to extend the capabilities of your application.

For detailed information on how to use split views, popovers, and the new modal presentation styles, see "Views and View Controllers" (page 35). For information on input views and input accessory views, see "Custom Text Processing and Input" (page 75). For guidance on how to design your overall user interface, see *iPad Human Interface Guidelines*.

Enhanced Support for Text Input and Display

In earlier versions of iOS, text support was optimized for simple text entry and presentation. Now, the larger screen of iPad makes more sophisticated text editing and presentation possible. In addition, the ability to connect a physical keyboard to an iPad device enables more intense text entry. To support enhanced text entry and presentation, iOS 3.2 also includes several new features that you can use in your applications:

- The Core Text framework provides support for sophisticated text rendering and layout.
- The UIKit framework includes several enhancements to support text, including:
  - New protocols that allow your own custom views to receive input from the system keyboard
  - A new `UITextChecker` class to manage spell checking
  - Support for adding custom commands to the editing menu that is managed by the `UIMenuController` class
- Core Animation now includes the `CATextLayer` class, which you can use to display text in a layer.

These features give you the ability to create everything from simple text entry controls to sophisticated text editing applications. For example, the ability to interact with the system keyboard now makes it possible for you to create custom text views that handle everything from basic input to complex text selection and editing behaviors. And to draw that text, you now have access to the Core Text framework, which you can use to present your text using custom layouts, multiple fonts, multiple colors, and other style attributes.

For more information about how you use these technologies to handle text in your applications, see "Custom Text Processing and Input" (page 75).

Support for External Displays and Projectors

An iPad can now be connected to an external display through a supported cable. Applications can use this connection to present content in addition to the content on the device's main screen. Depending on the cable, you can output content at up to a 720p (1280 x 720) resolution. A resolution of 1024 by 768 resolution may also be available if you prefer to use that aspect ratio.

To display content on an external display, do the following:

1. Use the `screens` class method of the `UIScreen` class to determine if an external display is available.
2. If an external screen is available, get the screen object and look at the values in its `availableModes` property. This property contains the configurations supported by the screen.
3. Select the `UIScreenMode` object corresponding to the desired resolution and assign it to the `currentMode` property of the screen object.
4. Create a new window object (`UIWindow`) to display your content.
5. Assign the screen object to the `screen` property of your new window.
6. Configure the window (by adding views or setting up your OpenGL ES rendering context).
7. Show the window.

Important: You should always assign a screen object to your window before you show that window. Although you can change the screen while a window is already visible, doing so is an expensive operation and not recommended.

Screen mode objects identify a specific resolution supported by the screen. Many screens support multiple resolutions, some of which may include different pixel aspect ratios. The decision for which screen mode to use should be based on performance and which resolution best meets the needs of your user interface. When you are ready to start drawing, use the bounds provided by the `UIScreen` object to get the proper size for rendering your content. The screen's bounds take into account any aspect ratio data so that you can focus on drawing your content.

If you want to detect when screens are connected and disconnected, you can register to receive screen connection and disconnection notifications. For more information about screens and screen notifications, see *UIScreen Class Reference*. For information about screen modes, see *UIScreenMode Class Reference*.

Formalized Support for Handling Documents and Files

To support the ability to create productivity applications, iOS 3.2 includes several new features aimed at support the creation and handling of documents and files:

- Applications can now register themselves as being able to open specific types of files. This support allows applications that do need to work with files (such as email programs) the ability to pass those files to other applications.
- The UIKit framework now provides the `UIDocumentInteractionController` class for interacting with files of unknown types. You can use this class to preview files, copy their contents to the pasteboard, or pass them to another application for opening.

Of course, it is important to remember that although you can manipulate files in your iPad applications, files should never be a focal part of your application. There are no open and save panels in iOS for a very good reason. The save panel in particular implies that it is the user's responsibility to save all data, but this is not the model that iPhone applications should ever use. Instead, applications should save data incrementally to prevent the loss of that data when the application quits or is interrupted by the system. To do this, your application must take responsibility for managing the creation and saving the user's content at appropriate times.

For more information on how to interact with documents and files, see "The Core Application Design" (page 27).

PDF Generation

In iOS 3.2, UIKit introduces support for creating PDF content from your application. You can use this support to create PDF files in your application's home directory or data objects that you can incorporate into your application's content. Creation of the PDF content is simple because it takes advantage of the same native drawing technologies that are already available. After preparing the PDF canvas, you can use UIKit, Core Graphics, and Core Text to draw the text and graphics you need. You can also use the PDF creation functions to embed links in your PDF content.

For more information about how to use the new PDF creation functions, see "Generating PDF Content" (page 68).

Starting Your Project

The process for creating an iPad application depends on whether you are creating a new application or porting an existing iPhone application. If you are creating a new application, you can use the Xcode templates to get started. However, because it runs iOS, it is possible to port existing iPhone applications so that they run natively on an iPad, and not in compatibility mode. Porting an existing application requires making some modifications to your code and resources to support iPad devices, but with well-factored applications, the work should be relatively straightforward. Xcode also makes the porting process easier by automating much of the setup process for your projects.

If you do decide to port an existing iPhone application, you should consider both how you want to deliver the resulting applications and what is the best development process for you. The following table lists the possible porting approaches and what each one involves.

- Create a universal application that is optimized for all device types.
- Use a single Xcode project to create two separate applications: one for iPhone and iPod touch devices and one for iPad devices.
- Use separate Xcode projects to create applications for each type of device.

Apple highly recommends creating a universal application or a single Xcode project. Both techniques enable you to reuse code from your existing iPhone application. Creating a universal application allows you to sell one application that supports all device types, which is a much simpler experience for users. Of course, creating two separate applications might require less development and testing time than a universal application.

Creating a Universal Application

A universal application is a single application that runs optimized for iPhone, iPod touch, and iPad devices. Creating such a binary simplifies the user experience considerably by guaranteeing that your application can run on any device the user owns. Creating such a binary does involve a little more work on your part though. Even a well-factored application requires some work to run cleanly on both types of devices.

The following sections highlight the key changes you must make to an existing application to ensure that it runs natively on any type of device.

Configuring Your Xcode Project

The first step to creating a universal application is to configure your Xcode project. If you are creating a new project, you can create a universal application using the Window-based application template. If you are updating an existing project, you can use Xcode's Upgrade Current Target for iPad command to update your project:

1. Open your Xcode project.

2. In the Targets section, select the target you want to update to a universal application.
3. Select Project > Upgrade Current Target for iPad and follow the prompts to create one universal application.

   Xcode updates your project by modifying several build settings to support both iPhone and iPad.

Important: You should always use the Upgrade Current Target for iPad command to migrate existing projects. Do not try to migrate files manually.

The main change that is made to your project is to set the Targeted Device Family build setting to iPhone/Pad. The Base SDK of your project is also typically changed to iPhone Device 3.2 if that is not already the case. (You must develop with the 3.2 SDK to target iPad.) The deployment target of your project should remain unchanged and should be an earlier version of the SDK (such as 3.1.3) so that your application can run on iPhone and iPod touch devices.

In addition to updating your build settings, Xcode also creates a new main nib file to support iPad. Only the main nib file is transitioned. You must create new nib files for your application's existing view controllers. Your application's `Info.plist` is also updated to support the loading of the new main nib file when running on iPad.

When running on iOS 3.1.3 or earlier, your application must not use symbols introduced in iOS 3.2. For example, an application trying to use the `UISplitViewController` class while running in iOS 3.1 would crash because the symbol would not be available. To avoid this problem, your code must perform runtime checks to see if a particular symbol is available before using it. For information about how to perform the needed runtime checks, see "Adding Runtime Checks for Newer Symbols" (page 21).

Updating Your Info.plist Settings

Most of the existing keys in your `Info.plist` should remain the same to ensure that your application behaves properly on iPhone and iPod touch devices. However, you should add the `UISupportedInterfaceOrientations` key to your `Info.plist` to support iPad devices. Depending on the features of your application, you might also want to add other new keys introduced in iOS 3.2.

If you need to configure your iPad application differently from your iPhone application, you can specify device-specific values for `Info.plist` keys in iOS 3.2 and later. When reading the keys of your `Info.plist` file, the system interprets each key using the following pattern:

key_root-*<platform>*~*<device>*

In this pattern, the *key_root* portion represents the original name of the key. The *<platform>* and *<device>* portions are both optional endings that you can use to apply keys to specific platforms or devices. Specifying the string `iphoneos` for the platform indicates the key applies to all iOS applications. (Of course, if you are deploying your application only to iOS anyway, you can omit the platform portion altogether.) To apply a key to a specific device, you can use one of the following values:

- `iphone` - The key applies to iPhone devices.
- `ipod` - The key applies to iPod touch devices.
- `ipad` - The key applies to iPad devices.

For example, to indicate that you want your application to launch in a portrait orientation on iPhone and iPod touch devices but in landscape-right on iPad, you would configure your `Info.plist` with the following keys:

```
<key>UIInterfaceOrientation</key>
<string>UIInterfaceOrientationPortrait</string>
<key>UIInterfaceOrientation~ipad</key>
<string>UIInterfaceOrientationLandscapeRight</string>
```

For more information about the keys supported for iPad applications, see "New Keys for the Application's Info.plist File" (page 29).

Updating Your Views and View Controllers

Of all the changes you must make to support both iPad and iPhone devices, updating your views and view controllers is the biggest. The different screen sizes mean that you may need to completely redesign your existing interface to support both types of device. This also means that you must create separate sets of view controllers (or modify your existing view controllers) to support the different view sizes.

For views, the main modification is to redesign your view layouts to support the larger screen. Simply scaling existing views may work but often does not yield the best results. Your new interface should make use of the available space and take advantage of new interface elements where appropriate. Doing so is more likely to result in an interface that feels more natural to the user and not just an iPhone application on a larger screen.

Some additional things you must consider when updating your view and view controller classes include:

- **For view controllers:**
  - ❏ If your view controller uses nib files, you must specify different nib files for each device type when creating the view controller.
  - ❏ If you create your views programmatically, you must modify your view-creation code to support both device types.
- **For views:**
  - ❏ If you implement the `drawRect:` method for a view, your drawing code needs to be able to draw to different view sizes.
  - ❏ If you implement the `layoutSubviews` method for a view, your layout code must be adaptable to different view sizes.

For information about integrating some of the views and view controllers introduced in iOS 3.2, see "Views and View Controllers" (page 35).

Adding Runtime Checks for Newer Symbols

Any code that uses symbols introduced in iOS 3.2 must be protected by runtime checks to verify that those symbols are available. These checks allow you to determine if newer features are available in the system and give you the opportunity to follow alternate code paths if they are not. Failure to include such checks will result in crashes when your application runs on iOS 3.1 or earlier.

There are several types of checks that you can make:

- For classes introduced in iOS 3.2, you can use the `NSClassFromString` function to see if the class is defined. If the function returns a non-`nil` value, you may use the class. For example:

  ```
  Class splitVCClass = NSClassFromString(@"UISplitViewController");
  if (splitVCClass)
  {
     UISplitViewController* mySplitViewController = [[splitVCClass alloc] init];
     // Configure the split view controller.
  }
  ```

- To determine if a method is available on an existing class, use the `instancesRespondToSelector:` class method.
- To determine if a function is available, perform a Boolean comparison of the function name to `NULL`. If the result is `YES`, you can use the function. For example:

  ```
  if (UIGraphicsBeginPDFPage != NULL)
  {
      UIGraphicsBeginPDFPage();
  }
  ```

For more information and examples of how to write code that supports multiple deployment targets, see *SDK Compatibility Guide*.

Using Runtime Checks to Create Conditional Code Paths

If your code needs to follow a different path depending on the underlying device type, you can use the `userInterfaceIdiom` property of `UIDevice` to determine which path to take. This property provides an indication of the style of interface to create: iPad or iPhone. Because this property is available only in iOS 3.2 and later, you must determine if it is available before calling it. The simplest way to do this is to use the `UI_USER_INTERFACE_IDIOM` macro as shown below:

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
{
     // The device is an iPad running iPhone 3.2 or later.
}
else
{
     // The device is an iPhone or iPod touch.
}
```

Updating Your Resource Files

Because resource files are generally used to implement your application's user interface, you need to make the following changes:

- In addition to the `Default.png` file displayed when your application launches on iPhone devices, you must add new launch images for iPad devices as described in "Providing Launch Images for Different Orientations" (page 30).
- If you use images, you may need to add larger (or higher-resolution) versions to support iPad devices.

- If you use nib files, you need to provide a new set of nib files for iPad devices.
- Your application icons must be sized appropriately for iPad, as described in "Providing Application Icons for iPad" (page 31).

When using different resource files for each platform, you can conditionally load those resources just like you would conditionally execute code. For more information about how to use runtime checks, see "Using Runtime Checks to Create Conditional Code Paths" (page 22).

Using a Single Xcode Project to Build Two Applications

Maintaining a single Xcode project for both iPhone and iPad development simplifies the development process tremendously by allowing you to share code between two separate applications. The Project menu in Xcode includes a new Upgrade Current Target for iPad command that makes it easy to add a target for iPad devices to your existing iPhone project. To use this command, do the following:

1. Open the Xcode project for your existing iPhone application.
2. Select the target for your iPhone application.
3. Select Project > Upgrade Current Target for iPad and follow the prompts to create two device-specific applications.

Important: You should always use the Upgrade Current Target for iPad command to migrate existing projects. Do not try to migrate files manually.

The Upgrade Current Target for iPad command creates a new iPad target and creates new nib files for your iPad project. The nib files are based on the existing nib files already in your project but the windows and top-level views in those nib files are sized for the iPad screen. Although the top-level views are resized, the command does not attempt to modify the size or position of any embedded subviews, instead leaving your view layout essentially the same as it was. It is up to you to adjust the layout of those embedded views.

Creating a new target is also just the first step in updating your project. In addition to adjusting the layout of your new nib files, you must update your view controller code to manage those nib files. In nearly all cases, you will want to define a new view controller class to manage the iPad version of your application interface, especially if that interface is at all different from your iPhone interface. You can use conditional compilation (as shown below) to coordinate the creation of the different view controllers. If you make few or no changes to your view hierarchy, you could also reuse your existing view controller class. In such a situation, you would similarly use conditional compilation to initialize your view controller with the appropriate nib file for the underlying device type.

The following example includes the iPad view controller code if the Base SDK of the target is set to iPhone Device 3.2 or later. Because the Base SDK for your iPhone application target would be set to an earlier version of the operating system, it would use the `#else` portion of the code.

```
#if __IPHONE_OS_VERSION_MAX_ALLOWED >= 30200
  MyIPadViewController* vc;
  // Create the iPad view controller
#else
  MyIPhoneViewController* vc;
```

```
   // Create the iPhone view controller
#endif
```

In addition to your view controllers, any classes that are shared between iPhone and iPad devices need to include conditional compilation macros to isolate device-specific code. Although you could also use runtime checks to determine if specific classes or methods were available, doing so would only increase the size of your executable by adding code paths that would not be followed on one device or the other. Letting the compiler remove this code helps keep your code cleaner.

Beyond conditionally compiling your code for each device type, you should feel free to incorporate whatever device-specific features you feel are appropriate. The other chapters in this document all describe features that are supported only on iPad devices. Any code you write using these features must be run only on iPad devices.

For more information on using conditional compilation and the availability macros, see *SDK Compatibility Guide*.

Starting from Scratch

Creating an iPad application from scratch follows the same process as creating an iPhone application from scratch. The most noticeable difference is the size of views you create to present your user interface. If you have an idea for a new application, then the decision to start from scratch is obvious. However, if you have an existing iPhone application and are simply unsure about whether you should leverage your existing Xcode project and resources to create two versions of your application, or a universal application supporting all device types, then ask yourself the following questions:

- Are your application's data model objects tightly integrated with the views that draw them?
- Are you planning to add significantly more features to the iPad version of your application?
- Is your application device-specific enough that porting would require changing large amounts of your code?

If you answered yes to any of the preceding questions, then you should consider creating a separate Xcode project for iPad devices. If you have to rewrite large portions of your code anyway, then creating a separate Xcode project is generally simpler. Creating a separate project gives you the freedom to tailor your code for iPad devices without having to worry about whether that code runs on other devices.

Important Porting Tip for Using the Media Player Framework

If you are porting an application that uses the `MPMoviePlayerController` class of the Media Player framework, you must change your code if you want it to run in iOS 3.2. The old version of this class supports only full-screen playback using a simplified interface. The new version supports both full- and partial-screen playback and offers you more control over various aspects of the playback. In order to support the new behaviors, however, many of the older methods and properties were deprecated or had their behavior modified significantly. Thus, older code will not behave as expected in iOS 3.2.

The major changes that are most likely to affect your existing code are the following:

- The movie player controller no longer manages the presentation of the movie. Instead, it vends a view object that serves as the playback surface for the movie content.
- Calling the `play` method still starts playback of the movie but it does not ensure that the movie is visible.

In order to display a movie, you must get the view from your `MPMoviePlayerController` object and add that view to a view hierarchy. Typically, you would do this from one of your view controllers. For example, if you load your views programmatically, you could do it in your `loadView` method; otherwise, you could do it in your `viewDidLoad` method. Upon presenting your view controller, you could then begin playback of the movie or let the user begin playback by displaying the movie player's built-in controls.

If you want to present a movie in full-screen mode, there are two ways to do it. The simplest way is to present your movie using an instance of the `MPMoviePlayerViewController` class, which is new in iOS 3.2. This class inherits from `UIViewController`, so it can be presented by your application like any other view controller. When presented modally using the `presentMoviePlayerViewControllerAnimated:` method, presentation of the movie replicates the experience previously provided by the `MPMoviePlayerController` class, including the transition animations used during presentation. To dismiss the view controller, use the `dismissMoviePlayerViewControllerAnimated` method.

Another way to present your movie full-screen is to incorporate the view from a `MPMoviePlayerController` object into your view hierarchy and then call its `setFullscreen:animated:` method. This method toggles the movie presentation between full-screen mode and displaying the movie content in just the view.

In addition to the changes you must make to your existing code, there are several new features that applications running in iOS 3.2 can use, including:

- You can change the movie being played programmatically without creating a new movie player controller.
- You can programmatically start, stop, pause, and scrub (forward and backward) through the current movie.
- You can now embed additional views on top of the video content.
- The movie player controller provides a background view to which you can incorporate custom background content.
- You can set both the start and stop times of the movie, and you can have the movie play in a loop and start automatically. (Previously, you could set only the start time.)
- You can generate thumbnail images from frames of the movie.
- You can get general information about the current state of the movie, including its duration, current playback position, and current playback rate.
- The movie player controller now generates notifications for most state changes.

The Core Application Design

Because it runs iOS, an iPad application uses all of the same objects and interfaces found in existing iPhone applications. As a result, the core architecture of the two application types is identical. However, iOS 3.2 introduces some new features that you can take advantage of in your iPad applications that you cannot use in your iPhone applications. This chapter describes those features and shows you how and when to use them in your application.

iPad Application Architecture

Although the architecture of iPhone and iPad applications is identical, there are places where you may need to adjust your code or resource files to support one device type or another. Figure 3-1 recaps the basic iPhone application architecture, showing the key objects that are most commonly found, and Table 3-1 describes the roles of each of these types of objects. (For a more in-depth introduction to the core architecture of iPhone (and thus iPad) applications, see *iOS Application Programming Guide*.)

Figure 3-1 Key objects in an iPad application

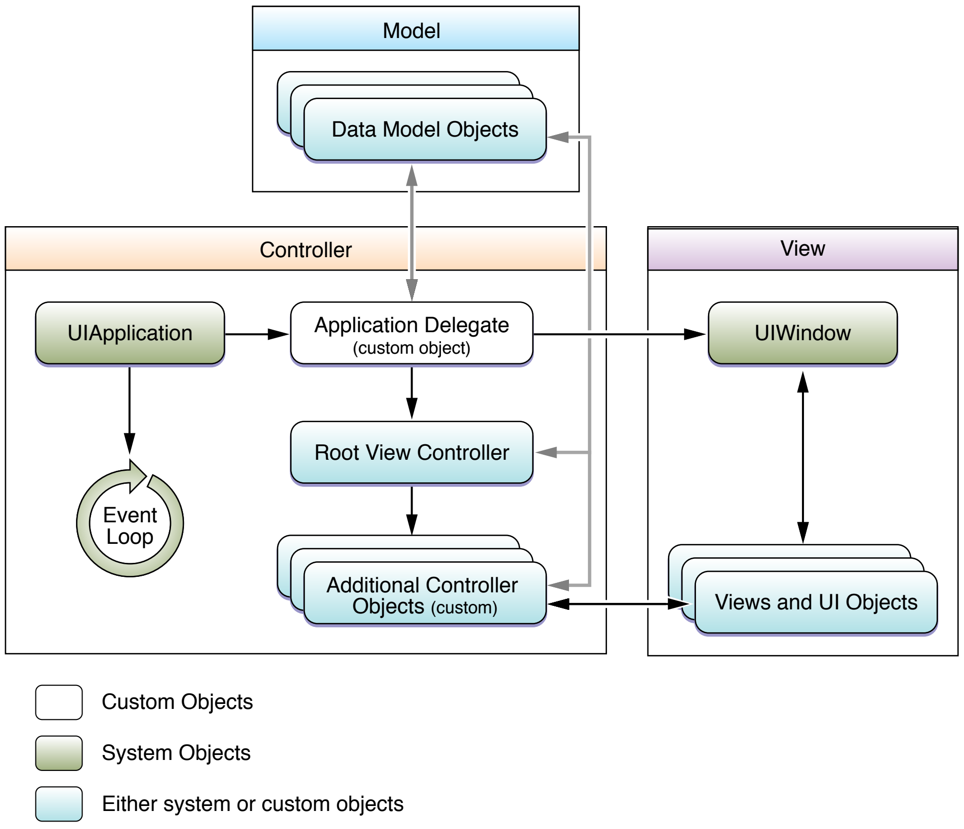


Table 3-1 The role of objects in an application

Object	Description
`UIApplication` object	The `UIApplication` object manages the application event loop and coordinates other high-level behaviors for your application. You use this object as-is, using it mostly to configure various aspects of your application's appearance. Your custom application-level code resides in your application delegate object, which works in tandem with this object.
Application delegate object	The application delegate is a custom object that you provide at application launch, usually by embedding it in your application's main nib file. The primary job of this object is to initialize the application and present its window onscreen. The `UIApplication` object also notifies this object about when specific application-level events occur, such as when the application needs to be interrupted (because of an incoming message) or terminated (because the user tapped the Home button). In an iPad application, you continue to use your delegate object to coordinate launch-time and quit-time behaviors for the application. However, you may need to include conditional checks in your delegate methods to provide custom support for each device type. Specifically, at launch time, you would typically need to load different nib files for your initial interface. Similarly, your initialization and termination code might also vary depending on the device type.
Data model objects	Data model objects store your application's content and are therefore custom to your application. Ideally, there should be few, if any, differences in your data objects on each device. The only time there might be differences is if you add or modify data objects to support iPad-specific features.
View controller objects	View controller objects manage the presentation of your application's user interface and also coordinate interactions between your data model objects and the views used to present that data. The `UIViewController` class is the base class for all view controller objects and provides a significant amount of default behavior so as to minimize the work you have to do. When porting an iPhone application, most of the changes occur in your views and view controllers. How much you need to modify your view controllers depends entirely on how much you change your views. If the changes are small, you might be able to reuse your existing view controllers and make minor changes to support each device. If the changes are significant, you might need to define separate view controller classes for your iPad and iPhone applications. I
`UIWindow` object	A `UIWindow` object manages the drawing surface for your application. You use windows in essentially the same way in both iPad and iPhone applications. After creating the window and installing your root views, you essentially ignore it. Any changes to your user interface happen through manipulations to your view controllers and not to your window object.

Object	Description
Views and UI objects	Views and controls provide the visual representation of your application's content. The UIKit framework provides standard views for implementing tables, buttons, picker controls, text labels, input fields, and many others. You can also define your own custom views by subclassing `UIView` (or its descendants) directly. In an iPad application, you need to adjust your views to fit the larger screen of the device. The scope of this "adjustment" can range from scaling up the size of your existing views to replacing some or all of them entirely. Replacing views might seem extreme but might also yield better results, especially if the new views are able to use the extra screen space more efficiently.

When porting an existing iPhone application to iPad, the biggest changes will be to your application's custom views and view controllers. Other changes might also be required but your views and view controllers are the ones you almost certainly have to change.

For examples of how to use the new views and view controllers in iOS 3.2, see "Views and View Controllers" (page 35). For a list of design guidelines you should consider when putting together your user interface, see *iPad Human Interface Guidelines*.

The Application Bundle

An iPad application uses the same bundle structure as an iPhone application. In other words, most of the application's code and resources reside in the top-level directory of the bundle. The contents of the bundle are also very similar, but there are some features that are available only in iPad applications.

New Keys for the Application's Info.plist File

There are additional keys for the information property list file (`Info.plist` file) that you use to support features specific to iPad applications. Most of these keys are optional, although one key is required and one key is strongly recommend. Table 3-2 lists the new keys and when you would include them in your application's `Info.plist` file. Whenever possible, you should modify `Info.plist` keys by changing the appropriate build settings in Xcode. However, the addition of some keys may require you to edit the file manually.

Table 3-2 New `Info.plist` keys in iOS 3.2

Key	Description
`UIDeviceFamily`	(Required) Identifies which devices the application supports. Set the value of this key by modifying the value in the Targeted Device Family build setting of your Xcode project. Any new applications you create specifically for iPad should include this key automatically. Similarly, any projects you transition over to support iPad should add this key automatically.

Key	Description
`UILaunchImageFile`	Contains a string that identifies the name of the launch image to use. If this key is not present, the application looks for an image with the name `Default.png`. Universal applications can use this key to specify different default images for iPad and iPhone applications. For more information about launch images, see "Providing Launch Images for Different Orientations" (page 30).
`UISupportedInterface-Orientations`	(Recommended) Contains an array of strings that specifies the orientations that the application supports at launch time. Possible values are the constants specified by the `UIInterfaceOrientation` type. The system uses this information to choose an appropriate launch image for the application, as described in "Providing Launch Images for Different Orientations" (page 30). Your application must similarly be prepared to configure its initial user interface in the any of the designated orientations.
`CFBundleDocument-Types`	Contains an array of dictionaries, each of which specifies a document type the application is able to open. You can use this key to let the system know that your application supports the opening of specific file types. To specify document type information, select your application target and open the inspector window. In the Properties pane, use the Document Types section to enter your document type information. The only fields you are required to fill in are the Name and UTI fields. Most other fields are ignored.
`CFBundleIconFiles`	Specifies an array of file names identifying the image resources to use for the application icon. If your application supports iPhone and iPad devices, you can specify different image resources for each. The system automatically uses the most appropriately sized image on each system. For more information about how to use this key, see the discussion of application icons in "Build-Time Configuration Details" in *iOS Application Programming Guide*.

For a complete list of the keys you can include in your application's `Info.plist` file, see *Information Property List Key Reference*.

Providing Launch Images for Different Orientations

A launch image is a static image file provided by the application and displayed by the system when the application is first launched. The system displays the launch image to give the user immediate feedback that the application launched and to give the application time to initialize itself and prepare its initial set of views for display. Because iPad applications can launch in any interface orientation, they can specify different launch images for each unique starting orientation.

For more information about how to specify launch images for different orientations, see the discussion of launch images in "Build-Time Configuration Details" in *iOS Application Programming Guide*.

Note: Although the image may be different at launch time, the configuration process for your application remains largely the same as for iPhone and iPod touch devices. Your `application:didFinishLaunchingWithOptions:` method should set up your window and views using a single preferred orientation. In other words, you should not attempt to match the initial orientation of your window and views to match the device's current orientation. Shortly after your `application:didFinishLaunchingWithOptions:` method returns, the system notifies your window of the correct starting orientation to give it a chance to reorient your content using the standard process.

Providing Application Icons for iPad

An iPad application supports the following icon sizes:

- A 72 x 72 pixel for the main application icon.
- A 50 x 50 pixel icon for displaying with Spotlight search results.
- A 29 x 29 pixel icon for the application's Settings bundle, if present.

For information about how to specify these icons in your application bundle, see the discussion of application icons in "Build-Time Configuration Details" in *iOS Application Programming Guide*.

Document Support on iPad Devices

Applications running on iPad devices have access to enhanced support for handling and managing documents and files. The purpose of this support is to make it easier for applications to work with files behind the scenes. When an application encounters a file of an unknown type, it can ask the system for help in displaying that file's contents or finding an application that can display them. If your application is able to display certain file formats, you can also register with the system as an application capable of displaying that file.

Previewing and Opening Files

When your application needs to interact with files of unknown types, you can use a `UIDocumentInteractionController` object to manage those interactions. A document interaction controller works with the system to determine whether files can be previewed in place or opened by another application. Your application works with the document interaction controller to present the available options to the user at appropriate times.

To use a document interaction controller in your application, you do the following:

1. Create an instance of the `UIDocumentInteractionController` class for each file you want to manage.
2. Present the file in your application's user interface. (Typically, you would do this by displaying the file name or icon somewhere in your interface.)
3. When the user interacts with the file, ask the document interaction controller to present one of the following interfaces:
    - A file preview view that displays the contents of the file

- A menu containing options to preview the file, copy its contents, or open it using another application
- A menu prompting the user to open it with another application

Any application that interacts with files can use a document interaction controller. Programs that download files from the network are the most likely candidates to need these capabilities. For example, an email program might use document interaction controllers to preview or open files attached to an email. Of course, you do not need to download files from the network to use this feature.

Creating and Configuring a Document Interaction Controller

To create a new document interaction controller, initialize a new instance of the `UIDocumentInteractionController` class with the file you want it to manage and assign an appropriate delegate object. Your delegate object is responsible for providing the document interaction controller with information it needs to present its views. You can also use the delegate to perform additional actions when those views are displayed. The following code creates a new document interaction controller and sets the delegate to the current object. Note that the caller of this method needs to retain the returned object.

```
- (UIDocumentInteractionController*)docControllerForFile:(NSURL*)fileURL
{
   UIDocumentInteractionController* docController =
       [UIDocumentInteractionController interactionControllerWithURL:fileURL];
   docController.delegate = self;

   return docController;
}
```

Once you have a document interaction controller object, you can use its properties to get information about the file, including its name, type information, and path information. The controller also has an `icons` property that contains `UIImage` objects representing the document's icon in various sizes. You can use all of this information when presenting the document in your user interface.

If you plan to let the user open the file in another application, you can use the `annotation` property of the document interaction controller to pass custom information to the opening application. It is up to you to provide information in a format that the other application will recognize. For example, this property is typically used by application suites that want to communicate additional information about a file to other applications in the suite. The opening application sees the annotation data in the `UIApplicationLaunchOptionsAnnotationKey` key of the options dictionary that is passed to it at launch time.

Presenting a Document Interaction Controller

When the user interacts with a file, you use the document interaction controller to display the appropriate user interface. You have the choice of displaying a document preview or of prompting the user to choose an appropriate action for the file using one of the following methods:

- Use the `presentOptionsMenuFromRect:inView:animated:` or `presentOptionsMenuFromBarButtonItem:animated:` method to present the user with a variety of options.
- Use the `presentPreviewAnimated:` method to display a document preview.

- Use the `presentOpenInMenuFromRect:inView:animated:` or `presentOpenInMenuFromBarButtonItem:animated:` method to present the user with a list of applications with which to open the file.

Each of the preceding methods attempts to display a custom view with the appropriate content. When calling these methods, you should always check the return value to see if the attempt was actually successful. These methods may return `NO` if the resulting interface would have contained no content. For example, the `presentOpenInMenuFromRect:inView:animated:` method returns `NO` if there are no applications capable of opening the file.

If you choose a method that might display a preview of the file, your delegate object must implement the `documentInteractionControllerViewControllerForPreview:` method. Document previews are displayed using a modal view, so the view controller you return becomes the parent of the modal document preview. If you do not implement this method, if your implementation returns `nil`, or if the specified view controller is unable to present another modal view controller, a document preview is not displayed.

Normally, the document interaction controller automatically handles the dismissal of the view it presents. However, you can dismiss the view programmatically as needed by calling the `dismissMenuAnimated:` or `dismissPreviewAnimated:` methods.

Registering the File Types Your Application Supports

If your application is capable of opening specific types of files, you should register that support with the system. To declare its support for file types, your application must include the `CFBundleDocumentTypes` key in its `Info.plist` file. The system gathers this information from your application and maintains a registry that other applications can access through a document interaction controller.

The `CFBundleDocumentTypes` key contains an array of dictionaries, each of which identifies information about a specific document type. A document type usually has a one-to-one correspondence with a particular document type. However, if your application treats more than one file type the same way, you can group those types together as a single document type. For example, if you have two different file formats for your application's native document type, you could group both the old type and new type together in a single document type entry. By doing so, both the new and old files would appear to be the same type of file and would be treated in the same way.

Each dictionary in the `CFBundleDocumentTypes` array can include the following keys:

- `CFBundleTypeName` specifies the name of the document type.
- `CFBundleTypeIconFiles` is an array of filenames for the image resources to use as the document's icon.
- `LSItemContentTypes` contains an array of strings with the UTI types that represent the supported file types in this group.
- `LSHandlerRank` describes whether this application owns the document type or is merely able to open it.

From the perspective of your application, a document is a file type (or file types) that the application supports and treats as a single entity. For example, an image processing application might treat different image file formats as different document types so that it can fine tune the behavior associated with each one. Conversely, a word processing application might not care about the underlying image formats and just manage all image formats using a single document type.

Listing 3-1 shows a sample XML snippet from the `Info.plist` of an application that is capable of opening a custom file type. The `LSItemContentTypes` key identifies the UTI associated with the file format and the `CFBundleTypeIconFiles` key points to the icon resources to use when displaying it.

Listing 3-1 Document type information for a custom file format

```
<dict>
   <key>CFBundleTypeName</key>
   <string>My File Format</string>
   <key>CFBundleTypeIconFiles</key>
       <array>
           <string>MySmallIcon.png</string>
           <string>MyLargeIcon.png</string>
       </array>
   <key>LSItemContentTypes</key>
       <array>
           <string>com.example.myformat</string>
       </array>
   <key>LSHandlerRank</key>
   <string>Owner</string>
</dict>
```

For more information about the contents of the `CFBundleDocumentTypes` key, see the description of that key in *Information Property List Key Reference*.

Opening Supported File Types

At launch time, the system may ask your application to open a specific file and present it to the user. This typically occurs because another application encountered the file and used a document interaction controller to handle it. You receive information about the file to be opened in the `application:didFinishLaunchingWithOptions:` method of your application delegate. If your application handles custom file types, you must implement this delegate method (instead of the `applicationDidFinishLaunching:` method) and use it to initialize your application.

The options dictionary passed to the `application:didFinishLaunchingWithOptions:` method contains information about the file to be opened. Specifically, your application should look in this dictionary for the following keys:

- `UIApplicationLaunchOptionsURLKey` contains an `NSURL` object that specifies the file to open.
- `UIApplicationLaunchOptionsSourceApplicationKey` contains an `NSString` with the bundle identifier of the application that initiated the open request.
- `UIApplicationLaunchOptionsAnnotationKey` contains a property list object that the source application wanted to associate with the file when it was opened.

If the `UIApplicationLaunchOptionsURLKey` key is present, your application must open the file referenced by that key and present its contents immediately. You can use the other keys in the dictionary to gather information about the circumstances surrounding the opening of the file.

Views and View Controllers

In iOS 3.2, the UIKit framework includes new capabilities to help you organize and present content on an iPad. These capabilities range from new view controller classes to modifications to existing interface features. For additional information about when it is appropriate to incorporate these features into your applications, see *iPad Human Interface Guidelines*.

Designing for Multiple Orientations

With few exceptions, applications should support all interface orientations on iPad devices. The steps for supporting orientation changes are the same on iPad devices as they are on iPhone and iPod touch devices. The application's window and view controllers provide the basic infrastructure needed to support rotations. You can use the existing infrastructure as-is or customize the behavior to suit the particulars of your application.

To implement basic support for all interface orientations, you must do the following:

- Implement the `shouldAutorotateToInterfaceOrientation:` method in each of your custom view controllers and return `YES` for all orientations.
- Configure the `autoresizingMask` property of your views so that they respond to layout changes appropriately. (You can configure this property either programmatically or using Interface Builder.)

To go beyond the basic support, there are additional tasks you can perform depending on your needs:

- For custom views that need to control the placement of subviews more precisely, override the `layoutSubviews` method and put your custom layout code there.
- To perform tasks before during or after the actual rotation of your views, use the one-step rotation notifications of the `UIViewController` class.

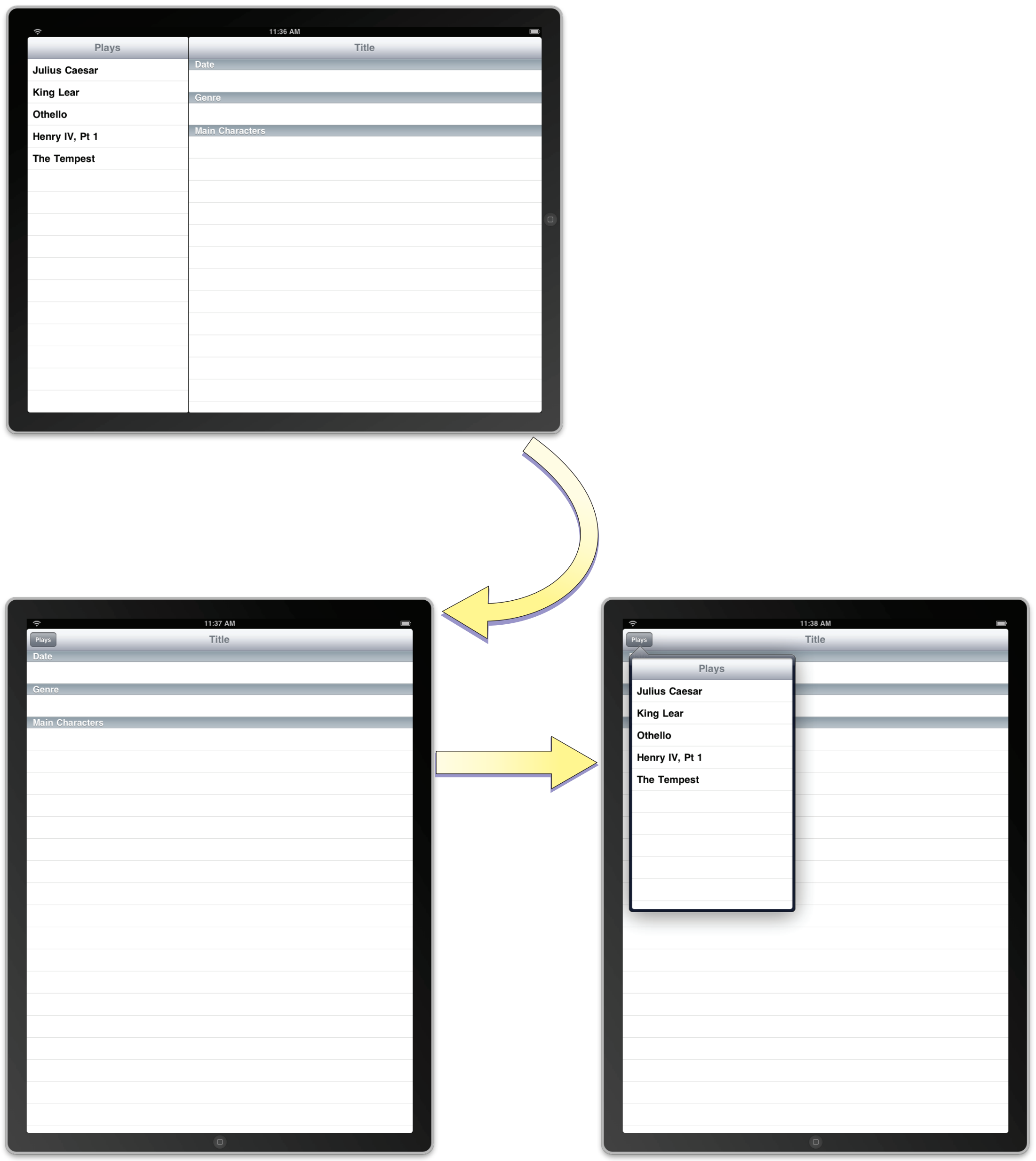
When an orientation change occurs, the window works with its frontmost view controller to adjust the content to match the new orientation. During this process, the view controller receives several notifications to give you a chance to perform additional tasks. Specifically, the view controller's `willRotateToInterfaceOrientation:duration:`, `willAnimateRotationToInterfaceOrientation:duration:`, and `didRotateFromInterfaceOrientation:` methods are called at appropriate points to give you a chance to perform tasks before and after the rotation of your views. You can use these methods to perform any tasks related to the orientation change. For example, you might use them to add or remove views, reload the data in any visible tables, or tweak the performance of your code during the rotation process.

For more information about responding to orientation changes in your view controllers, see *View Controller Programming Guide for iOS*.

Creating a Split View Interface

A split view consists of two side-by-side panes separated by a divider element. The first pane of a split view controller has a fixed width of 320 points and a height that matches the visible window height. The second pane fills the remaining space. In iOS, split views can be used in master-detail interfaces or wherever you want to display two different types of information side-by-side. When the device is in a landscape orientation, the split view shows both panes. However, in portrait orientations, the split view displays only the second pane, which grows to fill the available space. If you want the user to have access to the first pane, you must present that pane yourself. The most common way to display the first pane in portrait mode is to add a button to the toolbar of your second pane and use it to present a popover with the first pane contents, as shown in Figure 4-1.

Figure 4-1 A split view interface



The `UISplitViewController` class manages the presentation of the side-by-side panes. The panes themselves are each managed by a view controller that you provide. The split view controller handles rotations and other system-related behaviors that require coordination between the two panes. The split view controller's view should always be installed as the root view of your application window. You should never present a split view inside of a navigation or tab bar interface.

The easiest way to integrate a split view controller into your application is to start from a new project. The Split View-based Application template in Xcode provides a good starting point for building an interface that incorporates a split view controller. Everything you need to implement the split view interface is already provided. All you have to do is modify the array of view controllers to present your custom content. The process for modifying these view controllers is virtually identical to the process used in iPhone applications.

The only difference is that you now have more screen space available for displaying your detail-related content. However, you can also integrate split view controllers into your existing interfaces, as described in "Adding a Split View Controller in Interface Builder" (page 38).

For more information about configuring view controllers in your application, see *View Controller Programming Guide for iOS*.

Adding a Split View Controller in Interface Builder

If you do not want to start with the Split View-based Application template project, you can still add a split view controller to your user interface. The library in Interface Builder includes a split view controller object that you can add to your existing nib files. When adding a split view controller, you typically add it to your application's main nib file. This is because the split view is usually inserted as the top-level view of your application's window and therefore needs to be loaded at launch time.

To add a split view controller to your application's main nib file:

1. Open your application's main nib file.
2. Drag a split view controller object to the nib file window.

   The split view controller object includes generic view controllers for the two panes.
3. Add an outlet for the split view controller in your application delegate object and connect that outlet to the split view controller object.
4. In the `application:didFinishLaunchingWithOptions:` method of your application delegate, install the split view controller's view as the main view of the window:

   ```
   [window addSubview:mySplitViewController.view];
   ```
5. For each of the split view controller's contained view controllers:
   - Use the Identity inspector to set the class name of the view controller.
   - In the Attributes inspector, set the name of the nib file containing the view controller's view.

The contents of the two view controllers you embed in the split view are your responsibility. You configure these view controllers just as you would configure any other view controllers in your application. Setting the class and nib names is all you have to do in your application's main nib file. The rest of the configuration is dependent on the type of view controller. For example, for navigation and tab bar controllers, you may need to specify additional view controller information. The process for configuring navigation, tab bar, and custom view controllers is described in *View Controller Programming Guide for iOS*.

Creating a Split View Controller Programmatically

To create a split view controller programmatically, create a new instance of the `UISplitViewController` class and assign view controllers to its two properties. Because its contents are built on-the-fly from the view controllers you provide, you do not have to specify a nib file when creating a split view controller. Therefore, you can just use the `init` method to initialize it. Listing 4-1 shows an example of how to create and configure a split view interface at launch time. You would replace the first and second view controllers with the custom view controller objects that present your application's content. The `window` variable is assumed to be an outlet that points to the window loaded from your application's main nib file.

Listing 4-1 Creating a split view controller programmatically

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
   MyFirstViewController* firstVC = [[[MyFirstViewController alloc]
                     initWithNibName:@"FirstNib" bundle:nil] autorelease];
   MySecondViewController* secondVC = [[[MySecondViewController alloc]
                     initWithNibName:@"SecondNib" bundle:nil] autorelease];

   UISplitViewController* splitVC = [[UISplitViewController alloc] init];
   splitVC.viewControllers = [NSArray arrayWithObjects:firstVC, secondVC, nil];

   [window addSubview:splitVC.view];
   [window makeKeyAndVisible];

   return YES;
}
```

Supporting Orientation Changes in a Split View

A split view controller relies on its two view controllers to determine whether interface orientation changes should be made. If one or both of the view controllers do not support the new orientation, no change is made. This is true even in portrait mode, where the first view controller is not displayed. Therefore, you must override the `shouldAutorotateToInterfaceOrientation:` method for both view controllers and return `YES` for all supported orientations.

When an orientation change occurs, the split view controller automatically handles most of the rotation behaviors. Specifically, the split view controller automatically hides the first view controller in its `viewControllers` array when rotating to a portrait orientation and shows it when rotating to a landscape orientation.

When in a portrait orientation, if you want to display the first view controller using a popover, you can do so using a delegate object. When the view controller is hidden or shown, the split view controller notifies its delegate of the occurrence. When the view controller is hidden, the delegate is provided with a button and popover controller to use to show the view controller. All your delegate method has to do is add the specified button to the a button to a visible toolbar so as to provide access to the view controller. Similarly, when the view controller is shown again, the delegate is given a chance to remove the button. For more information about the delegate methods and how you use them, see *UISplitViewControllerDelegate Protocol Reference*.

Using Popovers to Display Content

A popover is a special type of interface element that you use to layer information temporarily on top of the current view. Popovers provide a lightweight way to present or gather information in a way that does not require user action. For example, popovers are ideally suited for the following situations:

- To present part of a split view interface when the device is in a portrait orientation

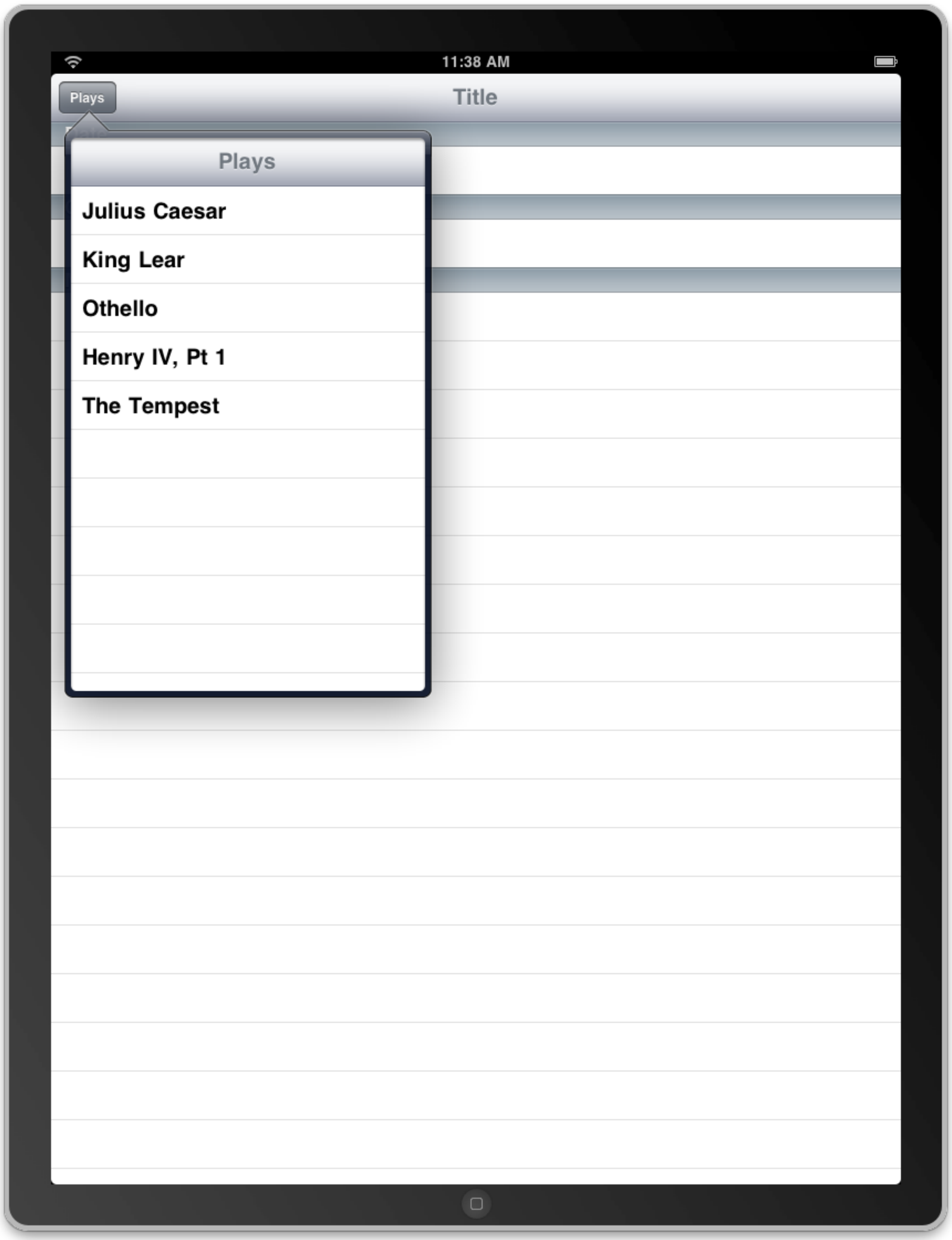
- To present a list of actions to perform on objects inside one of your views
- To display information about an object on the screen.
- To manage frequently accessed tools or configuration options

In an iPhone application, you might implement some of the preceding actions using a modal view. On iPad devices, popovers and modal views really have different purposes. In an iPad application, you would use modal views to interrupt the current workflow to gather some required piece of information from the user. The interruption is punctuated by the fact that the user must expressly accept or cancel the action. A popover provides a much less intrusive form of interruption and does not require express acceptance or cancellation by the user. The popover is displayed on top of the user's content and can be dismissed easily by tapping outside the popover's bounds. Thus, selecting items from a popover is an optional affair. The only time the state of your application should be affected is when the user actually interacts with the popover's contents.

A popover is displayed next to the content it is meant to modify and typically contains an arrow pointing to that content. The size of the popover itself is configurable and is based on the size of the view controller, although you can change that size as needed. In addition, the popover itself may change the size of the presented content in order to ensure that the popover fits neatly on the screen.

Figure 4-2 shows an example of a popover used to display the master portion of a split view interface. In portrait orientations, a custom button is added to the detail pane's toolbar. When the button is tapped, the application displays the popover.

Figure 4-2 Using a popover to display a master pane



For more information about when to use popovers, see *iPad Human Interface Guidelines*.

Creating and Presenting a Popover

The content of a popover is derived from the view controller that you provide. Popovers are capable of presenting most types of view controllers, including custom view controllers, table view controllers, navigation controllers, and even tab bar controllers. When you are ready to present that view controller in a popover, do the following:

1. Create an instance of the `UIPopoverController` class and initialize it with your custom view controller.
2. (Optional) Customize the size of the popover using the `popoverContentSize` property.
3. (Optional) Assign a delegate to the popover. For more information about the responsibilities of the delegate, see "Implementing a Popover Delegate" (page 43).
4. Present the popover.

When you present a popover, you associate it with a particular portion of your user interface. Popovers are commonly associated with toolbar buttons, so the `presentPopoverFromBarButtonItem:permittedArrowDirections:animated:` method is a convenient way to present popovers from your application's toolbar. If you want to associate the popover with the contents of one of your views, you can use the `presentPopoverFromRect:inView:permittedArrowDirections:animated:` method to present instead.

The popover derives its initial size from the `contentSizeForViewInPopover` property of the view controller being presented. The default size stored in this property is 320 pixels wide by 1100 pixels high. You can customize the default value by assigning a new value to the `contentSizeForViewInPopover` property. Alternatively, you can assign a value to the `popoverContentSize` property of the popover controller itself. If you change the content view controller displayed by a popover, any custom size information you put in the `popoverContentSize` property is replaced by the size of the new view controller. Changes to the content view controller or its size while the popover is visible are automatically animated. You can also change the size (with or without animations) using the `setPopoverContentSize:animated:` method.

Listing 4-2 shows a simple action method that presents a popover in response to user taps in a toolbar button. The popover is stored in a property (defined by the owning class) that retains the popover object. The size of the popover is set to the size of the view controller's view, but the two need not be the same. Of course, if the two are not the same, you must use a scroll view to ensure the user can see all of the popover's contents.

Listing 4-2 Presenting a popover

```
- (IBAction)toolbarItemTapped:(id)sender
{
   MyCustomViewController* content = [[MyCustomViewController alloc] init];
   UIPopoverController* aPopover = [[UIPopoverController alloc]
        initWithContentViewController:content];
   aPopover.delegate = self;
   [content release];

   // Store the popover in a custom property for later use.
   self.popoverController = aPopover;
   [aPopover release];

   [self.popoverController presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
```

To dismiss a popover programmatically, call the `dismissPopoverAnimated:` method of the popover controller. Dismissing the popover is required only if you want taps within the popover content area to cause the popover to go away. Taps outside the popover automatically cause it to be dismissed. In general, dismissing the popover in response to taps inside the content area is recommended, especially if those taps trigger a selection or some other change to the underlying content. However, it is up to you to decide whether such a behavior is appropriate for your application. Be aware, though, that it is your responsibility to store a reference to the popover controller so that you can dismiss it. The system does not provide one by default.

Implementing a Popover Delegate

When a popover is dismissed due to user taps outside the popover view, the popover automatically notifies its delegate. Before the popover is dismissed, the popover controller sends a `popoverControllerShouldDismissPopover:` message to its delegate. If your delegate's implementation of this method returns `YES`, or if the delegate does not implement the method at all, the controller dismisses the popover and sends a `popoverControllerDidDismissPopover:` message to the delegate.

In most situations, you should not need to override the `popoverControllerShouldDismissPopover:` method at all. The method is provided for situations where dismissing the popover might cause problems for your application. In such a situation, you can implement it and return `NO`. However, a better approach is to avoid putting your application into such a situation.

By the time the `popoverControllerDidDismissPopover:` method of your delegate is called, the popover itself has been removed from the screen. At this point, it is safe to release the popover controller if you do not plan to use it again. You can also use this message to refresh your user interface or update your application's state.

Tips for Managing Popovers in Your Application

Consider the following when writing popover-related code for your application:

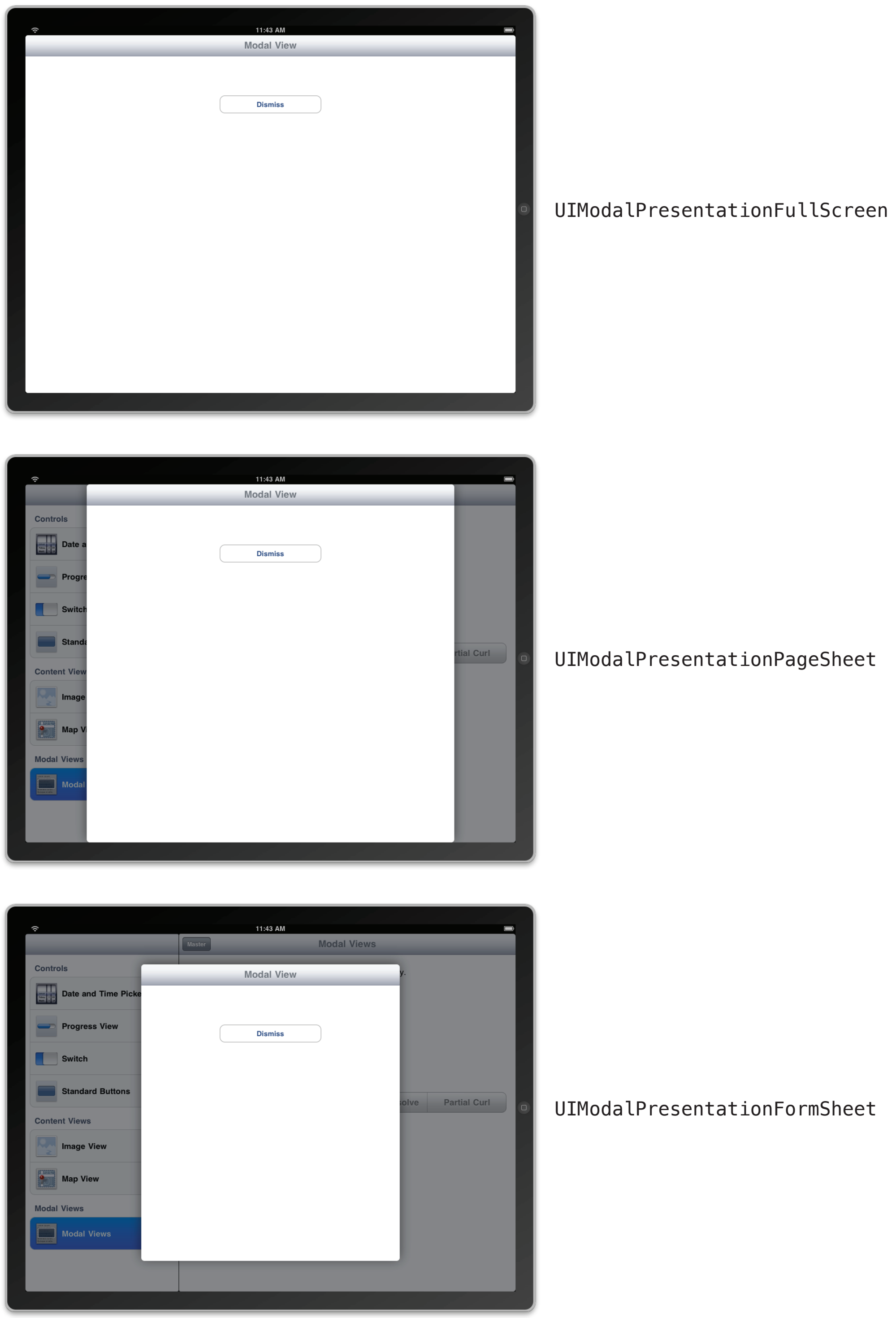
- Dismissing a popover programmatically requires a pointer to the popover controller. The only way to get such a pointer is to store it yourself, typically in the content view controller. This ensures that the content view controller is able to dismiss the popover in response to appropriate user actions.
- Cache frequently used popover controllers rather than creating new ones from scratch. Similarly, feel free to reuse popover controllers in your application rather than create new ones for each distinct popover. Popover controllers are fairly malleable objects and can be reused easily. They are also easy objects to release if your application receives a low-memory warning.
- When presenting a popover, specify the `UIPopoverArrowDirectionAny` constant for the permitted arrow direction whenever possible. Specifying this constant gives the UIKit the maximum flexibility in positioning and sizing the popover. If you specify a limited set of permitted arrow directions, the popover controller may have to shrink the size of your popover before displaying it.

Configuring the Presentation Style for Modal Views

In iOS 3.2, there are new options for presenting view controllers modally. Previously, modally presented views always covered the visible portion of the underlying window. Now, the `UIViewController` class has a `modalPresentationStyle` property that determines the appearance of the view controller when it is presented modally. The different options for this property allow you to present the view controller so that it fills the entire screen, as before, or only part of the screen.

Figure 4-3 shows the core presentation styles that are available. (The `UIModalPresentationCurrentContext` style lets a view controller adopt the presentation style of its parent.) In each modal view, the dimmed areas show the underlying content but do not allow taps in that content. Therefore, unlike a popover, your modal views must still have controls that allow the user to dismiss the modal view.

Figure 4-3 Modal presentation styles



For guidance on when to use the different presentation styles, see *iPad Human Interface Guidelines*.

Making Better Use of Toolbars

Although toolbars have been supported since iOS 2.0, they have a more prominent role in iPad applications. Prior to iOS 3.2, the user interface guidelines encouraged the placement of toolbars at the bottom of the application's window. The upper edge of the window was reserved for a navigation bar, which provided common navigation to and from views. With the expanded space available on iPad devices, toolbars can now be placed along the top edge of the application's window in place of a navigation bar. This positioning lets you give your toolbar commands more prominence in your application.

For guidelines about the configuration and usage of toolbars in your application, see *iPad Human Interface Guidelines*.

Gesture Recognizers

Applications for iOS are driven largely through events generated when users touch buttons, toolbars, table-view rows and other objects in an application's user interface. The classes of the UIKit framework provide default event-handling behavior for most of these objects. However, some applications, primarily those with custom views, have to do their own event handling. They have to analyze the stream of touch objects in a multitouch sequence and determine the intention of the user.

Most event-handling views seek to detect common gestures that users make on their surface—things such as triple-tap, touch-and-hold (also called long press), pinching, and rotating gestures, The code for examining a raw stream of multitouch events and detecting one or more gestures is often complex. Prior to iOS 3.2, you cannot reuse the code except by copying it to another project and modifying it appropriately.

To help applications detect gestures, iOS 3.2 introduces gesture recognizers, objects that inherit directly from the `UIGestureRecognizer` class. The following sections tell you about how these objects work, how to use them, and how to create custom gesture recognizers that you can reuse among your applications.

Note: For an overview of multitouch events on iOS, see ""Event Handling"" in *iOS Application Programming Guide*.

Gesture Recognizers Simplify Event Handling

`UIGestureRecognizer` is the abstract base class for concrete gesture-recognizer subclasses (or, simply, gesture recognizers). The `UIGestureRecognizer` class defines a programmatic interface and implements the behavioral underpinnings for gesture recognition. The UIKit framework provides six gesture recognizers for the most common gestures. For other gestures, you can design and implement your own gesture recognizer (see "Creating Custom Gesture Recognizers" (page 55) for details).

Recognized Gestures

The UIKit framework supports the recognition of the gestures listed in Table 5-1. Each of the listed classes is a direct subclass of `UIGestureRecognizer`.

Table 5-1 Gestures recognized by the gesture-recognizer classes of the UIKit framework

Gesture	UIKit class
Tapping (any number of taps)	`UITapGestureRecognizer`
Pinching in and out (for zooming a view)	`UIPinchGestureRecognizer`
Panning or dragging	`UIPanGestureRecognizer`

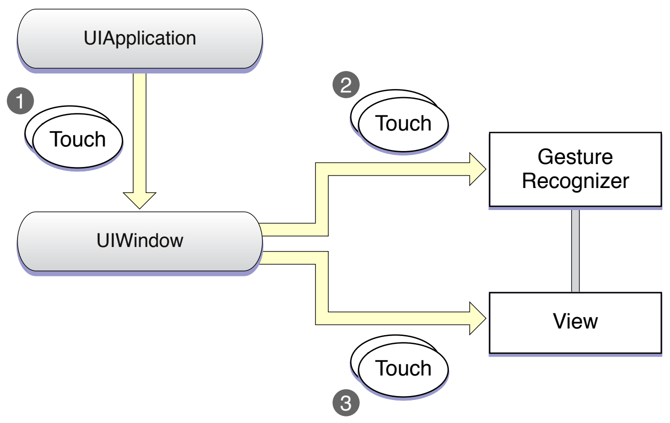
Gesture	UIKit class
Swiping (in any direction)	`UISwipeGestureRecognizer`
Rotating (fingers moving in opposite directions)	`UIRotationGestureRecognizer`
Long press (also known as "touch and hold")	`UILongPressGestureRecognizer`

Before you decide to use a gesture recognizer, consider how you are going to use it. Respond to gestures only in ways that users expect. For example, a pinching gesture should scale a view, zooming it in and out; it should not be interpreted as, say, a selection request, for which a tap is more appropriate. For guidelines about the proper use of gestures, see *iPad Human Interface Guidelines*.

Gestures Recognizers Are Attached to a View

To detect its gestures, a gesture recognizer must be attached to the view that a user is touching. This view is known as the *hit-tested view*. Recall that events in iOS are represented by represented by `UIEvent` objects, and each event object encapsulates the `UITouch` objects of the current multitouch sequence. A set of those `UITouch` objects is specific to a given phase of a multitouch sequence. Delivery of events initially follows the usual path: from operating system to the application object to the window object representing the window in which the touches are occurring. But before sending an event to the hit-tested view, the window object sends it to the gesture recognizer attached to that view or to any of that view's subviews. Figure 5-1 illustrates this general path, with the numbers indicating the order in which touches are received.

Figure 5-1 Path of touch objects when gesture recognizer is attached to a view



Thus gesture recognizers act as observers of touch objects sent to their attached view or view hierarchy. However, they are not part of that view hierarchy and do not participate in the responder chain. Gesture recognizers may delay the delivery of touch objects objects to the view while they are recognizing gestures, and by default they cancel delivery of remaining touch objects to the view once they recognize their gesture. For more on the possible scenarios of event delivery from a gesture recognizer to its view, see "Regulating the Delivery of Touches to Views" (page 54).

For some gestures, the `locationInView:` and the `locationOfTouch:inView:` methods of `UIGestureRecognizer` enable clients to find the location of gestures or specific touches in the attached view or its subviews. See "Responding to Gestures" (page 51) for more information.

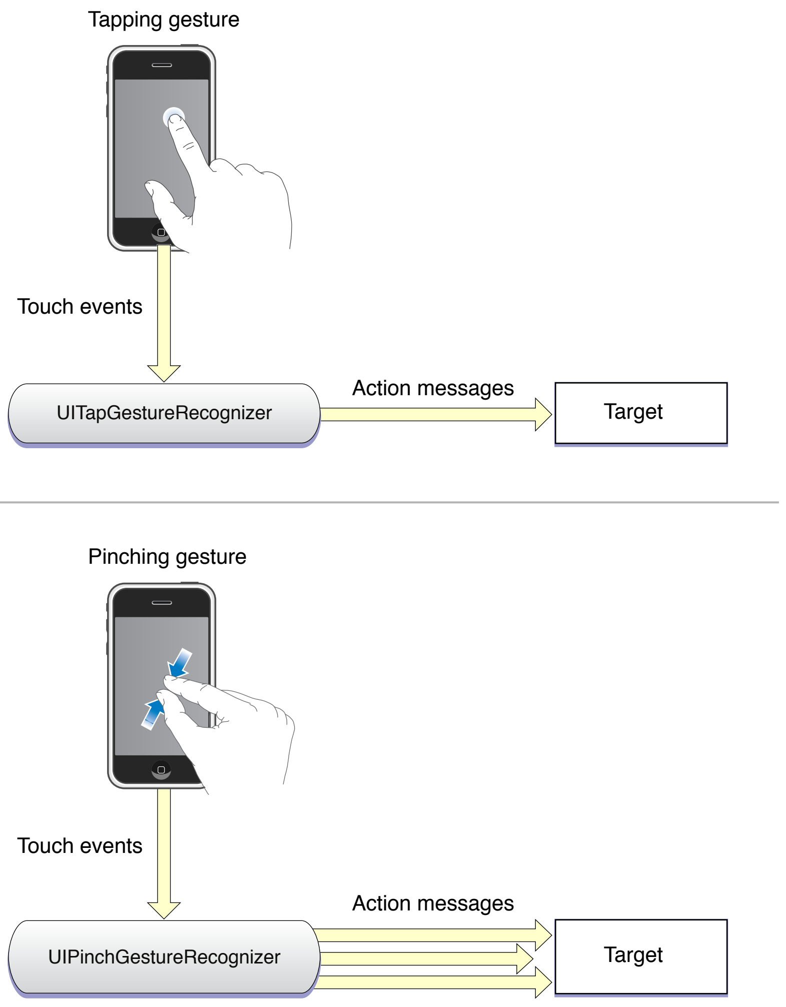
Gestures Trigger Action Messages

When a gesture recognizer recognizes its gesture, it sends one or more action messages to one or more targets. When you create a gesture recognizer, you initialize it with an action and a target. You may add more target-action pairs to it thereafter. The target-action pairs are not additive; in other words, an action is only sent to the target it was originally linked with, and not to other targets (unless they're specified in another target-action pair).

Discrete Gestures and Continuous Gestures

When a gesture recognizer recognizes a gesture, it sends either a single action message to its target or multiple action messages until the gesture ends. This behavior is determined by whether the gesture is discrete or continuous. A discrete gesture, such as a double-tap, happens just once; when a gesture recognizer recognizes a discrete gesture, it sends its target a single action message. A continuous gesture, such as pinching, takes place over a period and ends when the user lifts the final finger in the multitouch sequence. The gesture recognizer sends action messages to its target at short intervals until the multitouch sequence ends.

Figure 5-2 Discrete versus continuous gestures



The reference documents for the gesture-recognizer classes note whether the instances of the class detect discrete or continuous gestures.

Implementing Gesture Recognition

To implement gesture recognition, you create a gesture-recognizer instance to which you assign a target, action, and, in some cases, gesture-specific attributes. You attach this object to a view and then implement the action method in your target object that handles the gesture.

Preparing a Gesture Recognizer

To create a gesture recognizer, you must allocate and initialize an instance of a concrete `UIGestureRecognizer` subclass. When you initialize it, specify a target object and an action selector, as in the following code:

```
UITapGestureRecognizer *doubleFingerDTap = [[UITapGestureRecognizer alloc]
   initWithTarget:self action:@selector(handleDoubleDoubleTap:)];
```

The action methods for handling gestures—and the selector for identifying them—are expected to conform to one of two signatures:

- `(void)`*handleGesture*
- `(void)`*handleGesture*`:(UIGestureRecognizer *)`*sender*

where *handleGesture* and *sender* can be any name you choose. Methods having the second signature allow the target to query the gesture recognizer for addition information. For example, the target of a `UIPinchGestureRecognizer` object can ask that object for the current scale factor related to the pinching gesture.

After you create a gesture recognizer, you must attach it to the view receiving touches—that is, the hit-test view—using the `UIView` method `addGestureRecognizer:`. You can find out what gesture recognizers a view currently has attached through the `gestureRecognizers` property, and you can detach a gesture recognizer from a view by calling `removeGestureRecognizer:`.

The sample method in Listing 5-1 creates and initializes three gesture recognizers: a single-finger double-tap, a panning gesture, and a rotation gesture. It then attaches each gesture-recognizer object to the same view. For the `singleFingerDTap` object, the code specifies that two taps are required for the gesture to be recognized. Each method adds the created gesture recognizer to a view and then releases it (because the view now retains it).

Listing 5-1 Creating and initializing discrete and continuous gesture recognizers

```
- (void)createGestureRecognizers {
    UITapGestureRecognizer *singleFingerDTap = [[UITapGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleSingleDoubleTap:)];
    singleFingerDTap.numberOfTapsRequired = 2;
    [self.theView addGestureRecognizer:singleFingerDTap];
    [singleFingerDTap release];

    UIPanGestureRecognizer *panGesture = [[UIPanGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePanGesture:)];
    [self.theView addGestureRecognizer:panGesture];
    [panGesture release];

    UIPinchGestureRecognizer *pinchGesture = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinchGesture:)];
    [self.theView addGestureRecognizer:pinchGesture];
    [pinchGesture release];
}
```

You may also add additional targets and actions to a gesture recognizer using the `addTarget:action:` method of `UIGestureRecognizer`. Remember that action messages for each target and action pair are restricted to that pair; if you have multiple targets and actions, they are not additive.

Responding to Gestures

To handle a gesture, the target for the gesture recognizer must implement a method corresponding to the action selector specified when you initialized the gesture recognizer. For discrete gestures, such as a tapping gesture, the gesture recognizer invokes the method once per recognition; for continuous gestures, the gesture recognizer invokes the method at repeated intervals until the gesture ends (that is, the last finger is lifted from the gesture recognizer's view).

In gesture-handling methods, the target object often gets additional information about the gesture from the gesture recognizer; it does this by obtaining the value of a property defined by the gesture recognizer, such as `scale` (for scale factor) or `velocity`. It can also query the gesture recognizer (in appropriate cases) for the location of the gesture.

Listing 5-2 shows handlers for two continuous gestures: a rotation gesture (`handleRotate:`) and a panning gesture (`handlePanGesture:`). It also gives an example of a handler for a discrete gesture; in this example, when the user double-taps the view with a single finger, the handler (`handleSingleDoubleTap:`) centers the view at the location of the double-tap.

Listing 5-2 Handling pinch, pan, and double-tap gestures

```
- (IBAction)handlePinchGesture:(UIGestureRecognizer *)sender {
    CGFloat factor = [(UIPinchGestureRecognizer *)sender scale];
    self.view.transform = CGAffineTransformMakeScale(factor, factor);
}

- (IBAction)handlePanGesture:(UIPanGestureRecognizer *)sender {
    CGPoint translate = sender.translation;

    CGRect newFrame = currentImageFrame;
    newFrame.origin.x += translate.x;
    newFrame.origin.y += translate.y;
    sender.view.frame = newFrame;

    if (sender.state == UIGestureRecognizerStateEnded)
        currentImageFrame = newFrame;
}

- (IBAction)handleSingleDoubleTap:(UIGestureRecognizer *)sender {
    CGPoint tapPoint = [sender locationInView:sender.view.superview];
    [UIView beginAnimations:nil context:NULL];
    sender.view.center = tapPoint;
    [UIView commitAnimations];
}
```

These action methods handle the gestures in distinctive ways:

- In the `handlePinchGesture:` method, the target communicates with its gesture recognizer (`sender`) to get the scale factor (`scale`). The method uses the scale value in a Core Graphics function that scales the view and assigns the computed value to the view's affine `transform` property.
- The `handlePanGesture:` method applies the `translationInView:` values obtained from its gesture recognizer to a cached frame value for the attached view. When the gesture concludes, it caches the newest frame value.
- In the `handleSingleDoubleTap:` method, the target gets the location of the double-tap gesture from its gesture recognizer by calling the `locationInView:` method. It then uses this point, converted to superview coordinates, to animate the center of the view to the location of the double-tap.

The scale factor obtained in the `handlePinchGesture:` method, as with the rotation angle and the translation value related to other recognizers of continuous gestures, is to be applied to the state of the view when the gesture is first recognized. It is not a delta value to be concatenated over each handler invocation for a given gesture.

A hit-test with an attached gesture recognizer does not have to be passive when there are incoming touch events. Instead, it can determine which gesture recognizers, if any, are involved with a particular `UITouch` object by querying the `gestureRecognizers` property. Similarly, it can find out which touches a given gesture recognizer is analyzing for a given event by calling the `UIEvent` method `touchesForGestureRecognizer:`.

Interacting with Other Gesture Recognizers

More than one gesture recognizer may be attached to a view. In the default behavior, touch events in a multitouch sequence go from one gesture recognizer to another in a nondeterministic order until the events are finally delivered to the view (if at all). Often this default behavior is what you want. But sometimes you might want one or more of the following behaviors:

- Have one gesture recognizer fail before another can start analyzing touch events.
- Prevent other gesture recognizers from analyzing a specific multitouch sequence or a touch object in that sequence.
- Permit two gesture recognizers to operate simultaneously.

The `UIGestureRecognizer` class provides client methods, delegate methods, and methods overridden by subclasses to enable you to effect these behaviors.

Requiring a Gesture Recognizer to Fail

You might want a relationship between two gesture recognizers so that one can operate only if the other one fails. For example, recognizer A doesn't begin analyzing a multitouch sequence until recognizer B fails and, conversely, if recognizer B does recognize its gesture, recognizer A never looks at the multitouch sequence. An example where you might specify this relationship is when you have a gesture recognizer for a single tap and another gesture recognizer for a double tap; the single-tap recognizer requires the double-tap recognizer to fail before it begins operating on a multitouch sequence.

The method you call to specify this relationship is `requireGestureRecognizerToFail:`. After sending the message, the receiving gesture recognizer must stay in the `UIGestureRecognizerStatePossible` state until the specified gesture recognizer transitions to `UIGestureRecognizerStateFailed`. If the specified gesture recognizer transitions to `UIGestureRecognizerStateRecognized` or `UIGestureRecognizerStateBegan` instead, then the receiving recognizer can proceed, but no action message is sent if it recognizes its gesture.

For a discussion of gesture-recognition states and possible transition between these states, see "State Transitions" (page 56).

Preventing Gesture Recognizers from Analyzing Touches

You can prevent gesture recognizers from looking at specific touches or from even recognizing a gesture. You can specify these "prevention" relationships using either delegation methods or overriding methods declared by the `UIGestureRecognizer` class.

The `UIGestureRecognizerDelegate` protocol declares two optional methods that prevent specific gesture recognizers from recognizing gestures on a case-by-case basis:

- `gestureRecognizerShouldBegin:` — This method is called when a gesture recognizer attempts to transition out of `UIGestureRecognizerStatePossible`. Return `NO` to make it transition to `UIGestureRecognizerStateFailed` instead. (The default value is `YES`.)
- `gestureRecognizer:shouldReceiveTouch:` —This method is called before the window object calls `touchesBegan:withEvent:` on the gesture recognizer when there are one or more new touches. Return `NO` to prevent the gesture recognizer from seeing the objects representing these touches. (The default value is `YES`.)

In addition, there are two `UIGestureRecognizer` methods (declared in `UIGestureRecognizerSubclass.h`) that effect the same behavior as these delegation methods. A subclass can override these methods to define classwide prevention rules:

```
- (BOOL)canPreventGestureRecognizer:(UIGestureRecognizer
*)preventedGestureRecognizer;
- (BOOL)canBePreventedByGestureRecognizer:(UIGestureRecognizer
*)preventingGestureRecognizer;
```

Permitting Simultaneous Gesture Recognition

By default, no two gesture recognizers can attempt to recognize their gestures simultaneously. But you can change this behavior by implementing `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:`, an optional method of the `UIGestureRecognizerDelegate` protocol. This method is called when the recognition of the receiving gesture recognizer would block the operation of the specified gesture recognizer, or vice versa. Return `YES` to allow both gesture recognizers to recognize their gestures simultaneously.

Note: Returning `YES` is guaranteed to allow simultaneous recognition, but returning `NO` is not guaranteed to prevent simultaneous recognition because the other gesture's delegate may return `YES`.

Regulating the Delivery of Touches to Views

Generally, a window delivers `UITouch` objects (packaged in `UIEvent` objects) to a gesture recognizer before it delivers them to the attached hit-test view. But there are some subtle detours and dead-ends in this general delivery path that depend on whether a gesture is recognized. You can alter this delivery path to suit the requirements of your application.

Default Touch-Event Delivery

By default a window in a multitouch sequence delays the delivery of touch objects in Ended phases to the hit-test view and, if the gesture is recognized, both prevents the delivery of current touch objects to the view and cancels touch objects previously received by the view. The exact behavior depends on the phase of touch objects and on whether a gesture recognizer recognizes its gesture or fails to recognize it in a multitouch sequence.

To clarify this behavior, consider a hypothetical gesture recognizer for a discrete gesture involving two touches (that is, two fingers). Touch objects enter a system and are passed from the `UIApplication` object to the `UIWindow` object for the hit-test view. The following sequence occurs when the gesture is recognized:

1. The window sends two touch objects in the Began phase (`UITouchPhaseBegan`) to the gesture recognizer, which doesn't recognize the gesture. The window sends these same touches to the view attached to the gesture recognizer.

2. The window sends two touch objects in the Moved phase (`UITouchPhaseMoved`) to the gesture recognizer, and the recognizer still doesn't detect its gesture. The window then sends these touches to the attached view.

3. The window sends *one* touch object in the Ended phase (`UITouchPhaseEnded`) to the gesture recognizer. This touch object doesn't yield enough information for the gesture, but the window withholds the object from the attached view.

4. The window sends the other touch object in the Ended phase. The gesture recognizer now recognizes its gesture and so it sets its state to `UIGestureRecognizerStateRecognized`. Just before the first (or only) action message is sent, the view receives a `touchesCancelled:withEvent:` message to invalidate the touch objects previously sent (in the Began and Moved phases). The touches in the Ended phase are canceled.

Now assume that the gesture recognizer in the last step instead decides that this multitouch sequence it's been analyzing is *not* its gesture. It sets its state to `UIGestureRecognizerStateFailed`. The window then sends the two touch objects in the Ended phase to the attached view in a `touchesEnded:withEvent:` message.

A gesture recognizer for a continuous gesture goes through a similar sequence, except that it is more likely to recognize its gesture before touch objects reach the Ended phase. Upon recognizing its gesture, it sets its state to `UIGestureRecognizerStateBegan`. The window sends all subsequent touch objects in the multitouch sequence to the gesture recognizer but not to the attached view.

Note: For a discussion of gesture-recognition states and possible transition between these states, see "State Transitions" (page 56).

Affecting the Delivery of Touches to Views

You can change the values of three `UIGestureRecognizer` properties to alter the default delivery path of touch objects to views in certain ways. These properties and their default values are:

`cancelsTouchesInView` (default of `YES`)

`delaysTouchesBegan` (default of `NO`)

`delaysTouchesEnded` (default of `YES`)

If you change the default values of these properties, you get the following differences in behavior:

- `cancelsTouchesInView` set to `NO` — Causes `touchesCancelled:withEvent:` to *not* be sent to the view for any touches belonging to the recognized gesture. As a result, any touch objects in Began or Moved phases previously received by the attached view are not invalidated.
- `delaysTouchesBegan` set to `YES` — Ensures that when a gesture recognizer recognizes a gesture, no touch objects that were part of that gesture are delivered to the attached view. This setting provides a behavior similar to that offered by the `delaysContentTouches` property on `UIScrollView`; in this case, when scrolling begins soon after the touch begins, subviews of the scroll-view object never receive the touch, so there is no flash of visual feedback. You should be careful about this setting because it can easily make your interface feel unresponsive.
- `delaysTouchesEnded` set to `NO` — Prevents a gesture recognizer that's recognized its gesture after a touch has ended from canceling that touch on the view. For example, say a view has a `UITapGestureRecognizer` object attached with its `numberOfTapsRequired` set to 2, and the user double-taps the view. If this property is set to `NO`, the view gets the following sequence of messages: `touchesBegan:withEvent:`, `touchesEnded:withEvent:`, `touchesBegan:withEvent:`, and `touchesCancelled:withEvent:`. With the property set to `YES`, the view gets `touchesBegan:withEvent:`, `touchesBegan:withEvent:`, `touchesCancelled:withEvent:`, and `touchesCancelled:withEvent:`. The purpose of this property is to ensure that a view won't complete an action as a result of a touch that the gesture will want to cancel later.

Creating Custom Gesture Recognizers

If you are going to create a custom gesture recognizer, you need to have a clear understanding of how gesture recognizers work. The following section gives you the architectural background of gesture recognition, and the subsequent section goes into details of actually creating a gesture recognizer.

State Transitions

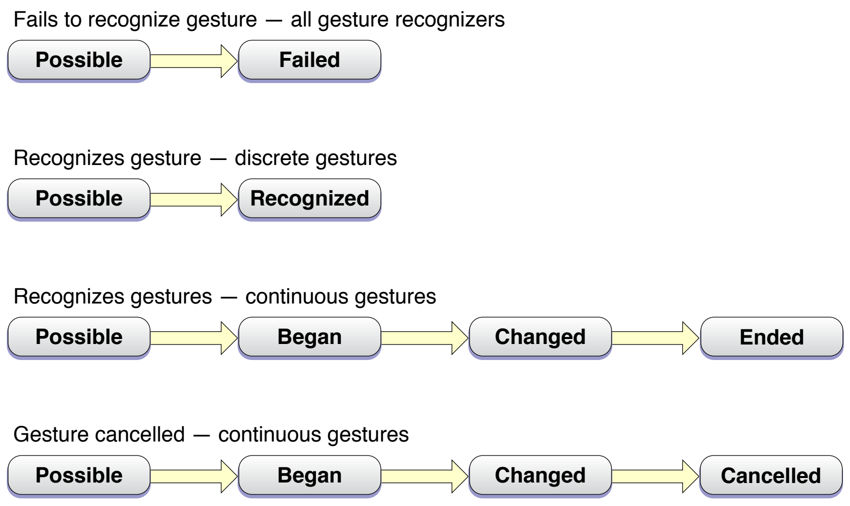
Gesture recognizers operate in a predefined state machine. They transition from one state to another depending on whether certain conditions apply. The following `enum` constants from `UIGestureRecognizer.h` define the states for gesture recognizers:

```
typedef enum {
    UIGestureRecognizerStatePossible,
    UIGestureRecognizerStateBegan,
    UIGestureRecognizerStateChanged,
    UIGestureRecognizerStateEnded,
    UIGestureRecognizerStateCancelled,
    UIGestureRecognizerStateFailed,
    UIGestureRecognizerStateRecognized = UIGestureRecognizerStateEnded
} UIGestureRecognizerState;
```

The sequence of states that a gesture recognizer may transition through varies, depending on whether a discrete or continuous gesture is being recognized. All gesture recognizers start in the Possible state (`UIGestureRecognizerStatePossible`). They then analyze the multitouch sequence targeted at their attached hit-test view, and they either recognize their gesture or fail to recognize it. If a gesture recognizer does not recognize its gesture, it transitions to the Failed state(`UIGestureRecognizerStateFailed`); this is true of all gesture recognizers, regardless of whether the gesture is discrete or continuous.

When a gesture is recognized, however, the state transitions differ for discrete and continuous gestures. A recognizer for a discrete gesture transitions from Possible to Recognized (`UIGestureRecognizerStateRecognized`). A recognizer for a continuous gesture, on the other hand, transitions from Possible to Began (`UIGestureRecognizerStateBegan`) when it first recognizes the gesture. Then it transitions from Began to Changed (`UIGestureRecognizerStateChanged`), and subsequently from Changed to Changed every time there is a change in the gesture. Finally, when the last finger in the multitouch sequence is lifted from the hit-test view, the gesture recognizer transitions to the Ended state (`UIGestureRecognizerStateEnded`), which is an alias for the `UIGestureRecognizerStateRecognized` state. A recognizer for a continuous gesture can also transition from the Changed state to a Cancelled state (`UIGestureRecognizerStateCancelled`) if it determines that the recognized gesture no longer fits the expected pattern for its gesture. Figure 5-3 (page 56) illustrates these transitions.

Figure 5-3 Possible state transitions for gesture recognizers


Fails to recognize gesture — all gesture recognizers
Possible
Failed
Recognizes gesture — discrete gestures
Possible
Recognized
Recognizes gestures — continuous gestures
Possible
Began
Changed
Ended
Gesture cancelled — continuous gestures
Possible
Began
Changed
Cancelled

Note: The Began, Changed, Ended, and Cancelled states are not necessarily associated with `UITouch` objects in corresponding touch phases. They strictly denote the phase of the gesture itself, not the touch objects that are being recognized.

When a gesture is recognized, every subsequent state transition causes an action message to be sent to the target. When a gesture recognizer reaches the Recognized or Ended state, it is asked to reset its internal state in preparation for a new attempt at recognizing the gesture. The `UIGestureRecognizer` class then sets the gesture recognizer's state back to Possible.

Implementing a Custom Gesture Recognizer

To implement a custom gesture recognizer, first create a subclass of `UIGestureRecognizer` in Xcode. Then, add the following import directive in your subclass's header file:

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

Next copy the following method declarations from `UIGestureRecognizerSubclass.h` to your header file; these are the methods you override in your subclass:

```
- (void)reset;
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
```

You must be sure to call the superclass implementation (`super`) in all of the methods you override.

Examine the declaration of the `state` property in `UIGestureRecognizerSubclass.h`. Notice that it is now given a `readwrite` option instead of `readonly` (in `UIGestureRecognizer.h`). Your subclass can now change its state by assigning `UIGestureRecognizerState` constants to the property.

The `UIGestureRecognizer` class sends action messages for you and controls the delivery of touch objects to the hit-test view. You do not need to implement these tasks yourself.

Implementing the Multitouch Event-Handling Methods

The heart of the implementation for a gesture recognizer are the four methods `touchesBegan:withEvent:`, `touchesMoved:withEvent:`, `touchesEnded:withEvent:`, and `touchesCancelled:withEvent:`. You implement these methods much as you would implement them for a custom view.

Note: See "Handling Multi-Touch Events" in *iOS Application Programming Guide* in "Event Handling" for information about handling events delivered during a multitouch sequence.

The main difference in the implementation of these methods for a gesture recognizer is that you transition between states at the appropriate moment. To do this, you must set the value of the state property to the appropriate `UIGestureRecognizerState` constant. When a gesture recognizer recognizes a discrete gesture, it sets the state property to `UIGestureRecognizerStateRecognized`. If the gesture is continuous, it sets the state property first to `UIGestureRecognizerStateBegan`; then, for each change in position of the gesture, it sets (or resets) the property to `UIGestureRecognizerStateChanged`. When the gesture ends, it sets `state` to `UIGestureRecognizerStateEnded`. If at any point a gesture recognizer realizes that this multitouch sequence is not its gesture, it sets its state to `UIGestureRecognizerStateFailed`.

Listing 5-3 is an implementation of a gesture recognizer for a discrete single-touch "checkmark" gesture (actually any V-shaped gesture). It records the midpoint of the gesture—the point at which the upstroke begins—so that clients can obtain this value.

Listing 5-3 Implementation of a "checkmark" gesture recognizer.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesBegan:touches withEvent:event];
    if ([touches count] != 1) {
        self.state = UIGestureRecognizerStateFailed;
        return;
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesMoved:touches withEvent:event];
    if (self.state == UIGestureRecognizerStateFailed) return;
    CGPoint nowPoint = [[touches anyObject] locationInView:self.view];
    CGPoint prevPoint = [[touches anyObject] previousLocationInView:self.view];
    if (!strokeUp) {
        // on downstroke, both x and y increase in positive direction
        if (nowPoint.x >= prevPoint.x && nowPoint.y >= prevPoint.y) {
            self.midPoint = nowPoint;
            // upstroke has increasing x value but decreasing y value
        } else if (nowPoint.x >= prevPoint.x && nowPoint.y <= prevPoint.y) {
            strokeUp = YES;
        } else {
            self.state = UIGestureRecognizerStateFailed;
        }
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesEnded:touches withEvent:event];
    if ((self.state == UIGestureRecognizerStatePossible) && strokeUp) {
        self.state = UIGestureRecognizerStateRecognized;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    [super touchesCancelled:touches withEvent:event];
    self.midPoint = CGPointZero;
    strokeUp = NO;
    self.state = UIGestureRecognizerStateFailed;
}
```

If a gesture recognizer detects a touch (as represented by a `UITouch` object) that it determines is not part of its gesture, it can pass it on directly to its view. To do this, it calls `ignoreTouch:forEvent:` on itself, passing in the touch object. Ignored touches are not withheld from the attached view even if the value of the `cancelsTouchesInView` property is `YES`.

Resetting State

When your gesture recognizer transitions to either the `UIGestureRecognizerStateRecognized` state or the`UIGestureRecognizerStateEnded` state, the `UIGestureRecognizer` class calls the `reset` method of the gesture recognizer just before it winds back the gesture recognizer's state to `UIGestureRecognizerStatePossible`. A gesture recognizer class should implement this method to reset any internal state so that it is ready for a new attempt at recognizing the gesture. After a gesture recognizer returns from this method, it receives no further updates for touches that have already begun but haven't ended.

Listing 5-4 Resetting a gesture recognizer

```
- (void)reset {
    [super reset];
    self.midPoint = CGPointZero;
    strokeUp = NO;
}
```

Graphics and Drawing

In addition to the standard frameworks you use for drawing, iOS 3.2 introduces some new features for generating rendered content. The `UIBezierPath` class is an Objective-C wrapper around a Core Graphics path that makes creating vector-based paths easier. And if you use PDF content, there are now functions that you can use to generate PDF data and save it to a file or data object.

Drawing Shapes Using Bezier Paths

In iOS 3.2, you can now use the `UIBezierPath` class to create vector-based paths. This class is an Objective-C wrapper for the path-related features in the Core Graphics framework. You can use the class to define simple shapes, such as ovals and rectangles, as well as complex complex shapes that incorporate multiple straight and curved line segments.

You use path objects to draw shapes in your application's user interface. You can draw the path's outline, fill the space it encloses, or both. You can also use paths to define a clipping region for the current graphics context, which you can then use to modify subsequent drawing operations in that context.

Bezier Path Basics

A `UIBezierPath` object is a wrapper for a `CGPathRef` data type. **Paths** are vector-based shapes that are built using line and curve segments. You use line segments to create rectangles and polygons and you use curve segments to create arcs, circles, and complex curved shapes. Each segment consists of one or more points (in the current coordinate system) and a drawing command that defines how those points are to be interpreted. The end of one line or curve segment defines the beginning of the next. Each set of connected line and curve segments form what is referred to as a **subpath**. And a single `UIBezierPath` object may contain one or more subpaths that define the overall path.

The processes for building and using a path object are separate. Building the path is the first process and involves the following steps:

1. Create the path object.
2. Set the starting point of the initial segment using the `moveToPoint:` method.
3. Add line and curve segments to define one or more subpaths.
4. Modify any relevant drawing attributes of your `UIBezierPath` object. For example, you might set the `lineWidth` or `lineJoinStyle` properties for stroked paths or the `usesEvenOddFillRule` property for filled paths. You can always change these values later as needed.

When building your path, you should arrange the points of your path relative to the origin point (0, 0). Doing so makes it easier to move the path around later. During drawing, the points of your path are applied as-is to the coordinate system of the current graphics context. If your path is oriented relative to the origin, all you have to do to reposition it is apply an affine transform with a translation factor to the current graphics context. The advantage of modifying the graphics context (as opposed to the path object itself) is that you can easily undo the transformation by saving and restoring the graphics state.

To draw your path object, you use the `stroke` and `fill` methods. These methods render the line and curve segments of your path in the current graphics context. The rendering process involves rasterizing the line and curve segments using the attributes of the path object. The rasterization process does not modify the path object itself. As a result, you can render the same path object multiple times in the current context.

Adding Lines and Polygons to Your Path

Lines and polygons are simple shapes that you build point-by-point using the `moveToPoint:` and `addLineToPoint:` methods. The `moveToPoint:` method sets the starting point of the shape you want to create. From that point, you create the lines of the shape using the `addLineToPoint:` method. You create the lines in succession, with each line being formed between the previous point and the new point you specify.

Listing 6-1 shows the code needed to create a pentagon shape using individual line segments. This code sets the initial point of the shape and then adds four connected line segments. The fifth segment is added by the call to the `closePath` method, which connects the last point (0, 40) with the first point (100, 0).

Listing 6-1 Creating a pentagon shape

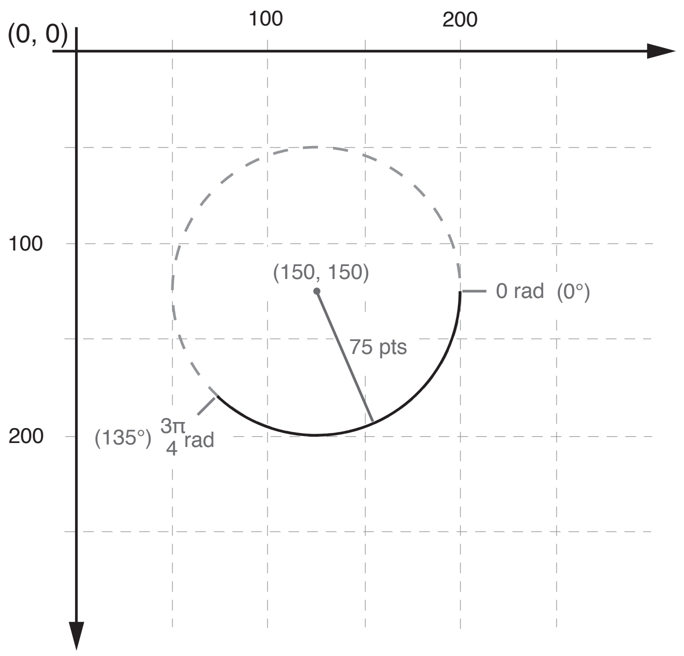
```
UIBezierPath*    aPath = [UIBezierPath bezierPath];

// Set the starting point of the shape.
[aPath moveToPoint:CGPointMake(100.0, 0.0)];

// Draw the lines
[aPath addLineToPoint:CGPointMake(200.0, 40.0)];
[aPath addLineToPoint:CGPointMake(160, 140)];
[aPath addLineToPoint:CGPointMake(40.0, 140)];
[aPath addLineToPoint:CGPointMake(0.0, 40.0)];
[aPath closePath];
```

Using the `closePath` method not only ends the shape, it also draws a line segment between the first and last points. This is a convenient way to finish a polygon without having to draw the final line.

Adding Arcs to Your Path

The `UIBezierPath` class provides support for initializing a new path object with an arc segment. The parameters of the `bezierPathWithArcCenter:radius:startAngle:endAngle:clockwise:` method define the circle that contains the desired arc and the start and end points of the arc itself. Figure 6-1 shows the components that go into creating an arc, including the circle that defines the arc and the angle measurements used to specify it. In this case, the arc is created in the clockwise direction. (Drawing the arc in the counterclockwise direction would paint the dashed portion of the circle instead.) The code for creating this arc is shown in Listing 6-2 (page 63).

Figure 6-1 An arc in the default coordinate system



Listing 6-2 Creating a new arc path

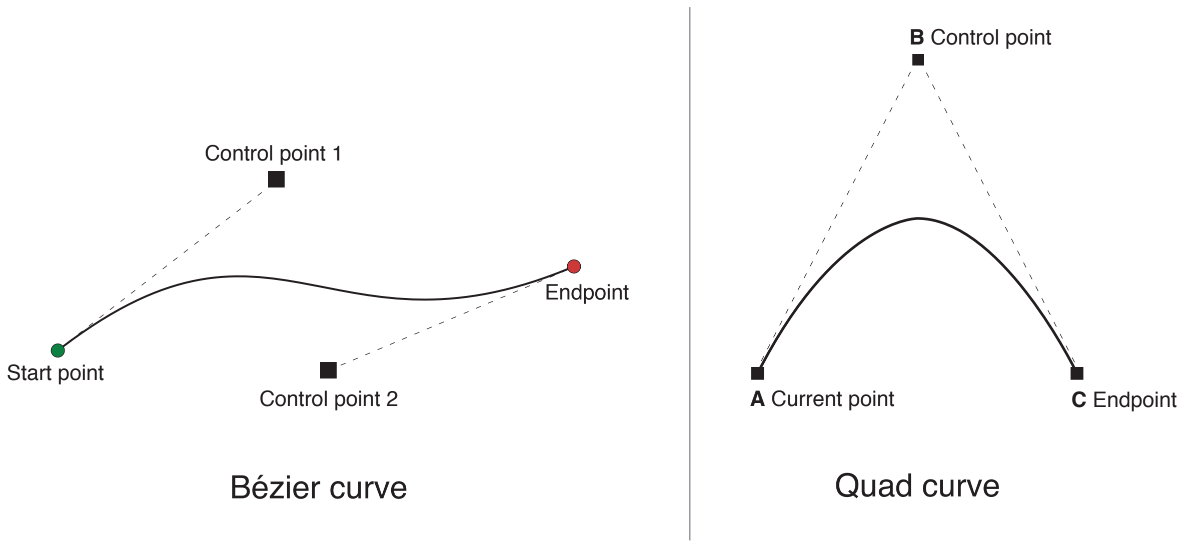
```
// pi is approximately equal to 3.14159265359
#define   DEGREES_TO_RADIANS(degrees)  ((pi * degrees)/ 180)

- (UIBezierPath*)createArcPath
{
   UIBezierPath* aPath = [UIBezierPath bezierPathWithArcCenter:CGPointMake(150,
 150)
                            radius:75
                            startAngle:0
                            endAngle:DEGREES_TO_RADIANS(135)
                            clockwise:YES];
   return aPath;
}
```

If you want to incorporate an arc segment into the middle of a path, you must modify the path object's `CGPathRef` data type directly. For more information about modifying the path using Core Graphics functions, see "Modifying the Path Using Core Graphics Functions" (page 65).

Adding Curves to Your Path

The `UIBezierPath` class provides support for adding cubic and quadratic Bézier curves to a path. Curve segments start at the current point and end at the point you specify. The shape of the curve is defined using tangent lines between the start and end points and one or more control points. Figure 6-2 shows approximations of both types of curve and the relationship between the control points and the shape of the curve. The exact curvature of each segment involves a complex mathematical relationship between all of the points and is well documented online and at Wikipedia.

Figure 6-2 Curve segments in a path



To add curves to a path, you use the following methods:

- **Cubic curve:** `addCurveToPoint:controlPoint1:controlPoint2:`
- **Quadratic curve:** `addQuadCurveToPoint:controlPoint:`

Because curves rely on the current point of the path, you must set the current point before calling either of the preceding methods. Upon completion of the curve, the current point is updated to the new end point you specified.

Creating Oval and Rectangular Paths

Ovals and rectangles are common types of paths that are built using a combination of curve and line segments. The `UIBezierPath` class includes the `bezierPathWithRect:` and `bezierPathWithOvalInRect:` convenience methods for creating paths with oval or rectangular shapes. Both of these methods create a new path object and initialize it with the specified shape. You can use the returned path object right away or add more shapes to it as needed.

If you want to add a rectangle to an existing path object, you must do so using the `moveToPoint:`, `addLineToPoint:`, and `closePath` methods as you would for any other polygon. Using the `closePath` method for the final side of the rectangle is a convenient way to add the final line of the path and also mark the end of the rectangle subpath.

If you want to add an oval to an existing path, the simplest way to do so is to use Core Graphics. Although you can use the `addQuadCurveToPoint:controlPoint:` to approximate an oval surface, the `CGPathAddEllipseInRect` function is much simpler to use and more accurate. For more information, see "Modifying the Path Using Core Graphics Functions" (page 65).

Modifying the Path Using Core Graphics Functions

The `UIBezierPath` class is really just a wrapper for a `CGPathRef` data type and the drawing attributes associated with that path. Although you normally add line and curve segments using the methods of the `UIBezierPath` class, the class also exposes a `CGPath` property that you can use to modify the underlying path data type directly. You can use this property when you would prefer to build your path using the functions of the Core Graphics framework.

There are two ways to modify the path associated with a `UIBezierPath` object. You can modify the path entirely using Core Graphics functions, or you can use a mixture of Core Graphics functions and `UIBezierPath` methods. Modifying the path entirely using Core Graphics calls is easier in some ways. You create a mutable `CGPathRef` data type and call whatever functions you need to modify its path information. When you are done you assign your path object to the corresponding `UIBezierPath` object, as shown in Listing 6-3.

Listing 6-3 Assigning a new `CGPathRef` to a `UIBezierPath` object

```
// Create the path data
CGMutablePathRef cgPath = CGPathCreateMutable();
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(0, 0, 300, 300));
CGPathAddEllipseInRect(cgPath, NULL, CGRectMake(50, 50, 200, 200));

// Now create the UIBezierPath object
UIBezierPath* aPath = [UIBezierPath bezierPath];
aPath.CGPath = cgPath;
aPath.usesEvenOddFillRule = YES;

// After assigning it to the UIBezierPath object, you can release
// your CGPathRef data type safely.
CGPathRelease(cgPath);
```

If you choose to use a mixture of Core Graphics functions and `UIBezierPath` methods, you must carefully move the path information back and forth between the two. Because a `UIBezierPath` object owns its underlying `CGPathRef` data type, you cannot simply retrieve that type and modify it directly. Instead, you must make a mutable copy, modify the copy, and then assign the copy back to the `CGPath` property as shown in Listing 6-4.

Listing 6-4 Mixing Core Graphics and `UIBezierPath` calls

```
UIBezierPath*    aPath = [UIBezierPath bezierPathWithOvalInRect:CGRectMake(0,
0, 300, 300)];

// Get the CGPathRef and create a mutable version.
CGPathRef cgPath = aPath.CGPath;
CGMutablePathRef  mutablePath = CGPathCreateMutableCopy(cgPath);

// Modify the path and assign it back to the UIBezierPath object
CGPathAddEllipseInRect(mutablePath, NULL, CGRectMake(50, 50, 200, 200));
aPath.CGPath = mutablePath;

// Release both the mutable and immutable copies of the path.
CGPathRelease(cgPath);
CGPathRelease(mutablePath);
```

Rendering the Contents of a Bezier Path Object

After creating a `UIBezierPath` object, you can render it in the current graphics context using its `stroke` and `fill` methods. Before you call these methods, though, there are usually a few other tasks to perform to ensure your path is drawn correctly:

- Set the desired stroke and fill colors using the methods of the `UIColor` class.
- Position the shape where you want it in the target view.

  If you created your path relative to the point (0, 0), you can apply an appropriate affine transform to the current drawing context. For example, to draw your shape starting at the point (10, 10), you would call the `CGContextTranslateCTM` function and specify `10` for both the horizontal and vertical translation values. Adjusting the graphics context (as opposed to the points in the path object) is preferred because you can undo the change more easily by saving and restoring the previous graphics state.
- Update the drawing attributes of the path object. The drawing attributes of your `UIBezierPath` instance override the values associated with the graphics context when rendering the path.

Listing 6-5 shows a sample implementation of a `drawRect:` method that draws an oval in a custom view. The upper-left corner of the oval's bounding rectangle is located at the point (50, 50) in the view's coordinate system. Because fill operations paint right up to the path boundary, this method fills the path before stroking it. This prevents the fill color from obscuring half of the stroked line.

Listing 6-5 Drawing a path in a view

```
- (void)drawRect:(CGRect)rect
{
    // Create an oval shape to draw.
    UIBezierPath* aPath = [UIBezierPath bezierPathWithOvalInRect:
                                CGRectMake(0, 0, 200, 100)];

    // Set the render colors
    [[UIColor blackColor] setStroke];
    [[UIColor redColor] setFill];

    CGContextRef aRef = UIGraphicsGetCurrentContext();

    // If you have content to draw after the shape,
    // save the current state before changing the transform
    //CGContextSaveGState(aRef);

    // Adjust the view's origin temporarily. The oval is
    // now drawn relative to the new origin point.
    CGContextTranslateCTM(aRef, 50, 50);

    // Adjust the drawing options as needed.
    aPath.lineWidth = 5;

    // Fill the path before stroking it so that the fill
    // color does not obscure the stroked line.
    [aPath fill];
    [aPath stroke];

    // Restore the graphics state before drawing any other content.
    //CGContextRestoreGState(aRef);
}
```

Doing Hit-Detection on a Path

To determine whether a touch event occurred on the filled portion of a path, you can use the `containsPoint:` method of `UIBezierPath`. This method tests the specified point against all closed subpaths in the path object and returns `YES` if it lies on or inside any of those subpaths.

Important: The `containsPoint:` method and the Core Graphics hit-testing functions operate only on closed paths. These methods always return `NO` for hits on open subpaths. If you want to do hit detection on an open subpath, you must create a copy of your path object and close the open subpaths before testing points.

If you want to do hit-testing on the stroked portion of the path (instead of the fill area), you must use Core Graphics. The `CGContextPathContainsPoint` function lets you test points on either the fill or stroke portion of the path currently assigned to the graphics context. Listing 6-6 shows a method that tests to see whether the specified point intersects the specified path. The *`inFill`* parameter lets the caller specify whether the point should be tested against the filled or stroked portion of the path. The path passed in by the caller must contain one or more closed subpaths for the hit detection to succeed.

Listing 6-6 Testing points against a path object

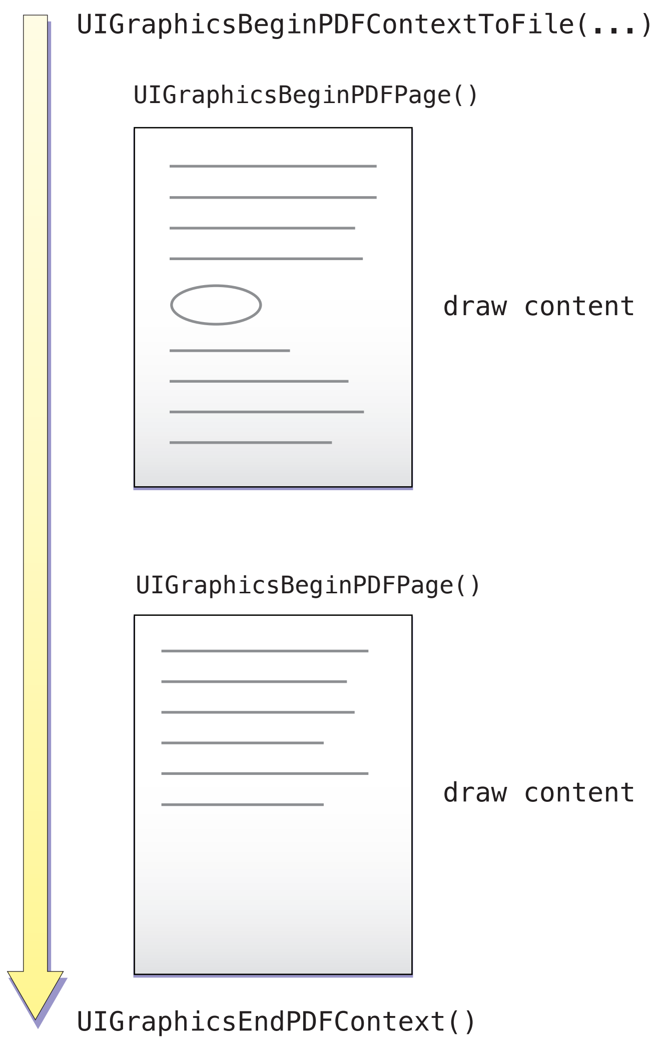
```
- (BOOL)containsPoint:(CGPoint)point onPath:(UIBezierPath*)path
inFillArea:(BOOL)inFill
{
   CGContextRef context = UIGraphicsGetCurrentContext();
   CGPathRef cgPath = path.CGPath;
   BOOL    isHit = NO;

   // Determine the drawing mode to use. Default to
   // detecting hits on the stroked portion of the path.
   CGPathDrawingMode mode = kCGPathStroke;
   if (inFill)
   {
      // Look for hits in the fill area of the path instead.
      if (path.usesEvenOddFillRule)
         mode = kCGPathEOFill;
      else
         mode = kCGPathFill;
   }

   // Save the graphics state so that the path can be
   // removed later.
   CGContextSaveGState(context);
   CGContextAddPath(context, cgPath);

   // Do the hit detection.
   isHit = CGContextPathContainsPoint(context, point, mode);

   CGContextRestoreGState(context);

   return isHit;
}
```

Generating PDF Content

In iOS 3.2, the UIKit framework provides a set of functions for generating PDF content using native drawing code. These functions let you create a graphics context that targets a PDF file or PDF data object. You can then draw into this graphics context using the same UIKit and Core Graphics drawing routines you use when drawing to the screen. You can create any number of pages for the PDF, and when you are done, what you are left with is a PDF version of what you drew.

Figure 6-3 shows the workflow for creating a PDF file on the local filesystem. The `UIGraphicsBeginPDFContextToFile` function creates the PDF context and associates it with a filename. After creating the context, you open the first page using the `UIGraphicsBeginPDFPage` function. Once you have a page, you can begin drawing your content for it. To create new pages, simply call `UIGraphicsBeginPDFPage` again and begin drawing. When you are done, calling the `UIGraphicsEndPDFContext` function closes the graphics context and writes the resulting data to the PDF file.

Figure 6-3 Workflow for creating a PDF document



The following sections describe the PDF creation process in more detail using a simple example. For information about the functions you use to create PDF content, see *UIKit Function Reference*.

Creating and Configuring the PDF Context

You create a PDF graphics context using either the `UIGraphicsBeginPDFContextToData` or `UIGraphicsBeginPDFContextToFile` function. These functions create the graphics context and associate it with a destination for the PDF data. For the `UIGraphicsBeginPDFContextToData` function, the destination is an `NSMutableData` object that you provide. And for the `UIGraphicsBeginPDFContextToFile` function, the destination is a file in your application's home directory.

PDF documents organize their content using a page-based structure. This structure imposes two restrictions on any drawing you do:

- There must be an open page before you issue any drawing commands.
- You must specify the size of each page.

The functions you use to create a PDF graphics context allow you to specify a default page size but they do not automatically open a page. After creating your context, you must explicitly open a new page using either the `UIGraphicsBeginPDFPage` or `UIGraphicsBeginPDFPageWithInfo` function. And each time you want to create a new page, you must call one of these functions again to mark the start of the new page. The `UIGraphicsBeginPDFPage` function creates a page using the default size, while the `UIGraphicsBeginPDFPageWithInfo` function lets you customize the page size and other page attributes.

When you are done drawing, you close the PDF graphics context by calling the `UIGraphicsEndPDFContext`. This function closes the last page and writes the PDF content to the file or data object you specified at creation time. This function also removes the PDF context from the graphics context stack.

Listing 6-7 shows the processing loop used by an application to create a PDF file from the text in a text view. Aside from three function calls to configure and manage the PDF context, most of the code is related to drawing the desired content. The `textView` member variable points to the `UITextView` object containing the desired text. The application uses the Core Text framework (and more specifically a `CTFramesetterRef` data type) to handle the text layout and management on successive pages. The implementations for the custom `renderPageWithTextRange:andFramesetter:` and `drawPageNumber:` methods are shown in Listing 6-8 (page 71).

Listing 6-7 Creating a new PDF file

```
- (IBAction)savePDFFile:(id)sender
{
   // This is a custom method to retrieve the name of the PDF file
   NSString* pdfFileName = [self getPDFFileName];

   // Create the PDF context using the default page size of 612 x 792.
   UIGraphicsBeginPDFContextToFile(pdfFileName, CGRectZero, nil);

   // Prepare the text and create a CTFramesetter to handle the layout.
   CFAttributedStringRef currentText = CFAttributedStringCreate(NULL,
(CFStringRef)textView.text, NULL);
   CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString(currentText);
   if (!framesetter)
      return;
   // Set up some local variables.
   CFRange currentRange = CFRangeMake(0, 0);
   NSInteger currentPage = 0;
   BOOL done = NO;

   // Begin the main loop to create the individual pages
   do
   {
      // Mark the beginning of a new page.
      UIGraphicsBeginPDFPageWithInfo(CGRectMake(0, 0, 612, 792), nil);

      // Draw a page number at the bottom of each page
      currentPage++;
      [self drawPageNumber:currentPage];

      // Render the current page and update the current range to
      // point to the beginning of the next page.
      currentRange = [self renderPageWithTextRange:currentRange
andFramesetter:framesetter];

      // If we're at the end of the text, exit the loop.
      if (currentRange.location ==
CFAttributedStringGetLength((CFAttributedStringRef)currentText))
         done = YES;
   }
   while (!done);

   // Close the PDF context and write the contents out.
   UIGraphicsEndPDFContext();

   // Clean up.
   CFRelease(currentText);
   CFRelease(framesetter);
}
```

Drawing PDF Pages

All PDF drawing must be done in the context of a page. Every PDF document has at least one page and many may have multiple pages. You specify the start of a new page by calling the `UIGraphicsBeginPDFPage` or `UIGraphicsBeginPDFPageWithInfo` function. These functions close the previous page (if one was open), create a new page, and prepare it for drawing. The `UIGraphicsBeginPDFPage` creates the new page using the default size while the `UIGraphicsBeginPDFPageWithInfo` function lets you customize the page size or customize other aspects of the PDF page.

After you create a page, all of your subsequent drawing commands are captured by the PDF graphics context and translated into PDF commands. You can draw anything you want in the page, including text, vector shapes, and images just as you would in your application's custom views. The drawing commands you issue are captured by the PDF context and translated into PDF data. Placement of content on the the page is completely up to you but must take place within the bounding rectangle of the page.

Listing 6-8 shows two custom methods used to draw content inside a PDF page. The `renderPageWithTextRange:andFramesetter:` method uses Core Text to create a text frame that fits the page and then lay out some text inside that frame. After laying out the text, it returns an updated range that reflects the end of the current page and the beginning of the next page. The `drawPageNumber:` method uses the `NSString` drawing capabilities to draw a page number string at the bottom of each PDF page.

Listing 6-8 Drawing page-based content

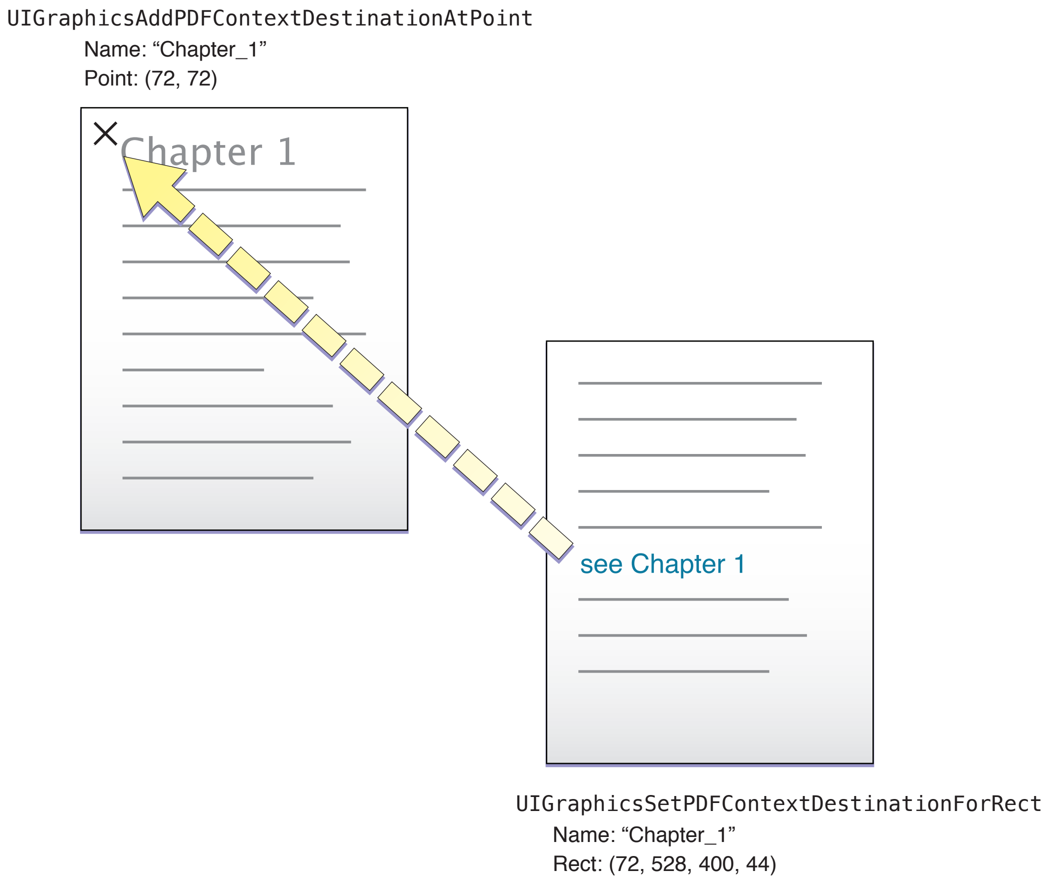
```
// Use Core Text to draw the text in a frame on the page.
- (CFRange)renderPage:(NSInteger)pageNum withTextRange:(CFRange)currentRange
                andFramesetter:(CTFramesetterRef)framesetter
{
   // Get the graphics context.
   CGContextRef    currentContext = UIGraphicsGetCurrentContext();

   // Put the text matrix into a known state. This ensures
   // that no old scaling factors are left in place.
   CGContextSetTextMatrix(context, CGAffineTransformIdentity);

   // Create a path object to enclose the text. Use 72 point
   // margins all around the text.
   CGRect    frameRect = CGRectMake(72, 72, 468, 648);
   CGMutablePathRef framePath = CGPathCreateMutable();
   CGPathAddRect(framePath, NULL, frameRect);

   // Get the frame that will do the rendering.
  // The currentRange variable specifies only the starting point. The framesetter
   // lays out as much text as will fit into the frame.
   CTFrameRef frameRef = CTFramesetterCreateFrame(framesetter, currentRange,
framePath, NULL);
   CGPathRelease(framePath);

   // Core Text draws from the bottom-left corner up, so flip
   // the current transform prior to drawing.
   CGContextTranslateCTM(currentContext, 0, 792);
   CGContextScaleCTM(currentContext, 1.0, -1.0);

   // Draw the frame.
   CTFrameDraw(frameRef, currentContext);

   // Update the current range based on what was drawn.
   currentRange = CTFrameGetVisibleStringRange(frameRef);
   currentRange.location += currentRange.length;
   currentRange.length = 0;
   CFRelease(frameRef);

   return currentRange;
}

- (void)drawPageNumber:(NSInteger)pageNum
{
   NSString* pageString = [NSString stringWithFormat:@"Page %d", pageNum];
   UIFont* theFont = [UIFont systemFontOfSize:12];
   CGSize maxSize = CGSizeMake(612, 72);

   CGSize pageStringSize = [pageString sizeWithFont:theFont
                                    constrainedToSize:maxSize
                                    lineBreakMode:UILineBreakModeClip];
   CGRect stringRect = CGRectMake(((612.0 - pageStringSize.width) / 2.0),
                               720.0 + ((72.0 - pageStringSize.height) / 2.0)
 ,
                                pageStringSize.width,
                                pageStringSize.height);

   [pageString drawInRect:stringRect withFont:theFont];
}
```

Creating Links Within Your PDF Content

Besides drawing content, you can also include links that take the user to another page in the same PDF file or to an external URL. To create a single link, you must add a source rectangle and a link destination to your PDF pages. One of the attributes of the link destination is a string that serves as the unique identifier for that link. To create a link to a specific destination, you specify the unique identifier for that destination when creating the source rectangle.

To add a new link destination to your PDF content, you use the `UIGraphicsAddPDFContextDestinationAtPoint` function. This function associates a named destination with a specific point on the current page. When you want to link to that destination point, you use `UIGraphicsSetPDFContextDestinationForRect` function to specify the source rectangle for the link. Figure 6-4 shows the relationship between these two function calls when applied to the pages of your PDF documents. Tapping on the rectangle surrounding the "see Chapter 1" text takes the user to the corresponding destination point, which is located at the top of Chapter 1.

Figure 6-4 Creating a link destination and jump point



In addition to creating links within a document, you can also use the `UIGraphicsSetPDFContextURLForRect` function to create links to content located outside of the document. When using this function to create links, you do not need to create a link destination first. All you have to do is use this function to specify the target URL and the source rectangle on the current page.

Custom Text Processing and Input

With the larger screen of the iPad, not only simple text entry but complex text processing and custom input are now compelling possibilities for many applications. Applications can have features such as custom text layout, font management, autocorrection, custom keyboards, spell-checking, selection-based modification, and multistage input. iOS 3.2 includes several technologies that make these features realizable. This chapter describes these technologies and tells you what you need to do to incorporate them in your applications.

Input Views and Input Accessory Views

The UIKit framework includes support for custom input views and input accessory views. Your application can substitute its own input view for the system keyboard when users edit text or other forms of data in a view. For example, an application could use a custom input view to enter characters from a runic alphabet. You may also attach an input accessory view to the system keyboard or to a custom input view; this accessory view runs along the top of the main input view and can contain, for example, controls that affect the text in some way or labels that display some information about the text.

To get this feature if your application is using `UITextView` and `UITextField` objects for text editing, simply assign custom views to the `inputView` and `inputAccessoryView` properties. Those custom views are shown when the text object becomes first responder.

You are not limited to input views and input accessory views in framework-supplied text objects. Any class inheriting directly or indirectly from `UIResponder` (usually a custom view) can specify its own input view and input accessory view. The `UIResponder` class declares two properties for input views and input accessory views:

```
@property (readonly, retain) UIView *inputView;
@property (readonly, retain) UIView *inputAccessoryView;
```

When the responder object becomes the first responder and `inputView` (or `inputAccessoryView`) is not `nil`, UIKit animates the input view into place below the parent view (or attaches the input accessory view to the top of the input view). The first responder can reload the input and accessory views by calling the `reloadInputViews` method of `UIResponder`.

The `UITextView` class redeclares the `inputView` and `inputAccessoryView` properties as `readwrite`. Clients of `UITextView` objects need only obtain the input and input-accessory views—either by loading a nib file or creating the views in code—and assign them to their properties. Custom view classes (and other subclasses that inherit from `UIResponder`) should redeclare one or both of these properties and their backing instance variables and override the getter method for the property—that is, don't synthesize the properties' accessor methods. In their getter-method implementations, they should return it the view, loading or creating it if it doesn't already exist.

You have a lot of flexibility in defining the size and content of an input view or input accessory view. Although the height of these views can be what you'd like, they should be the same width as the system keyboard. If UIKit encounters an input view with an`UIViewAutoresizingFlexibleHeight` value in its autoresizing mask, it changes the height to match the keyboard. There are no restrictions on the number of subviews (such as controls) that input views and input accessory views may have. For more guidance on input views and input accessory views, see *iPad Human Interface Guidelines*.

To load a nib file at run time, first create the input view or input accessory view in Interface Builder. Then at runtime get the application's main bundle and call `loadNibNamed:owner:options:` on it, passing the name of the nib file, the File's Owner for the nib file, and any options. This method returns an array of the top-level objects in the nib, which includes the input view or input accessory view. Assign the view to its corresponding property. For more on this subject, see "Nib Files" in *Resource Programming Guide*.

Listing 7-1 illustrates a custom view class lazily creating its input accessory view in the `inputAccessoryView` getter method.

Listing 7-1 Creating an input accessory view programmatically

```
- (UIView *)inputAccessoryView {
    if (!inputAccessoryView) {
        CGRect accessFrame = CGRectMake(0.0, 0.0, 768.0, 77.0);
        inputAccessoryView = [[UIView alloc] initWithFrame:accessFrame];
        inputAccessoryView.backgroundColor = [UIColor blueColor];
       UIButton *compButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
        compButton.frame = CGRectMake(313.0, 20.0, 158.0, 37.0);
       [compButton setTitle: @"Word Completions" forState:UIControlStateNormal];
        [compButton setTitleColor:[UIColor blackColor]
forState:UIControlStateNormal];
        [compButton addTarget:self action:@selector(completeCurrentWord:)
            forControlEvents:UIControlEventTouchUpInside];
        [inputAccessoryView addSubview:compButton];
    }
    return inputAccessoryView;
}
```

Just as it does with the system keyboard, UIKit posts `UIKeyboardWillShowNotification`, `UIKeyboardDidShowNotification`, `UIKeyboardWillHideNotification`, and `UIKeyboardDidHideNotification` notifications. The object observing these notifications can get geometry information related to the input view and input accessory view and adjust the edited view accordingly.

Simple Text Input

You can implement custom views that allow users to enter text at an insertion point and delete characters before that insertion point when they tap the Delete key. An instant-messaging application, for example, could have a view that allows users to enter their part of a conversation.

You can acquire this capability for simple text entry by subclassing `UIView` or any other view class that inherits from `UIResponder` and adopting the `UIKeyInput` protocol. When an instance of your view class becomes first responder, UIKit displays the system keyboard. `UIKeyInput` itself adopts the `UITextInputTraits` protocol, so you can set keyboard type, return-key type, and other attributes of the keyboard.

Note: Only a small subset of the available keyboards and languages are available to classes that adopt the `UIKeyInput` protocol.

To adopt `UIKeyInput`, you must implement the three methods it declares: `hasText`, `insertText:`, and `deleteBackward`. To do the actual drawing of the text, you may use any of the technologies summarized in "Facilities for Text Drawing and Text Processing" (page 80). However, for simple text input, such as for a single line of text in a custom control, the `UIStringDrawing` and `CATextLayer` APIs are most appropriate.

Listing 7-2 illustrates the `UIKeyInput` implementation of a custom view class. The `textStore` property in this example is an `NSMutableString` object that serves as the backing store of text. The implementation either appends or removes the last character in the string (depending on whether an alphanumeric key or the delete key is pressed) and then redraws `textStore`.

Listing 7-2 Implementing simple text entry

```
- (BOOL)hasText {
    if (textStore.length > 0) {
        return YES;
    }
    return NO;
}

- (void)insertText:(NSString *)theText {
    [self.textStore appendString:theText];
    [self setNeedsDisplay];
}

- (void)deleteBackward {
    NSRange theRange = NSMakeRange(self.textStore.length-1, 1);
    [self.textStore deleteCharactersInRange:theRange];
    [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect {
    CGRect rectForText = [self rectForTextWithInset:2.0]; // custom method
    [self.theColor set];
    UIRectFrame(rect);
    [self.textStore drawInRect:rectForText withFont:self.theFont];
}
```

Note that this code uses the `drawInRect:withFont:` from the `UIStringDrawing` category on `NSString` to actually draw the text in the view. See "Facilities for Text Drawing and Text Processing" (page 80) for more about `UIStringDrawing`.

Communicating with the Text Input System

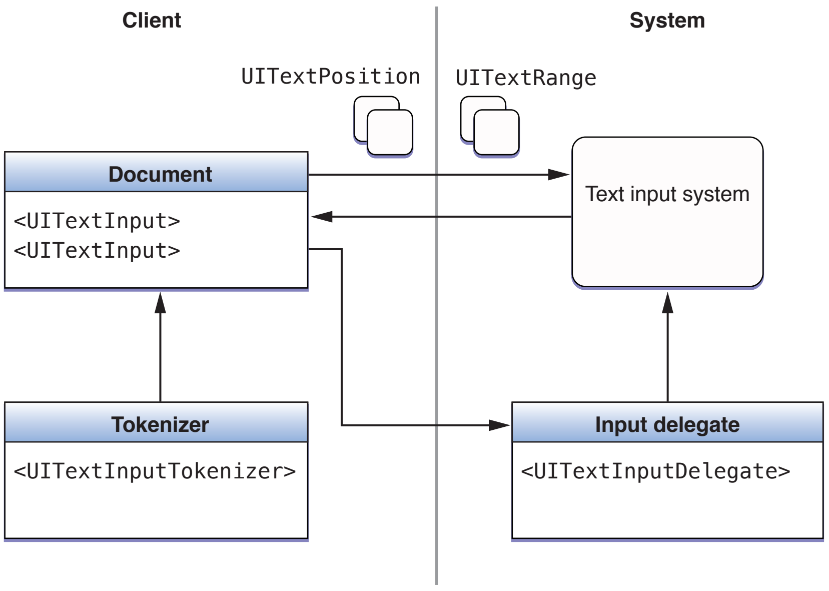
The text input system of iOS manages the keyboard, interpreting taps as presses of specific keys in specific keyboards suitable for certain languages and sending the associated character to the target view for insertion. As explained in "Simple Text Input" (page 76), view classes must adopt the `UIKeyInput` protocol to insert and delete characters at the caret (insertion point).

However, the text input system does more than simple text entry. It also manages autocorrection, and multistage input, which are all based upon the current selection and context. Multistage text input is required for ideographic languages such as hanji (Japanese) and hanzi (Chinese) that take input from phonetic keyboards. To acquire these features, a custom text view must communicate with the text input system by adopting the `UITextInput` protocol and implementing the related client-side classes and protocols.

Overview of the Client Side of Text Input

A class of a text document must adopt the `UITextInput` protocol to communicate fully with the text input system. The class needs to inherit from `UIResponder` and is in most cases a custom view. It must implement its own text layout and font management; for this purpose, the Core Text framework is recommended. ("Facilities for Text Drawing and Text Processing" (page 80) gives an overview of Core Text.) The class should also adopt and implement the `UIKeyInput` protocol, although it does inherit the default implementation of the `UITextInputTraits` protocol.

Figure 7-1 Paths of communication with the text input system



The `UITextInput` methods that the text document implements are called by the text input system. Many of these methods request text-position and text-range objects from the text document and pass text-position and text-range objects back to the text document in other method calls. The reasons for these exchanges of text positions and text ranges are summarized in "Tasks of a UITextInput Object" (page 79).

These text-position and text-range objects are custom objects that for the document represent locations and ranges in its displayed text. "Text Positions and Text Ranges" (page 79) discusses these objects in more detail.

The `UITextInput`-conforming document also maintains references to a tokenizer and an input delegate. The document calls methods declared by the `UITextInputDelegate` protocol to notify a system-provided input delegate about changes in text and selection. It also communicates with a tokenizer object to determine the granularity of text units—for example, character, word, and paragraph. The tokenizer is an object that adopts the `UITextInputTokenizer` protocol.

Text Positions and Text Ranges

The client application must create two classes whose instances represent positions and ranges in the text of a document. These classes must be subclasses of `UITextPosition` and `UITextRange`, respectively.

Although `UITextPosition` itself declares no interface, it is an essential part of the information exchanged between a text document and the text input system. The text system requires an object to represent a location in the text instead of, say, an integer or a structure. Moreover, a `UITextPosition` object can serve a practical purpose by representing a position in the visible text when the string backing the text has a different offset to that position. This happens when the string contains invisible formatting characters, such as with RTF and HTML documents, or embedded objects, such as an attachment. The custom `UITextPosition` class could account for these invisible characters when locating the string offsets of visible characters. In the simplest case—a plain text document with no embedded objects—a custom `UITextPosition` object could encapsulate a single offset integer.

`UITextRange` declares a simple interface in which two of its properties are starting and ending custom `UITextPosition` objects. The third property holds a Boolean value that indicates whether the range is empty (that is, has no length).

Tasks of a UITextInput Object

A text-document class adopting the `UITextInput` protocol is required to implement most of the protocol's methods and properties. With a few exceptions, these methods take custom `UITextPosition` or `UITextRange` objects as parameters or return one of these objects. At runtime the text system invokes these methods and, again in almost all cases, expects some object or value back.

The methods implemented by a `UITextInput` objects can be divided into distinctive tasks:

- **Computing text ranges and text positions.** Create and return a `UITextRange` object (or, simply, a text range) given two text positions; or create and return a `UITextPosition` object (or, simply, a text position) given a text position and an offset.
- **Evaluating text positions.** Compare two text positions or return the offset from one text position to another.
- **Answering layout questions.** Determine a text position or text range by extending in a given layout direction.
- **Hit-testing.** Given a point, return the closest text position or text range.
- **Returning rectangles for text ranges and text positions.** Return the rectangle that encloses a text range or the rectangle at the text position of the caret.
- **Returning and setting text by text range.**

In addition, the `UITextInput` object must maintain the range of the currently selected text and the range of the currently marked text, if any. Marked text, which is part of multistage text input, represents provisionally inserted text the user has yet to confirm. It is styled in a distinctive way. The range of marked text always contains within it a range of selected text, which might be a range of characters or the caret.

The `UITextInput` object might also choose to implement one or more optional protocol methods. These enable it to return text styles (font, text color, background color) beginning at a specified text position and to reconcile visible text position and character offset (for those `UITextPosition` objects where these values are not the same).

At appropriate junctures, the `UITextInput` object should send `textWillChange:`, `textDidChange:`, `selectionWillChange:`, and `selectionDidChange:` messages to the input delegate (which it holds a reference to).

Tokenizers

Tokenizers are objects that can determine whether a text position is within or at the boundary of a text unit with a given granularity. A tokenizer returns ranges of text units with the granularity or the boundary text position for a text unit with the granularity. Currently defined granularities are character, word, sentence, paragraph, line, and document; `enum` constants of the `UITextGranularity` type represent these granularities. Granularities of text units are always evaluated with reference to a storage or layout direction.

A tokenizer is an instance of a class that conforms to the `UITextInputTokenizer` protocol. The `UITextInputStringTokenizer` class provides a default base implementation of the `UITextInputTokenizer` protocol that is suitable for western-language keyboards. If you require a tokenizer with an entirely new interpretation of text units of varying granularity, you should adopt `UITextInputTokenizer` and implement all of its methods. If instead you need only to specify line granularities and directions affected by layout (left, right, up, and down), you should subclass `UITextInputStringTokenizer`.

When you initialize a `UITextInputStringTokenizer` object, you must supply it with the `UITextInput` object. In turn, the `UITextInput` object should lazily create its tokenizer object in the getter method of the tokenizer property.

Facilities for Text Drawing and Text Processing

The UIKit framework includes several classes whose main purpose is to display text in an application's user interface: `UITextField`, `UILabel`, `UITextView`, and `UIWebView`. You might have an application, however, that requires greater flexibility than these classes afford; in other words, you want greater control over where and how your application draws and manipulates text. For these situations, iOS makes available programmatic interfaces from from the Core Text, Core Graphics, and Core Animation frameworks as well as from UIKit itself.

Note: If you use Core Text or Core Graphics to draw text, remember that you must apply a flip transform to the current graphics context to have text displayed in its proper orientation—that is, with the drawing origin at the upper-left corner of the string's bounding box. In addition, text drawing in Core Text and Core Graphics requires a graphics context set with the text matrix.

Core Text

Core Text is a technology for sophisticated text layout and font management. It is intended to be used by applications with a heavy reliance on text processing—for example, book readers and word processors. It is implemented as a framework that publishes a procedural (ANSI C) API. This API is consistent with that of Core Foundation and Core Graphics, and is integrated with these other frameworks. For example, Core Text uses Core Foundation and Core Graphics objects in many input and output parameters. Moreover, because many Core Foundation objects are "toll-free bridged" with their counterparts in the Foundation framework, you may use some Foundation objects in the parameters of Core Text functions.

You should not use Core Text unless you want to do custom text layout.

Note: Although Core Text is new in iOS 3.2, the framework has been available in Mac OS X since Mac OS X v10.5. For a detailed description of Core Text and some examples of its usage (albeit in the context of Mac OS X), see *Core Text Programming Guide*.

Core Text has two major parts: a layout engine and font technology, each backed by its own collection of opaque types.

Core Text Layout Opaque Types

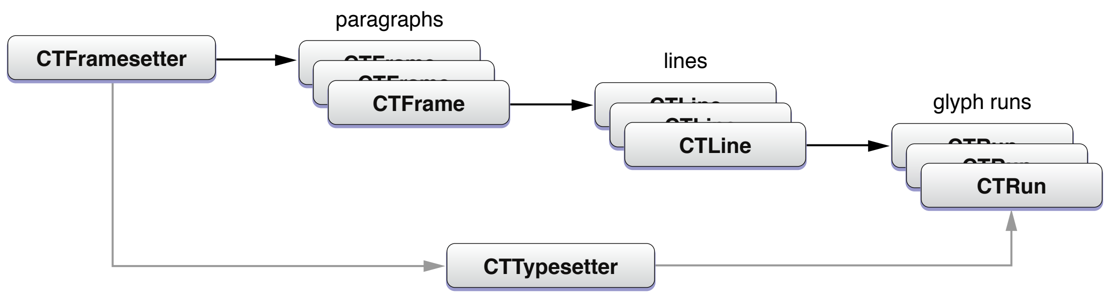
Core Text requires two objects whose opaque types are not native to it: an **attributed string** (`CFAttributedStringRef`) and a **graphics path** (`CGPathRef`). An attributed-string object encapsulates a string backing the displayed text and includes properties (or, "attributes") that define stylistic aspects of the characters in the string—for example, font and color. The graphics path defines the shape of a frame of text, which is equivalent to a paragraph.

Core Text objects at runtime form a hierarchy that is reflective of the level of the text being processed (see). At the top of this hierarchy is the **framesetter** object (`CTFramesetterRef`). With an attributed string and a graphics path as input, a framesetter generates one or more **frames** of text (`CTFrameRef`). As the text is laid out in a frame, the framesetter applies paragraph styles to it, including such attributes as alignment, tab stops, line spacing , indentation, and line-breaking mode.

To generate frames, the framesetter calls a **typesetter** object (`CTTypesetterRef`). The typesetter converts the characters in the attributed string to glyphs and fits those glyphs into the lines that fill a text frame. (A glyph is a graphic shape used to represent a character.) A line in a frame is represented by a **CFLine** object (`CTLineRef`). A CTFrame object contains an array of CTLine objects.

A CTLine object, in turn, contains an array of **glyph runs**, represented by objects of the `CTRunRef` type. A glyph run is a series of consecutive glyphs that have the same attributes and direction.

Figure 7-2 Architecture of the Core Text layout engine



Using functions of the CTLine opaque type, you can draw a line of text from an attributed string without having to go through the CTFramesetter object. You simply position the origin of the text on the text baseline and request the line object to draw itself.

Core Text Font Opaque Types

Fonts are essential to text processing in Core Text. The typesetter object uses fonts (along with the source attributed string) to convert glyphs from characters and then position those glyphs relative to one another. A graphics context establishes the current font for all text drawing that occurs in that context. The Core Text font system handles Unicode fonts natively.

The font system includes objects of three opaque types: CTFont, CTFontDescriptor, and CTFontCollection.

- Font objects (`CTFontRef`) are initialized with a point size and specific characteristics (from a transformation matrix). You can query the font object for its character-to-glyph mapping, its encoding, glyph data, and metrics such as ascent, leading, and so on. Core Text also offers an automatic font-substitution mechanism called font cascading.
- Font descriptor objects (`CTFontDescriptorRef`) are typically used to create font objects. Instead of dealing with a complex transformation matrix, they allow you to specify a dictionary of font attributes that include such properties as PostScript name, font family and style, and traits (for example, bold or italic).
- Font collection objects (`CTFontCollectionRef`) are groups of font descriptors that provide services such as font enumeration and access to global and custom font collections.

Core Text and the UIKit Framework

Core Text and the text layout and rendering facilities of the UIKit framework are not compatible. This incompatibility has the following implications:

- You cannot use Core Text to compute the layout of text and then use APIs such as `UIStringDrawing` to draw the text.
- If your application uses Core Text, it does not have access to text-related UIKit features such as copy-paste. If you use Core Text and want these features, you must implement them yourself.

By default, UIKit does not do kerning, which can cause lines to be dropped.

UIStringDrawing and CATextLayer

`UIStringDrawing` and `CATextLayer` are programmatic facilities that are ideal for simple text drawing. `UIStringDrawing` is a category on `NSString` implemented by the UIKit framework. `CATextLayer` is part of the Core Animation technology.

The methods of `UIStringDrawing` enable iOS applications to draw strings at a given point (for single lines of text) or within a specified rectangle (for multiple lines). You can pass in attributes used in drawing—for example, font, line-break mode, and baseline adjustment. Some methods, given certain parameters such as font, line-breaking mode, and width constraints, return the size of a drawn string and thus let you compute the bounding rectangle for that string when you draw it.

The `CATextLayer` class of Core Animation stores a plain string or attributed string as its content and offers a set of attributes that affect that content, such as font, font size, text color, and truncation behavior. The advantage of `CATextLayer` is that (being a subclass of `CALayer`) its properties are inherently capable of animation. Core Animation is associated with the QuartzCore framework.

Because instances of `CATextLayer` know how to draw themselves in the current graphics context, you don't need to issue any explicit drawing commands when using those instances.

To learn more about `UIStringDrawing`, read *NSString UIKit Additions Reference*. To learn more about `CATextLayer`, `CALayer`, and the other classes of Core Animation, read *Core Animation Programming Guide*.

Core Graphics Text Drawing

Core Graphics (or Quartz) is the system framework that handles two-dimensional imaging at the lowest level. Text drawing is one of its capabilities. Generally, because Core Graphics is so low-level, it is recommended that you use Core Text or one of the system's other facilities for drawing text. However, drawing text with Core Graphics does bring some advantages. It gives you more control of the fonts you use when drawing and allows more precise rendering and placement of glyphs.

You select fonts, set text attributes, and draw text using functions of the CGContext opaque type. For example, you can call `CGContextSelectFont` to set the font used, and then call `CGContextSetFillColor` to set the text color. You then set the text matrix (`CGContextSetTextMatrix`) and draw the text using `CGContextShowGlyphsAtPoint`.

To learn more about the text-drawing API of Core Graphics, read "Text" in *Quartz 2D Programming Guide*.

Foundation-Level Regular Expressions

The `NSString` class of the Foundation framework includes a simple programmatic interface for regular expressions. You call one of three methods that return a range, passing in a specific option constant and a regular-expression string. If there is a match, the method returns the range of the substring. The option is the `NSRegularExpressionSearch` constant, which is of bit-mask type `NSStringCompareOptions`; this constant tells the method to expect a regular-expression pattern rather than a literal string as the search value. The supported regular expression syntax is that defined by ICU (International Components for Unicode).

Note: The `NSString` regular-expression feature was introduced in iOS 3.2. The ICU User Guide describes how to construct ICU regular expressions (http://userguide.icu-project.org/strings/regexp).

The `NSString` methods for regular expressions are the following:

```
rangeOfString:options:
rangeOfString:options:range:
rangeOfString:options:range:locale:
```

If you specify the `NSRegularExpressionSearch` option in these methods, the only other `NSStringCompareOptions` options you may specify are `NSCaseInsensitiveSearch` and `NSAnchoredSearch`. If a regular-expression search does not find a match or the regular-expression syntax is malformed, these methods return an `NSRange` structure with a value of `{NSNotFound, 0}`.

Listing 7-3 gives an example of using the `NSString` regular-expression API.

Listing 7-3 Finding a substring using a regular expression

```
// finds phone number in format nnn-nnn-nnnn
NSRange r;
```

```
NSString *regEx = @"[0-9]{3}-[0-9]{3}-[0-9]{4}";
r = [textView.text rangeOfString:regEx options:NSRegularExpressionSearch];
if (r.location != NSNotFound) {
    NSLog(@"Phone number is %@", [textView.text substringWithRange:r]);
} else {
    NSLog(@"Not found.");
}
```

Because these methods return a single range value for the substring matching the pattern, certain regular-expression capabilities of the ICU library are either not available or have to be programmatically added. In addition, `NSStringCompareOptions` options such backward search, numeric search, and diacritic-insensitive search are not available and capture groups are not supported.

Note: As noted in "ICU Regular-Expression Support" (page 84), the ICU libraries related to regular expressions are included in iOS 3.2. However, you should only use the ICU facilities if the `NSString` alternative is not sufficient for your needs.

When testing the returned range, you should be aware of certain behavioral differences between searches based on literal string and searches based on regular-expression patterns. Some patterns can successfully match and return an `NSRange` structure with a `length` of 0 (in which case the `location` field is of interest). Other patterns can successfully match against an empty string or, in those methods with a range parameter, with a zero-length search range.

ICU Regular-Expression Support

A modified version of the libraries from 4.2.1 is included in iOS 3.2 at the BSD (non-framework) level of the system. ICU (International Components for Unicode) is an open-source project for Unicode support and software internationalization. The installed version of ICU includes those header files necessary to support regular expressions along with some modifications related to those interfaces. Table 7-1 lists these files.

Table 7-1 ICU files included in iOS 3.2

`parseerr.h`	`platform.h`	`putil.h`
`uconfig.h`	`udraft.h`	`uintrnal.h`
`uiter.h`	`umachine.h`	`uregex.h`
`urename.h`	`ustring.h`	`utf_old.h`
`utf.h`	`utf16.h`	`utf8.h`
`utypes.h`	`uversion.h`	

You can read the ICU 4.2 API documentation and user guide at http://icu-project.org/apiref/icu4c/index.html.

Spell Checking and Word Completion

With an instance of the `UITextChecker` class you can check the spelling of a document or offer suggestions for completing partially entered words. When spell-checking a document, a `UITextChecker` object searches a document at a specified offset. When it detects a misspelled word, it can also return an array of possible correct spellings, ranked in the order which they should be presented to the user (that is, the most likely replacement word comes first). You typically use a single instance of `UITextChecker` per document, although you can use a single instance to spell-check related pieces of text if you want to share ignored words and other state.

Note: The `UITextChecker` class is intended for spell-checking and *not* for autocorrection. Autocorrection is a feature your text document can acquire by adopting the protocols and implementing the subclasses described in "Communicating with the Text Input System" (page 77).

The method you use for checking a document for misspelled words is `rangeOfMisspelledWordInString:range:startingAt:wrap:language:`; the method used for obtaining the list of possible replacement words is `guessesForWordRange:inString:language:`. You call these methods in the given order. To check an entire document, you call the two methods in a loop, resetting the starting offset to the character following the corrected word at each cycle through the loop, as shown in Listing 7-4.

Listing 7-4 Spell-checking a document

```
- (IBAction)spellCheckDocument:(id)sender {
    NSInteger currentOffset = 0;
    NSRange currentRange = NSMakeRange(0, 0);
    NSString *theText = textView.text;
    NSRange stringRange = NSMakeRange(0, theText.length-1);
    NSArray *guesses;
    BOOL done = NO;

   NSString *theLanguage = [[UITextChecker availableLanguages] objectAtIndex:0];
    if (!theLanguage)
        theLanguage = @"en_US";

    while (!done) {
        currentRange = [textChecker rangeOfMisspelledWordInString:theText
range:stringRange
            startingAt:currentOffset wrap:NO language:theLanguage];
        if (currentRange.location == NSNotFound) {
            done = YES;
            continue;
        }
       guesses = [textChecker guessesForWordRange:currentRange inString:theText
            language:theLanguage];
        NSLog(@"------------------------------------------");
        NSLog(@"Word misspelled is %@", [theText
substringWithRange:currentRange]);
        NSLog(@"Possible replacements are %@", guesses);
        NSLog(@" ");
        currentOffset = currentOffset + (currentRange.length-1);
    }
}
```

The `UITextChecker` class includes methods for telling the text checker to ignore or learn words. Instead of just logging the misspelled words and their possible replacements, as the method in Listing 7-4 does, you should display some user interface that allows users to select correct spellings, tell the text checker to ignore or learn a word, and proceed to the next word without making any changes. One possible approach would be to use a popover view that lists the guesses in a table view and includes buttons such as Replace, Learn, Ignore, and so on.

You may also use `UITextChecker` to obtain completions for partially entered words and display the completions in a table view in a popover view. For this task, you call the `completionsForPartialWordRange:inString:language:` method, passing in the range in the given string to check. This method returns an array of possible words that complete the partially entered word. Listing 7-5 shows how you might call this method and display a table view listing the completions in a popover view.

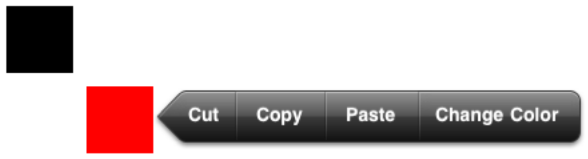
Listing 7-5 Presenting a list of word completions for the current partial string

```
- (IBAction)completeCurrentWord:(id)sender {

    self.completionRange = [self computeCompletionRange];
    // The UITextChecker object is cached in an instance variable
    NSArray *possibleCompletions = [textChecker
completionsForPartialWordRange:self.completionRange
        inString:self.textStore language:@"en"];

    CGSize popOverSize = CGSizeMake(150.0, 400.0);
    completionList = [[CompletionListController alloc]
initWithStyle:UITableViewStylePlain];
    completionList.resultsList = possibleCompletions;
    completionListPopover = [[UIPopoverController alloc]
initWithContentViewController:completionList];
    completionListPopover.popoverContentSize = popOverSize;
    completionListPopover.delegate = self;
    // rectForPartialWordRange: is a custom method
    CGRect pRect = [self rectForPartialWordRange:self.completionRange];
    [completionListPopover presentPopoverFromRect:pRect inView:self
        permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
```

Custom Edit Menu Items

You can add a custom item to the edit menu used for showing the system commands Copy, Cut, Paste, Select, Select All, and Delete. When users tap this item, a command is issued that affects the current target in an application-specific way. The UIKit framework accomplishes this through the target-action mechanism. The tap of an item results in a action message being sent to the first object in the responder chain that can handle the message. Figure 7-3 shows an example of a custom menu item ("Change Color").

Figure 7-3 An editing menu with a custom menu item



An instance of the `UIMenuItem` class represents a custom menu item. `UIMenuItem` objects have two properties, a title and an action selector, which you can change at any time. To implement a custom menu item, you must initialize a `UIMenuItem` instance with these properties, add the instance to the menu controller's array of custom menu items, and then implement the action method for handling the command in the appropriate responder subclass.

Other aspects of implementing a custom menu item are common to all code that uses the singleton `UIMenuController` object. In a custom or overridden view, you set the view to be the first responder, get the shared menu controller, set a target rectangle, and then display the editing menu with a call to `setMenuVisible:animated:`. The simple example in Listing 7-6 adds a custom menu item for changing a custom view's color between red and black.

Listing 7-6 Implementing a Change Color menu item

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    if ([theTouch tapCount] == 2) {
        [self becomeFirstResponder];
       UIMenuItem *menuItem = [[UIMenuItem alloc] initWithTitle:@"Change Color"
 action:@selector(changeColor:)];
        UIMenuController *menuCont = [UIMenuController sharedMenuController];
        [menuCont setTargetRect:self.frame inView:self.superview];
        menuCont.arrowDirection = UIMenuControllerArrowLeft;
        menuCont.menuItems = [NSArray arrayWithObject:menuItem];
        [menuCont setMenuVisible:YES animated:YES];
    }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {}

- (BOOL)canBecomeFirstResponder { return YES; }

- (void)changeColor:(id)sender {
    if ([self.viewColor isEqual:[UIColor blackColor]]) {
        self.viewColor = [UIColor redColor];
    } else {
        self.viewColor = [UIColor blackColor];
    }
    [self setNeedsDisplay];
}
```

Note: The `arrowDirection` property of `UIMenuController`, shown in Listing 7-6, is new in iOS 3.2. It allows you to specify the direction the arrow attached to the editing menu points at its target rectangle. Also new is the Delete menu command; if users tap this menu command, the `delete:` method implemented by an object in the responder chain (if any) is invoked. The `delete:` method is declared in the `UIResponderStandardEditActions` informal protocol.

Document Revision History

This table describes the changes to *iPad Programming Guide*.

Date	Notes
2010-04-13	Made minor updates to clarify the policy on application icons.
2010-03-24	New document describing how to write applications for iPad.