Text and Web Programming Guide for iOS

Data Management: Strings, Text, & Fonts

2010-07-07

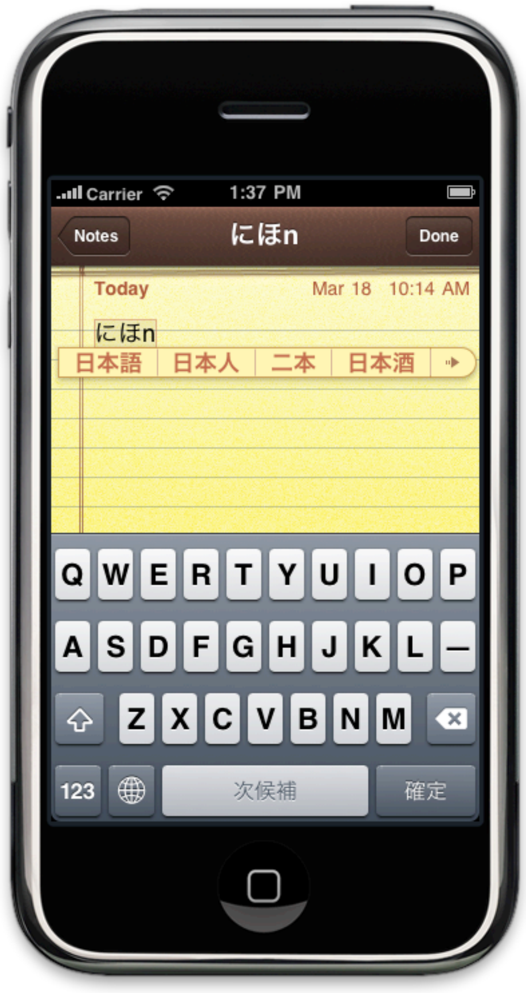
Contents

Figures and Listings

About Text and Web Support in iOS

iOS gives you many ways to display text in your applications and let users edit that text. It also lets you display web content in your application's views. The resources at your disposal range from framework objects such as text fields and web views to lower-level technologies that allow your application to draw, lay out, and otherwise manage text.

Note: This document contains information that used to be in *iPhone Application Programming Guide* and *iPad Programming Guide*.



At a Glance

The UIKit Framework Provides Your Application with Text and Web Objects

Instances of the `UITextView`, `UITextField`, and `UILabel` classes serve as ready-made text views, text fields, and labels that you can add to your application's user interface. You can add and configure them programmatically or by using the Interface Builder application. A `UIWebView` object can turn a view of your application into a miniature web browser capable of understanding and displaying HTML, CSS, and JavaScript content

Relevant Chapters: "Text Objects and Web Views" (page 9), "Displaying Web Content" (page 33)

When Users Edit Text, Your Application Must Manage the Keyboard

When a user taps a text field, text view, or form field in a web view, iOS animates a keyboard into view. An application can control which keyboard is presented; for example, for a numeric-value field, the application should select the number-pad keyboard. If the entered or edited text is obscured by the keyboard, the application should adjust the view displaying the text so that the text appears above the keyboard. The delegate of a text view, text field, or web view is responsible for validating edited text and for accessing and storing edited text when the user dismisses the keyboard.

Relevant Chapters: "Managing the Keyboard" (page 13)

Your Application Can Draw and Manage Text on Its Own

Instead of using the UIKit classes for displaying and and editing text, you could choose to implement an application that does simple text drawing or text input, or even one with its own text layout and management engine. For assistance in text layout and font management, use the Core Text framework. To communicate with the text-input system of iOS, implement the `UITextInput` protocol and related protocols and classes. Your application can also make use of technologies for spell-checking and regular expressions.

Relevant Chapter: "Drawing and Managing Text" (page 35)

See Also

If your application is going to do sophisticated text entry and management, it should use the Core Text technology; read *Core Text Programming Guide* to learn about Core Text.

The Core Graphics and Core Animation frameworks also have text capabilities. Core Animation, for example, offers the `CATextLayer` class. To learn more about these capabilities, read *Quartz 2D Programming Guide* (Core Graphics) and *Core Animation Programming Guide*.

Text Objects and Web Views

The text system in iOS provides a tremendous amount of power while still being very simple to use. The UIKit framework includes several high-level classes for managing the display and input of text. This framework also includes a more advanced class for displaying HTML and JavaScript-based content.

The following sections describe the basic support for text and web content in iOS.

Note: This chapter contains information that used to be in *iPhone Application Programming Guide*. The chapter in this document has not been updated specifically for iOS 4.0.

Text Views

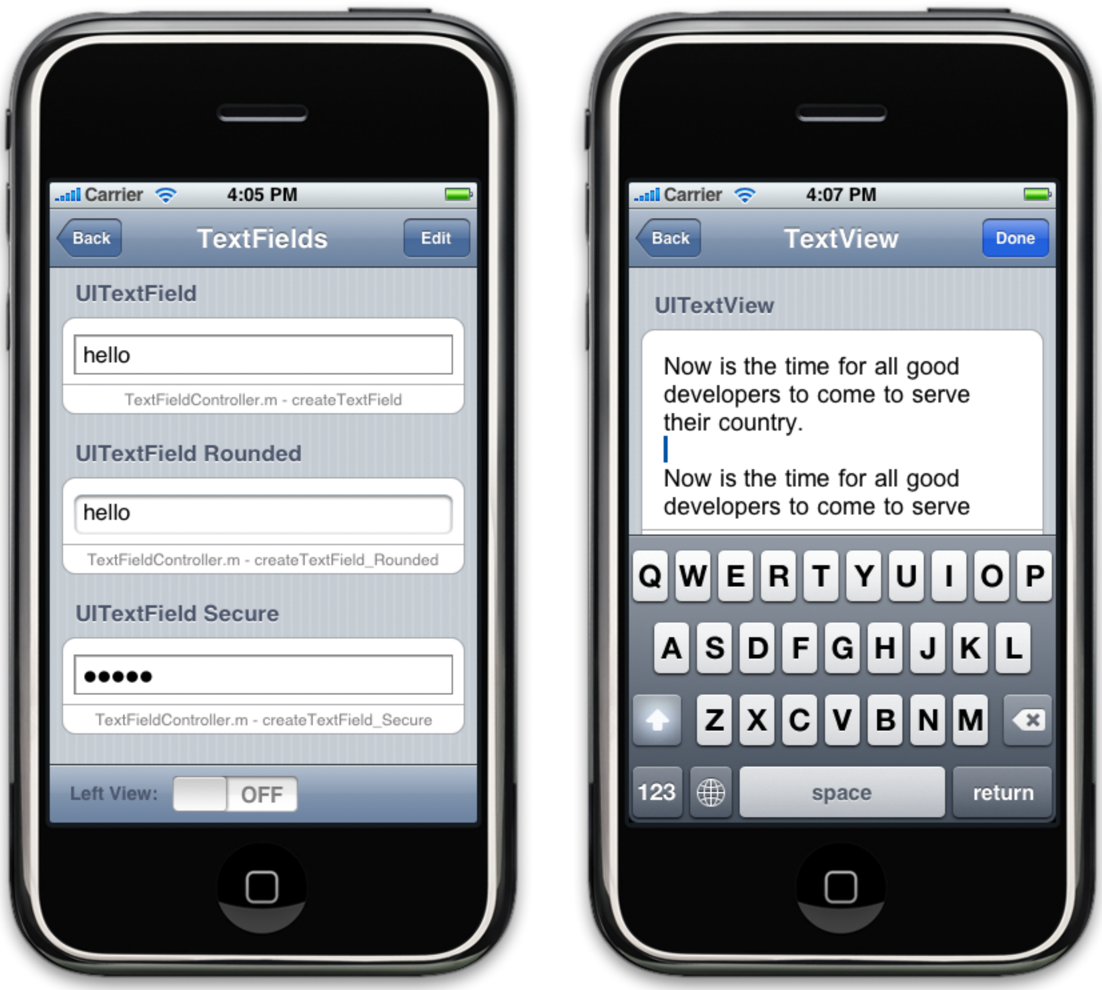
The UIKit framework provides three primary classes for displaying text content:

- `UILabel` displays static text strings
- `UITextField` displays a single line of editable text
- `UITextView` displays multiple lines of editable text

These classes support the display of arbitrarily large amounts of text, although labels and text fields are typically used for relatively small amounts of text. To make the displayed text easier to read on the smaller screens of iOS–based devices, however, these classes do not support the kinds of advanced formatting you might find in desktop operating systems like Mac OS X. All three classes still allow you to specify the font information, including size and styling options, that you might otherwise want, but the font information you specify is applied to all of the text associated with the object.

Figure 1-1 shows examples of the available text classes as they appear on screen. These examples were taken from the *UICatalog* sample application, which demonstrates many of the views and controls available in UIKit. The image on the left shows several different styles of text fields while the image on the right shows a single text view. The callouts displayed on the gray background are themselves `UILabel` objects embedded inside the table cells used to display the different views. There is also a `UILabel` object with the text “Left View” at the bottom of the screen on the left.

Figure 1-1 Text classes in the UICatalog application



hello

hello

●●●●●

Now is the time for all good developers to come to serve their country.

Now is the time for all good developers to come to serve

When working with editable text views, you should always provide a delegate object to manage the editing session. Text views send several different notifications to the delegate to let them know when editing begins, when it ends, and to give them a chance to override some editing actions. For example, the delegate can decide if the current text contains a valid value and prevent the editing session from ending if it does not. When editing does finally end, you also use the delegate to get the resulting text value and update your application's data model.

Because there are slight differences in their intended usage, the delegate methods for each text view are slightly different. A delegate that supports the `UITextField` class implements the methods of the `UITextFieldDelegate`protocol. Similarly, a delegate that supports the `UITextView` class implements the methods of the `UITextViewDelegate` protocol. In both cases, you are not required to implement any of the protocol methods but if you do not, the text field is not going to be of much use to you.

"Managing Text Fields and Text Views" (page 23) describes the sequence of delegation messages for both text fields and text views and discusses various tasks performed by the delegates of these objects. For more information about the methods of the `UITextFieldDelegate` and `UITextViewDelegate` protocols, see *UITextFieldDelegate Protocol Reference* and *UITextViewDelegate Protocol Reference*.

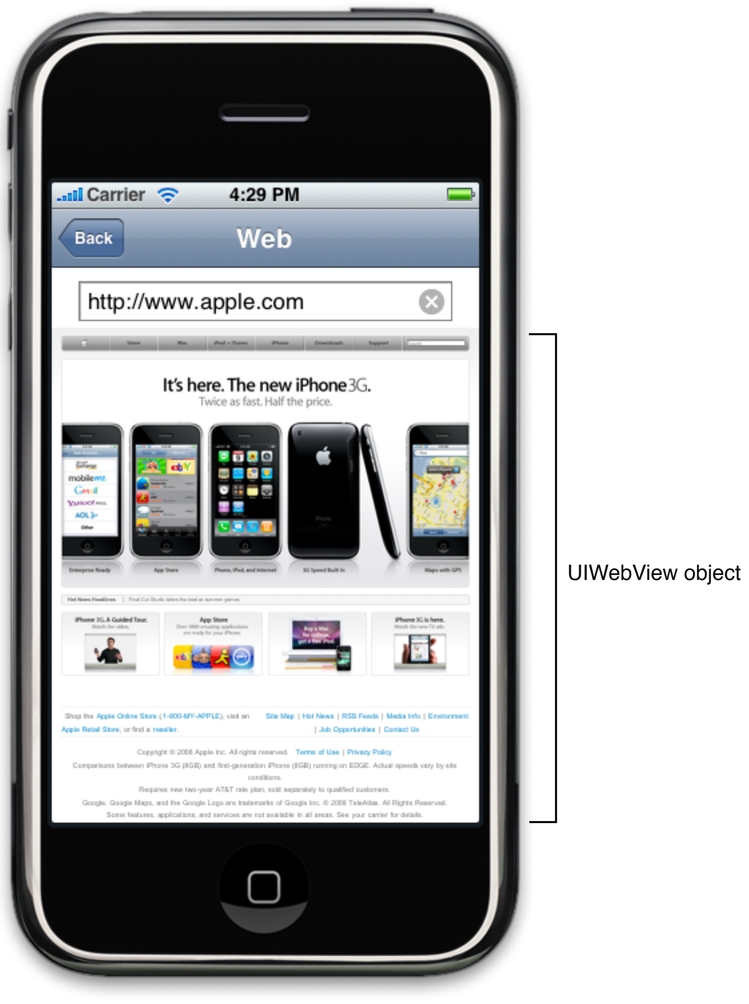
Web View

The `UIWebView` class lets you integrate what is essentially a miniature web browser into your application's user interface. The `UIWebView` class makes full use of the same web technologies used to implement Safari in iOS, including full support for HTML, CSS, and JavaScript content. The class also supports many of the built-in gestures that users are familiar with in Safari. For example, you can double-click and pinch to zoom in and out of the page and you can scroll around the page by dragging your finger.

In addition to displaying content, you can also use a web view object to gather input from the user through the use of web forms. Like the other text classes in UIKit, if you have an editable text field on a form in your web page, tapping that field brings up a keyboard so that the user can enter text. Because it is an integral part of the web experience, the web view itself manages the displaying and dismissing of the keyboard for you.

Figure 1-2 shows an example of a `UIWebView` object from the the *UICatalog* sample application, which demonstrates many of the views and controls available in UIKit. Because it just displays HTML content, if you want the user to be able to navigate pages much like they would in a web browser, you need to add controls to do so. For example, the web view in the figure occupies the space below the text field containing the target URL and does not contain the text field itself.

Figure 1-2 A web view



A web view provides information about when pages are loaded, and whether there were any load errors, through its associated delegate object. A web delegate is an object that implements one or more methods of the `UIWebViewDelegate`protocol. Your implementations of the delegate methods can respond to failures or perform other tasks related to the loading of a web page.

"Displaying Web Content" (page 33) describes how to display HTML and other content in a web view. For more information about the methods of the `UIWebViewDelegate` protocol, see *UIWebViewDelegate Protocol Reference*.

Managing the Keyboard

When users touch a text field, a text view, or a field in a web view, the system displays a keyboard. You can configure the type of keyboard that is displayed along with several attributes of the keyboard. You also have to manage the keyboard when the editing session begins and ends. Because the keyboard could hide the portion of your view that is the focus of editing, this management might include adjusting the user interface to raise the area of focus so that is visible above the keyboard.

Note: This chapter contains information that used to be in *iPhone Application Programming Guide*. The information in this chapter has not been updated specifically for iOS 4.0.

Keyboards and Input Methods

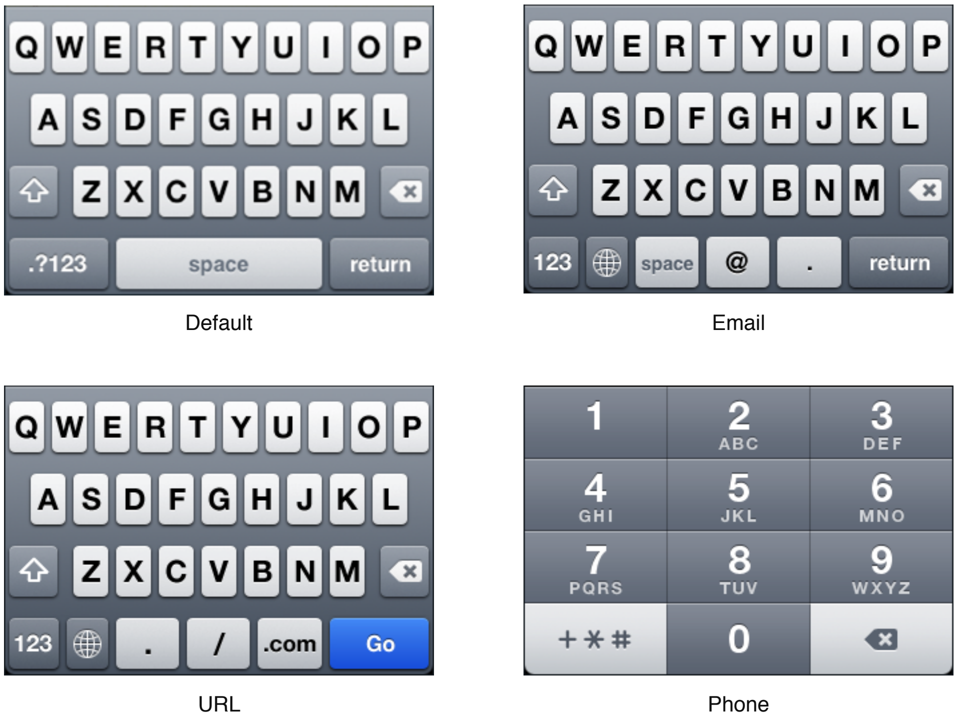
Whenever the user taps in an object capable of accepting text input, the object asks the system to display an appropriate keyboard. Depending on the needs of your program and the user's preferred language, the system might display one of several different keyboards. Although your application cannot control the user's preferred language (and thus the keyboard's input method), it can control attributes of the keyboard that indicate its intended use, such as the configuration of any special keys and its behaviors.

Configuring the Keyboard for Text Objects

You configure the attributes of the keyboard directly through the text objects of your application. The `UITextField` and `UITextView` classes both conform to the `UITextInputTraits`protocol, which defines the properties for configuring the keyboard. Setting these properties programmatically or in the Interface Builder inspector window causes the system to display the keyboard of the designated type.

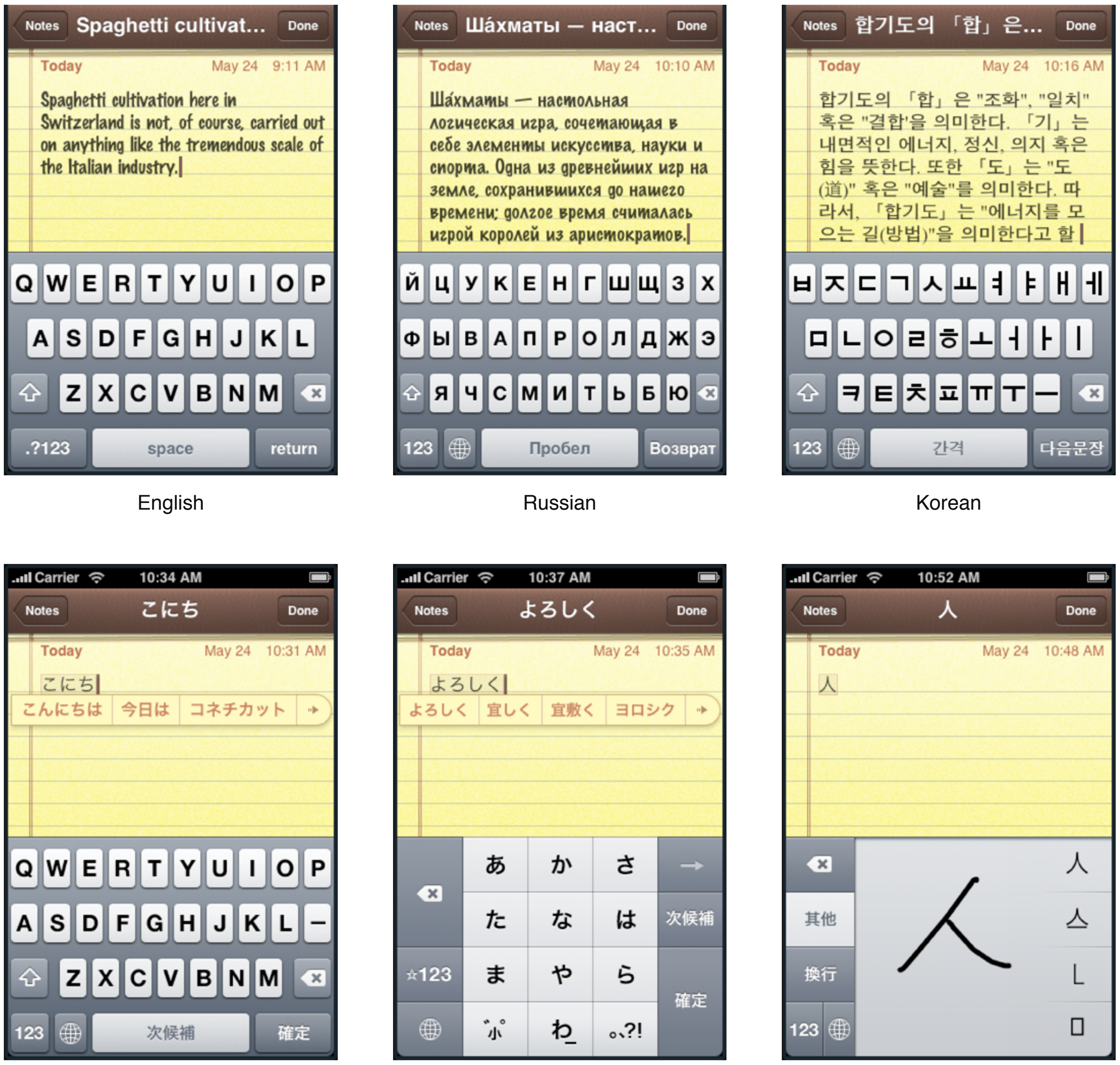
The default keyboard configuration is designed for general text input. Figure 2-1 displays the default keyboard along with several other keyboard configurations. The default keyboard displays an alphabetical keyboard initially but the user can toggle it and display numbers and punctuation as well. Most of the other keyboards offer similar features as the default keyboard but provide additional buttons that are specially suited to particular tasks. However, the phone and numerical keyboards offer a dramatically different layout that is tailored towards numerical input.

Figure 2-1 Several different keyboard types



To facilitate the language preferences of different users, iOS also supports different input methods and keyboard layouts for different languages, some of which are shown in Figure 2-2. The input method and layout for the keyboard is determined by the user's language preferences. Input for some of these keyboards takes place in multiple stages.

Figure 2-2 Several different keyboards and input methods


Spaghetti cultivation here in Switzerland is not, of course, carried out on anything like the tremendous scale of the Italian industry.

Ша́хматы — настольная логическая игра, сочетающая в себе элементы искусства, науки и спорта. Одна из древнейших игр на земле, сохранившихся до нашего времени; долгое время считалась игрой королей из аристократов.

합기도의 「합」은 "조화", "일치" 혹은 "결합"을 의미한다. 「기」는 내면적인 에너지, 정신, 의지 혹은 힘을 뜻한다. 또한 「도」는 "도(道)" 혹은 "예술"를 의미한다. 따라서, 「합기도」는 "에너지를 모으는 길(방법)"을 의미한다고 할

Configuring the Keyboard for Web Views

Although the `UIWebView` class does not support the `UITextInputTraits` protocol directly, you can configure some keyboard attributes for text input elements. For example, you can include `autocorrect` and `auto-capitalization` attributes in the definition of an input element to specify the keyboard's behaviors, as shown in the following example.

```
<input type="text" size="30" autocorrect="off" autocapitalization="on">
```

You can also control which type of keyboard is displayed when a user touches a text field in a web page. To display a telephone keypad, an email keyboard, or a URL keyboard, use the `tel`, `email`, or `url` keywords for the `type` attribute on an input element, respectively. To display a numeric keyboard, set the value of the `pattern` attribute to "`[0-9]*`" or "`\d*`".

These keywords and the pattern attribute are part of HTML 5, and are available in iOS 3.1 and later. The following list shows how to display each type of keyboard, including the standard keyboard.

Text: `<input type="text"></input>`
Telephone: `<input type="tel"></input>`
URL: `<input type="url"></input>`
Email: `<input type="email"></input>`
Zip code: `<input type="text" pattern="[0-9]*"></input>`

Managing the Keyboard

Although many UIKit objects display the keyboard automatically in response to user interactions, your application still has some responsibilities for configuring and managing the keyboard. The following sections describe those responsibilities.

Receiving Keyboard Notifications

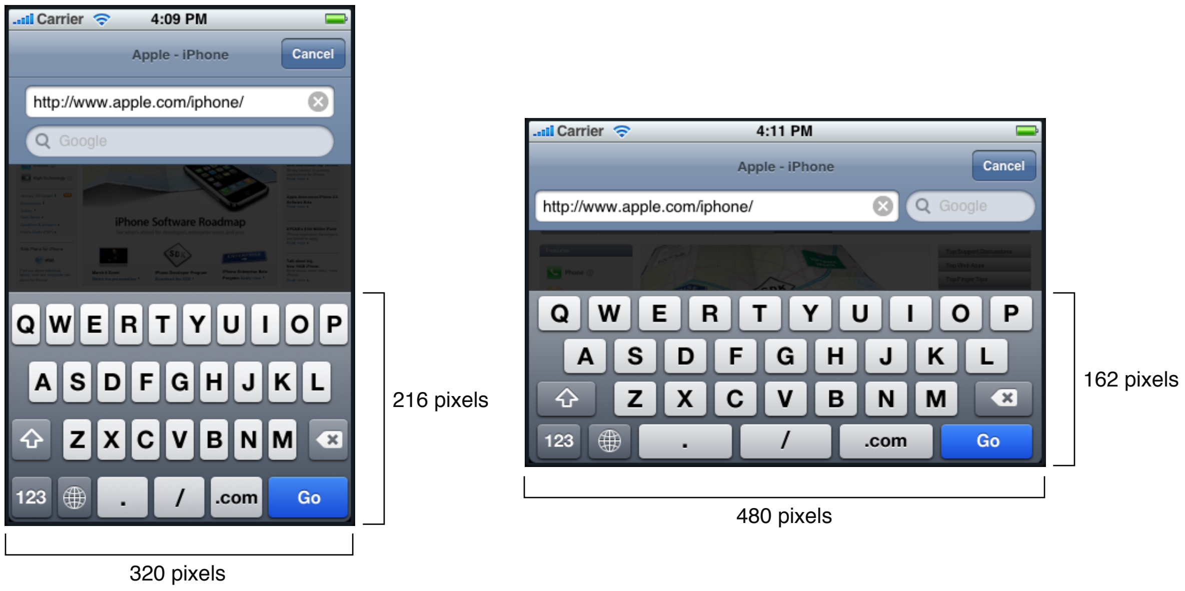
When the keyboard is shown or hidden, iOS sends out the following notifications to any registered observers:

- `UIKeyboardWillShowNotification`
- `UIKeyboardDidShowNotification`
- `UIKeyboardWillHideNotification`
- `UIKeyboardDidHideNotification`

The system sends keyboard notifications when the keyboard first appears, when it disappears, any time the owner of the keyboard changes, or any time your application's orientation changes. In each situation, the system sends only the appropriate subset of messages. For example, if the owner of the keyboard changes, the system sends a `UIKeyboardWillHideNotification` message, but not a `UIKeyboardDidHideNotification` message, to the current owner because the change never causes the keyboard to be hidden. The delivery of the `UIKeyboardWillHideNotification` is simply a way to alert the current owner that it is about to lose the keyboard focus. Changing the orientation of the keyboard does send both will and did hide notifications, however, because the keyboards for each orientation are different and thus the original must be hidden before the new one is displayed.

Each keyboard notification includes information about the size and position of the keyboard on the screen. You can access this information from the `userInfo` dictionary of each notification using the `UIKeyboardFrameBeginUserInfoKey`and `UIKeyboardFrameEndUserInfoKey` keys; the former gives the beginning keyboard frame, the latter the ending keyboard frame (both in screen coordinates). You should always use the information in these notifications as opposed to assuming the keyboard is a particular size or in a particular location. The size of the keyboard is not guaranteed to be the same from one input method to another and may also change between different releases of iOS. In addition, even for a single language and system release, the keyboard dimensions can vary depending on the orientation of your application. For example, Figure 2-3 shows the relative sizes of the URL keyboard in both the portrait and landscape modes. Using the information inside the keyboard notifications ensures that you always have the correct size and position information.

Figure 2-3 Relative keyboard sizes in portrait and landscape modes



> **Note:** The rectangle contained in the `UIKeyboardFrameBeginUserInfoKey` and `UIKeyboardFrameEndUserInfoKey` properties of the `userInfo` dictionary should be used only for the size information it contains. Do not use the origin of the rectangle (which is always {0.0, 0.0}) in rectangle-intersection operations. Because the keyboard is animated into position, the actual bounding rectangle of the keyboard changes over time.

One reason to use keyboard notifications is so that you can reposition content that is obscured by the keyboard when it is visible. For information on how to handle this scenario, see "Moving Content That Is Located Under the Keyboard" (page 18).

Displaying the Keyboard

When the user taps a view, the system automatically designates that view as the first responder. When this happens to a view that contains editable text, the view initiates an editing session for that text. At the beginning of that editing session, the view asks the system to display the keyboard, if it is not already visible. If the keyboard is already visible, the change in first responder causes text input from the keyboard to be redirected to the newly tapped view.

Because the keyboard is displayed automatically when a view becomes the first responder, you often do not need to do anything to display it. However, you can programmatically display the keyboard for an editable text view by calling that view's `becomeFirstResponder` method. Calling this method makes the target view the first responder and begins the editing process just as if the user had tapped on the view.

If your application manages several text-based views on a single screen, it is a good idea to track which view is currently the first responder so that you can dismiss the keyboard later.

Dismissing the Keyboard

Although it typically displays the keyboard automatically, the system does not dismiss the keyboard automatically. Instead, it is your application's responsibility to dismiss the keyboard at the appropriate time. Typically, you would do this in response to a user action. For example, you might dismiss the keyboard when the user taps the Return or Done button on the keyboard or taps some other button in your application's interface. Depending on how you configured the keyboard, you might need to add some additional controls to your user interface to facilitate the keyboard's dismissal.

To dismiss the keyboard, you call the `resignFirstResponder` method of the text-based view that is currently the first responder. When a text view resigns its first responder status, it ends its current editing session, notifies its delegate of that fact, and dismisses the keyboard. In other words, if you have a variable called `myTextField` that points to the `UITextField` object that is currently the first responder, dismissing the keyboard is as simple as doing the following:

```
[myTextField resignFirstResponder];
```

Everything from that point on is handled for you automatically by the text object.

Moving Content That Is Located Under the Keyboard

When asked to display the keyboard, the system slides it in from the bottom of the screen and positions it over your application's content. Because it is placed on top of your content, it is possible for the keyboard to be placed on top of the text object that the user wanted to edit. When this happens, you must adjust your content so that the target object remains visible.

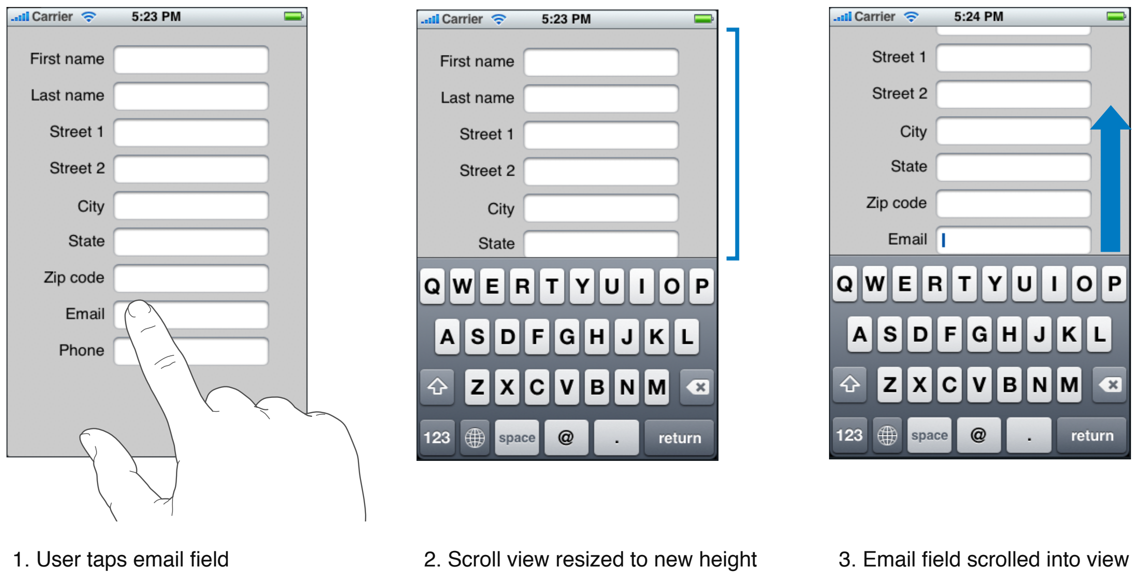
Adjusting your content typically involves temporarily resizing one or more views and positioning them so that the text object remains visible. The simplest way to manage text objects with the keyboard is to embed them inside a `UIScrollView` object (or one of its subclasses like `UITableView`). When the keyboard is displayed, all you have to do is resize the scroll view and scroll the desired text object into position. Thus, in response to a `UIKeyboardDidShowNotification`, your handler method would do the following:

1. Get the size of the keyboard.
2. Subtract the keyboard height from the height of your scroll view.
3. Scroll the target text field into view.

Note: As of iOS 3.0, the `UITableViewController` class automatically resizes and repositions its table view when there is in-line editing of text fields. See "View Controllers and Navigation-Based Applications" in *Table View Programming Guide for iOS*.

Figure 2-4 illustrates the preceding steps for a simple application that embeds several text fields inside a `UIScrollView` object. When the keyboard appears, the notification handler method resizes the scroll view and then uses the `scrollRectToVisible:animated:` method of `UIScrollView` to scroll the tapped text field (in this case the email field) into view.

Figure 2-4 Adjusting content to accommodate the keyboard



Note: When setting up your own scroll views, be sure to configure the autoresizing rules for any content views appropriately. In the preceding figure, the text fields are actually subviews of a generic `UIView` object, which is itself a subview of the `UIScrollView` object. If the generic view's `UIViewAutoresizingFlexibleWidth` and `UIViewAutoresizingFlexibleHeight` autoresizing options are set, changing the scroll view's frame size also changes the frame of the generic view, which could yield undesirable results. Disabling these options for that view ensures that the view retains its size and its contents are scrolled correctly.

Listing 2-1 shows the code for registering to receive keyboard notifications and shows the handler methods for those notifications. This code is implemented by the view controller that manages the scroll view, and the `scrollView` variable is an outlet that points to the scroll view object. Each handler method gets the keyboard size from the info dictionary of the notification and adjusts the scroll view height by the corresponding amount. In addition, the `keyboardWasShown:` method scrolls the rectangle of the active text field, which is stored in a custom variable (called `activeField` in this example) that is a member variable of the view controller and set in the `textFieldDidBeginEditing:` delegate method, which is itself shown in Listing 2-2 (page 21). (In this example, the view controller also acts as the delegate for each of the text fields.)

Listing 2-1 Handling the keyboard notifications

```
// Call this method somewhere in your view controller setup code.
- (void)registerForKeyboardNotifications
{
    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(keyboardWasShown:)
            name:UIKeyboardDidShowNotification object:nil];

    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(keyboardWasHidden:)
            name:UIKeyboardDidHideNotification object:nil];
}
```

```
// Called when the UIKeyboardDidShowNotification is sent.
- (void)keyboardWasShown:(NSNotification*)aNotification
{
    if (keyboardShown)
        return;

    NSDictionary* info = [aNotification userInfo];

    // Get the size of the keyboard.
    NSValue* aValue = [info objectForKey:UIKeyboardFrameBeginUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    // Resize the scroll view (which is the root view of the window)
    CGRect viewFrame = [scrollView frame];
    viewFrame.size.height -= keyboardSize.height;
    scrollView.frame = viewFrame;

    // Scroll the active text field into view.
    CGRect textFieldRect = [activeField frame];
    [scrollView scrollRectToVisible:textFieldRect animated:YES];

    keyboardShown = YES;
}


// Called when the UIKeyboardDidHideNotification is sent
- (void)keyboardWasHidden:(NSNotification*)aNotification
{
    NSDictionary* info = [aNotification userInfo];

    // Get the size of the keyboard.
    NSValue* aValue = [info objectForKey:UIKeyboardFrameEndUserInfoKey];
    CGSize keyboardSize = [aValue CGRectValue].size;

    // Reset the height of the scroll view to its original value
    CGRect viewFrame = [scrollView frame];
    viewFrame.size.height += keyboardSize.height;
    scrollView.frame = viewFrame;

    keyboardShown = NO;
}
```

The `keyboardShown` variable from the preceding listing is a Boolean value used to track whether the keyboard is already visible. If your interface has multiple text fields, the user can tap between them to edit the values in each one. When that happens, however, the keyboard does not disappear but the system does still generate `UIKeyboardDidShowNotification` notifications each time editing begins in a new text field. By tracking whether the keyboard was actually hidden, this code prevents the scroll view from being reduced in size more than once.

Listing 2-2 shows some additional code used by the view controller to set and clear the `activeField` variable in the preceding example. During initialization, each text field in the interface sets the view controller as its delegate. Therefore, when a text field becomes active, it calls these methods. For more information on text fields and their delegate notifications, see "Managing Text Fields and Text Views" (page 23).

Listing 2-2 Additional methods for tracking the active text field.

```
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    activeField = textField;
}

- (void)textFieldDidEndEditing:(UITextField *)textField
{
    activeField = nil;
}
```

There are other ways you can scroll the edited area in a scroll view above an obscuring keyboard. Instead of altering the frame of the scroll view, you can extend the height of the content view by the height of the keyboard and then scroll the edited area into view. The `UIScrollView` class has a `contentSize` property that you can set for this purpose; you can also adjust the frame of the content view, as shown in Listing 2-3. This code also uses the `setContentOffset:animated:` method to scroll the edited field into view, in this case scrolling it just below the top of the scroll view instead of just above the keyboard.

Listing 2-3 Adjusting the frame of the content view and scrolling a field above the keyboard

```
- (void)keyboardWasShown:(NSNotification*)aNotification
{
    if (keyboardShown)
        return;

    NSDictionary* info = [aNotification userInfo];
    CGSize kbSize = [[info objectForKey:UIKeyboardFrameBeginUserInfoKey]
CGRectValue].size;
    CGRect bkgndRect = activeField.superview.frame;
    bkgndRect.size.height += kbSize.height;
    [activeField.superview setFrame:bkgndRect];
    [scrollView setContentOffset:CGPointMake(0.0, activeField.frame.origin.y)
animated:YES];

    keyboardShown = YES;
}
```

Managing Text Fields and Text Views

Text fields and text views have two main functions: to display text and to enable the entry and editing of text. Several programming tasks are associated with these simple purposes, including configuring the text object, accessing the current text, validating what the user enters, and displaying overlay views such as bookmark buttons in text fields.

For a general discussion of text fields, text views, and labels, including their intended uses, see "Text Objects and Web Views" (page 9).

The delegate of a `UITextField` or `UITextView` object is responsible for most of these tasks. The delegate must adopt the `UITextFieldDelegate` or `UITextViewDelegate` protocols and implement one or more of the protocol methods. Implementation of all protocol methods is optional. To have these methods called, you must set the `delegate` properties of text fields and text views either programmatically or in Interface Builder.

The Sequence of Messages to the Delegate

In most cases, instances of the `UITextField` or `UITextView` classes send a sequence of similarly named messages to their delegates when there is a change (or impending change) in first-responder status for a given text object. Recall from the discussion of the system keyboard in "Managing the Keyboard" (page 13) that, when the user taps a text object, it automatically becomes first responder; as a result, the system displays the keyboard and an editing session begins for that text object. When the user taps another text object or taps a button to end editing, the current text object resigns first-responder status. If no other text object is selected, the system hides the keyboard; if, on the other hand, the user selects another text object, it becomes first responder and the keyboard for that object is displayed.

There are a couple of exceptions to this common behavior. On the iPad, if a view controller modally presents its view using the "form sheet" style, the keyboard, once shown, is not hidden until the user taps the dismiss key or the modal view controller is programmatically dismissed. The purpose of this behavior is to avoid excessive animations as a user moves between views that are largely, but not entirely, text fields. Another exception involves custom input views. An input view is a substitute for system keyboards that is assigned to the `inputView` property of a text view or a custom view. When there are input views, UIKit might swap out the keyboard even when a text object is first responder, and it might show an keyboard-like input view on the developer's behalf for non-text objects.

The sequence of messages that both text views and text fields send to their delegates is as follows:

1. **Just before a text object becomes first responder**—`textFieldShouldBeginEditing:` (text field) and `textViewShouldBeginEditing:` (text view).

   The delegate can verify whether the text object should become first responder by returning `YES` (the default) or `NO`.

2. **Just after a text object becomes first responder**—`textFieldDidBeginEditing:` (text field) and `textViewDidEndEditing:` (text view).

The delegate can respond to this message by updating state information or, for example, by showing an overlay view during the editing session.

3. **During the editing session**—various.

   While the user enters and edits text, the text object invokes certain delegation methods (if implemented). For example, the delegate of a text view can receive a `textViewDidChange:` message when any text changes. The delegate of a text field can receive a `textFieldShouldClear:` message when the user taps the clear button of a text field; the delegate returns a Boolean value indicating whether the text should be cleared.

4. **Just before a text object resigns first responder**—`textFieldShouldEndEditing:` (text field) and `textViewShouldEndEditing:` (text view).

   The primary reason for a delegate to implement these methods is to validate entered text. For example, if text should conform to a given format, the delegate validates the entered string here and returns `NO` if the string does not conform. The default return value is `YES`.

   A related method for text fields is `textFieldShouldReturn:`. When the user taps the return key, the text field class sends a `textFieldShouldReturn:` message to the delegate to ask whether it should resign first responder.

5. **Just after text a object resigns first responder**—`textFieldDidEndEditing:` (text field) and `textViewDidEndEditing:` (text view).

   A delegate can implement these methods to get the text that the user has just entered or edited.

Objects other than the delegate can be informed of changes in the first-responder status of text views and text fields by observing notifications. (They can't, however, approve or deny the transition to a new status.) The notifications have names such as `UITextFieldTextDidBeginEditingNotification`, `UITextViewTextDidEndEditingNotification`, and `UITextViewTextDidChangeNotification`. As with `textFieldDidEndEditing:` and `textViewDidEndEditing:`, the primary reason for observing and handling the `UITextFieldTextDidEndEditingNotification` and `UITextViewTextDidEndEditingNotification` notifications is to access the text in the associated text field or text view. See *UITextField Class Reference* and *UITextView Class Reference* to learn more about the notifications posted by these classes.

Configuring Text Fields and Text Views

As with any view object provided by the UIKit framework, you usually need to configure text fields and text views before they're displayed. You can configure them either programmatically or using the attribute inspector of Interface Builder. In either case, you are setting a property of the text object.

Some properties are common to text views and text fields, and others are specific to each type of object, including the following:

- **Text characteristics**—Text color, alignment, font family, font typeface, and font size.
- **Keyboard**—Keyboard type, return key name, secure text entry, and auto-enabled return key, all of which are declared by the `UITextInputTraits` protocol. (Note that an auto-enabled return key associated with a text view acts as a carriage-return key when tapped.) For more information, see "Configuring the Keyboard for Text Objects" (page 13).

- **Text-field specific—**Border, background image, disabled image, clear button, and placeholder text. As a `UIControl` object, text fields also have highlighted, selected, enabled, and other properties.
- **Text-view specific—**Editable status, data detectors (for phone numbers and URL links). Because a text view inherits from `UIScrollView`, you can also manage scroll-view behavior by setting the appropriate properties.

Tracking Multiple Text Fields or Text Views

All methods of the `UITextFieldDelegate` or `UITextViewDelegate` protocols have a parameter that identifies the text field or text view with the change in first-responder status, the change in value, or any other change that is the reason for the delegation message. If there is only one text object in the currently displayed view, the identity of the text object referenced by the parameter is obvious. However, if the currently displayed view has multiple text fields or text views, the delegate must find a way to identify the text object that is the subject of a delegation message.

You can make this determination using one of two approaches: outlets or tags. For the outlet approach, declare an outlet instance variable (using the `IBOutlet` keyword) and thenmake an outlet connection. In your delegation method, test whether the passed-in text object is the same object referenced by the outlet, using pointer comparison. For example, say you declare and connect an outlet named `SSN`. Your code might look something like Listing 3-1.

Listing 3-1 Identifying the passed-in text object using an outlet

```
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField {
    if (textField == SSN) {
            // .....
            return NO;
        }
    return YES;
}
```

Defining outlet connections for the text objects in a view is especially useful, even essential, when you need to write string values to these objects, not just obtain them.

For the tag approach, declare a set of `enum` constants, one constant for each tag.

```
enum {
    NameFieldTag = 0,
    EmailFieldTag,
    DOBFieldTag,
    SSNFieldTag
};
```

Then assign the integer value to the `tag` property of the text object, either programmatically or in the attribute inspector of Interface Builder. (The `tag` property is declared by `UIView`.) In a delegation method, you can use a switch statement to evaluate the tag value of the passed-in text object and proceed accordingly (as shown in Listing 3-2).

Listing 3-2 Identifying the passed-in text object using tags

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    switch (textField.tag) {
        case NameFieldTag:
            // do something with this text field
            break;
        case EmailFieldTag:
             // do something with this text field
            break;
        // remainder of switch statement....
    }
}
```

Getting the Entered Text and Setting Text

After a user enters or edits text in a text field or text view and the editing session ends, the delegate should get the text and store it in the application's data model. The best delegation methods for accessing entered text are `textFieldDidEndEditing:` (text fields) and `textViewDidEndEditing:` (text views).

Listing 3-3 illustrates how you might get text the user has entered in a text field (differentiating among multiple text fields in a view using tags). The `text` property of `UITextField` or `UITextView` holds the string currently displayed by the text object. The delegate gets the string from this property and stores it in a dictionary object using a key defined for each field. If the text field has no string value—that is, the field holds an empty string—the delegate simply returns.

Listing 3-3 Getting the text entered into a text field

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    if ([textField.text isEqualToString:@""])
        return;

    switch (textField.tag) {
        case NameFieldTag:
            [thePerson setObject:textField.text forKey:MyAppPersonNameKey];
            break;
        case EmailFieldTag:
            [thePerson setObject:textField.text forKey:MyAppPersonEmailKey];
            break;
        case SSNFieldTag:
            [thePerson setObject:textField.text forKey:MyAppPersonSSNKey];
            break;
        default:
            break;
    }
}
```

Listing 3-4 shows an implementation of the `textViewDidEndEditing:` method that gets the displayed string from the text view and stores it in a dictionary. Here the method doesn't ask the text view to resign first responder. (The `resignFirstResponder` method was called earlier in an action method invoked when the user tapped a Done button in the view's user interface.)

Listing 3-4 Getting the text entered into a text view

```
- (void)textViewDidEndEditing:(UITextView *)textView {
    NSString *theText = textView.text;
    if (![theText isEqualToString:@""]) {
        [thePerson setObject:theText forKey:MyAppPersonNotesKey];
    }
    doneButton.enabled = NO;
}
```

If you need to *write* string values to text objects—usually after retrieving them from the application's data model—simply assign the strings to the `text` property of the text object. For example:

```
NSString *storedValue = [thePerson objectForKey:MyAppPersonEmailKey];
emailField.text = storedValue;
```

To do this, it's useful to define outlets for each text field or text view that you want to write string values to (`emailField`, in this example).

Using Formatters with Text Fields

Formatter objects automatically parse strings in a specific format and convert the string to an object representing a number, date, or other value; they also work in reverse, converting `NSDate`, `NSNumber`, and similar objects to a formatted string that represents those object values. The Foundation framework provides the abstract base class `NSFormatter` and two concrete subclasses of that class, `NSDateFormatter` and `NSNumberFormatter`. Using these classes, users can enter values such as the following into a text field:

```
11/15/2010
-1,348.09
```

And your application can use formatter objects to convert the strings into an `NSDate` object and an `NSNumber` object, respectively.

The following code listings use a date-formatter object to illustrate the use of formatters. (Of course, you could use a `UIDatePicker` object for date input rather than a text field, but a text field with an attached date formatter is another option.) The code in Listing 3-5 creates an `NSDateFormatter` object and assigns it to an instance variable. It configures the date formatter to use the "short style" for dates, but in a way that is responsive to changes in calendar, locale, and time zone. It also assigns today's date in the given format as a placeholder string so that users have a model to follow when they enter dates.

Listing 3-5 Configuring a date formatter

```
- (void)viewDidLoad {
    [super viewDidLoad];
    dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setGeneratesCalendarDates:YES];
    [dateFormatter setLocale:[NSLocale currentLocale]];
    [dateFormatter setCalendar:[NSCalendar autoupdatingCurrentCalendar]];
    [dateFormatter setTimeZone:[NSTimeZone defaultTimeZone]];
   [dateFormatter setDateStyle:NSDateFormatterShortStyle]; // example: 4/13/10
   DOB.placeholder = [NSString stringWithFormat:@"Example: %@", [dateFormatter
 stringFromDate:[NSDate date]]];

    // code continues....
}
```

After you have configured the date formatter, the delegate can call the `dateFromString:` method on the formatter to convert the entered date string into an `NSDate` object, as shown in Listing 3-6.

Listing 3-6 Using an `NSDateFormatter` object to convert a date string to a date object

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    [textField resignFirstResponder];
    if ([textField.text isEqualToString:@""])
        return;
    switch (textField.tag) {
        case DOBField:
            NSDate *theDate = [dateFormatter dateFromString:textField.text];;
            if (theDate)
                [inputData setObject:theDate forKey:MyAppPersonDOBKey];
            break;
        // more switch case code here...
        default:
            break;
    }
}
```

The use of formatters does not guarantee that the entered string contains valid values—for example, a user could enter `13` for a month number in the Gregorian calendar. To ensure that the user has entered a correct value, the delegate must validate the string as explained in "Validating Entered Text" (page 28). And because validation often requires a known format and range of valid values, if you configure the date formatter as in Listing 3-5 so that it is sensitive to different calendars and locales, the format cannot be known with certainty. To specify a known date format, configure the date formatter by calling `setDateFormat:`, passing in a format pattern defined by the Unicode standard.

You can also reverse the procedure shown above: Convert a date object to a string in a given format by calling the `NSDateFormatter` method `stringFromDate:` and then assign that string to the `text` property of a text field, text view, or label.

For more information on `NSDateFormatter` and `NSNumberFormatter`, see *Data Formatting Guide*.

Validating Entered Text

An application sometimes cannot accept the strings entered in text fields and text views without validating the value first. Perhaps the string must be in a certain format, or the value (after it is converted to a numeric value) must fall within a certain range. The best delegation methods for validating entered strings are `textFieldShouldEndEditing:` for text fields and `textViewShouldEndEditing:` for text views. These methods are called just before the text field or text view resigns first responder status. Returning `NO` prevents that from happening, and consequently the text object remains the focus of editing. If an entered string is invalid, you should also display an alert to inform the user of the error.

Listing 3-7 uses a regular expression to verify that the string entered in a "Social Security Number" field conforms to the format for such numbers.

Listing 3-7 Validating the format of a text field's string using a regular expression

```
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField {
    if (textField == SSN) { // SSN is an outlet
        NSString *regEx = @"[0-9]{3}-[0-9]{2}-[0-9]{4}";
        NSRange r = [textField.text rangeOfString:regEx
options:NSRegularExpressionSearch];
        if (r.location == NSNotFound) {
           UIAlertView *av = [[[UIAlertView alloc] initWithTitle:@"Entry Error"
                message:@"Enter social security number in 'NNN-NN-NNNN' format"
                delegate:self cancelButtonTitle:@"OK" otherButtonTitles:nil]
autorelease];
            [av show];
            return NO;
        }
    }
        return YES;
}
```

The implementation of `textViewShouldEndEditing:` in Listing 3-8 enforces a character limit for the text entered in a text view.

Listing 3-8 Validating a text view's string for allowable length

```
- (BOOL)textViewShouldEndEditing:(UITextView *)textView {
      if (textView.text.length > 50) {
        UIAlertView *av = [[[UIAlertView alloc] initWithTitle:@"Entry Error"
            message:@"You must enter less than 50 characters." delegate:self
cancelButtonTitle:@"OK"
            otherButtonTitles:@"Clear", nil] autorelease];
        [av show];
        return NO;
    }
    return YES;
}
```

The delegate can also validate each character as it is entered into a text field by implementing the `textField:shouldChangeCharactersInRange:replacementString:` method. The code in Listing 3-9 verifies that each entered character (`string`) represents a digit. (You could accomplish the same goal by specifying a `UIKeyboardTypeNumberPad` keyboard for the text field.)

Listing 3-9 Validating each character as it's entered

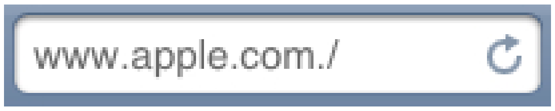
```
- (BOOL)textField:(UITextField *)textField
shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString *)string {
    if ([string isEqualToString:@""]) return YES;
    if (textField.tag == SalaryFieldTag) {
        unichar c = [string characterAtIndex:0];
        if ([[NSCharacterSet decimalDigitCharacterSet] characterIsMember:c]) {
            return YES;
        } else {
            return NO;
        }
    }

    return YES;
}
```

You can also implement the `textField:shouldChangeCharactersInRange:replacementString:` method to offer possible word completions or corrections to the user as they enter text.

Using Overlay Views in Text Fields

Overlay views are small views inserted into the left and right corners of a text field. They act as controls when users tap them (frequently they are buttons) and act on the current contents of the text field. Searching and bookmarking are two common tasks for overlay views, but others are possible. This overlay view loads a web browser using the (partial) URL in the text field:



To implement an overlay view, create a view of a size that fits within the height of the text field and give the view an appropriately sized image. If the view is a button or other control, specify a target object, an action selector, and the triggering control events. Usually you want an overlay view to appear when its text field is the focus of editing, so assign it to the text field's `leftView` or `rightView` property in the delegate's `textFieldDidBeginEditing:` method. You can control when an overlay view appears during the editing session—for example, before the user begins entering text or only after the user begins entering text—by assigning a `UITextFieldViewMode` constant to the `leftViewMode` or `rightViewMode` property. Listing 3-10 illustrates how you might implement an overlay view this.

Listing 3-10 Displaying an overlay view in a text field

```
- (void)textFieldDidBeginEditing:(UITextField *)textField {
     if (textField.tag == NameField && self.overlayButton) {
        textField.leftView = self.overlayButton;
        textField.leftViewMode = UITextFieldViewModeAlways;
    }
}

@dynamic overlayButton;

- (UIButton *)overlayButton {
    if (!overlayButton) {
        overlayButton = [[UIButton buttonWithType:UIButtonTypeCustom] retain];
        UIImage *overlayImage = [UIImage imageNamed:@"bookmark.png"];
        if (overlayImage) {
           [overlayButton setImage:overlayImage forState:UIControlStateNormal];
            [overlayButton addTarget:self action:@selector(bookmarkTapped:)
forControlEvents:UIControlEventTouchUpInside];
        }
    }
    return overlayButton;
}
```

If you use a control for an overlay view, be sure to implement the action method.

To remove an overlay view, simply set the `leftView` or `rightView` property to `nil` in the `textFieldDidEndEditing:` delegation method, as in Listing 3-11.

Listing 3-11 Removing the overlay view

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    if (textField.tag == NameFieldTag) {
        textField.leftView = nil;
    }
    // remainder of implementation....
}
```

Tracking the Selection in Text Views

The `textViewDidChangeSelection:` method of `UITextViewDelegate` lets you track changes to the selections that a user makes in a text view. You can implement the method to obtain the selected substring and do something with it. Listing 3-12 is a whimsical example that makes all characters in the selected substring uppercase.

Listing 3-12 Getting the selected substring and changing it

```
- (void)textViewDidChangeSelection:(UITextView *)textView {
    NSRange r = textView.selectedRange;
    if (r.length == 0) {
        return;
    }
    NSString *selText = [textView.text substringWithRange:r];
    NSString *upString = [selText uppercaseString];
    NSString *newString = [textView.text stringByReplacingCharactersInRange:r
withString:upString];
    textView.text = newString;
}
```

Displaying Web Content

Note: This chapter contains information that used to be in *iPhone Application Programming Guide*. The information in this chapter has not been updated specifically for iOS 4.0.

If your user interface includes a `UIWebView` object, you can display local content or content that is loaded from the network. When loading local content, you can either create the content dynamically or load it from a file and display it using the `loadData:MIMEType:textEncodingName:baseURL:` or `loadHTMLString:baseURL:` method. The method in Listing 4-1 uses the `loadData:MIMEType:textEncodingName:baseURL:` method to load the contents of a PDF file into a web view.

Listing 4-1 Loading a local PDF file into the web view

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

    NSString *thePath = [[NSBundle mainBundle]
pathForResource:@"iPhone_User_Guide" ofType:@"pdf"];
    if (thePath) {
        NSData *pdfData = [NSData dataWithContentsOfFile:thePath];
        [(UIWebView *)self.view loadData:pdfData MIMEType:@"application/pdf"
            textEncodingName:@"utf-8" baseURL:nil];
    }
}
```

To load content from the network, you create an `NSURLRequest` object and pass it to the `loadRequest:` method of your web view.

```
[self.myWebView loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com/"]]];
```

Because loading a web resource might take some time, you might display an activity indicator to indicate that the load is underway. You can do this by assigning a delegate to the web view and implementing the `UIWebViewDelegate` methods, as in Listing 4-2. The delegate displays an activity indicator when the load starts and hides it when the load ends. If there is a problem with the load, it creates an HTML error message and, using the `loadHTMLString:baseURL:` method, loads it into the web view for display.

Listing 4-2 The web-view delegate managing network loading

```
- (void)webViewDidStartLoad:(UIWebView *)webView
{
    // starting the load, show the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
}

- (void)webViewDidFinishLoad:(UIWebView *)webView
{
    // finished loading, hide the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}

- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error
{
    // load error, hide the activity indicator in the status bar
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;

    // report the error inside the webview
    NSString* errorString = [NSString stringWithFormat:
                            @"<html><center><font size=+5 color='red'>An error
 occurred:<br>%@</font></center></html>",
                             error.localizedDescription];
    [self.myWebView loadHTMLString:errorString baseURL:nil];
}
```

If, after initiating a network-based load request, you must release your web view for any reason, you must cancel the pending request before releasing the web view. You can cancel a load request using the web view's `stopLoading` method. A typical place to include this code would be in the `viewWillDisappear:` method of the owning view controller. To determine if a request is still pending, you can check the value in the web view's `loading` property. Listing 4-3 illustrates how you might do this.

Listing 4-3 Stopping a load request when the web view is to disappear

```
- (void)viewWillDisappear:(BOOL)animated
{
    if ( [self.myWebView loading] ) {
        [self.myWebView stopLoading];
    }
    self.myWebView.delegate = nil;    // disconnect the delegate as the webview
 is hidden
    [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}
```

The `loadRequest:` example is taken from the *UICatalog* sample code project.

Drawing and Managing Text

An application that wants to display and process text is not limited to the text and web objects of the UIKit framework. It can implement custom views that are capable of anything from simple text entry to complex text processing and custom input. Through the available programming interfaces, these applications can acquire features such as custom text layout, multistage input, and autocorrection, custom keyboards, and spell-checking.

Notes: This chapter contains information that used to be in *iPad Programming Guide*. The information in this chapter has not been updated specifically for iOS 4.0.

Device Features Guide for iOS describes custom input views and custom input accessory views.

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.

Simple Text Drawing

In addition to the UIKit classes for displaying and editing text, iOS also includes several ways to draw text directly on the screen. The easiest and most efficient way to draw simple strings is using the UIKit additions to the `NSString` class, which is in a category named `UIStringDrawing`. These extensions include methods for drawing strings using a variety of attributes wherever you want them on the screen. There are also methods for computing the size of a rendered string before you actually draw it, which can help you lay out your application content more precisely.

Important: There are some good reasons to avoid drawing text directly in favor of using the text objects of the UIKit framework. One is performance. Although, a `UILabel` object also draws its static text, it draws it only once whereas a text-drawing routine is typically called repeatedly. Text objects also afford more interaction; for example, they are selectable.

The methods of `UIStringDrawing` 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. The methods of `UIStringDrawing` allow you to adjust the position of the rendered text precisely and blend it with the rest of your view's content. They also let you compute the bounding rectangle for your text in advance based on the desired font and style attributes.

You can also use the `CATextLayer` class of Core Animation to do simple text drawing. An object of this class 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.

For information about the string-drawing extensions to `NSString`, see *NSString UIKit Additions Reference*. To learn more about `CATextLayer`, `CALayer`, and the other classes of Core Animation, read *Core Animation Programming Guide*.

Core Text

Core Text is a technology for sophisticated text layout and font management. It is intended to be used by applications that rely heavily on text processing—for example, book readers and word processors. It is implemented as a framework that publishes an API similar to that of Core Foundation—similar in that it is procedural (ANSI C) but is based on object-like opaque types. This API is integrated with both Core Foundation and Core Graphics. 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.

Notes: 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*.

If you use Core Text or Core Graphics to draw text, remember that you must apply a flip transform to the current text matrix 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. For more information, 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

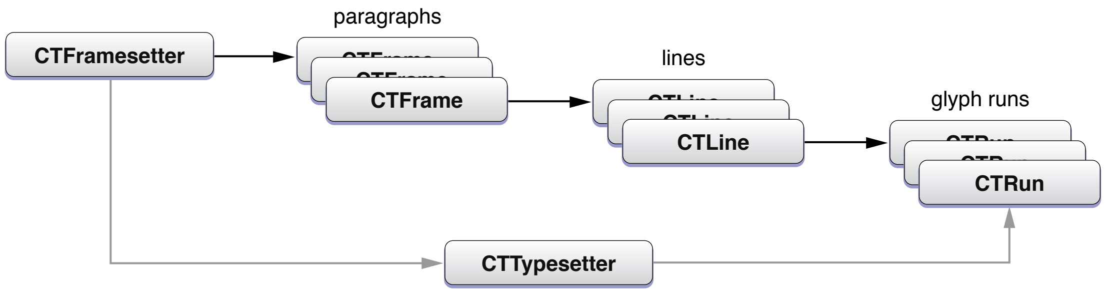
Core Text requires two objects whose opaque types are not native to it: an attributed string (`CFAttributedStringRef`) and a graphics path (`CGPathRef`). An attributed-string object encapsulates a string backing the displayed text and includes properties (or, "attributes") that define stylistic aspects 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 Figure 5-1 (page 37)). 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 `CTLine` 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. Although a typesetter object returns `CTLine` objects, it composes those lines from arrays of glyph runs.

Figure 5-1 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 is central to fonts in Core Text. You can use graphics-context functions to set the current font and draw glyphs; or you can create a `CTLine` object from an attributed string and use its functions to draw into the graphics 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.

It's possible to convert `UIFont` objects to `CTFont` objects by calling `CTFontCreateWithName`, passing the font name and point size encapsulated by the `UIFont` object.

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.

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`.

For more information about these functions and their use, see *Quartz 2D Programming Guide* and *Core Graphics Framework Reference*.

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 described here was introduced in iOS 3.2. iOS 4.0 introduces fuller support for regular expressions with the `NSRegularExpression` class. 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 5-1 gives an example of using the `NSString` regular-expression API.

Listing 5-1 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 as 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 39), the ICU libraries related to regular expressions are included in iOS 3.2. However, you should use the ICU facilities only 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 ICU 4.2.1 is included in iOS 3.2 at the BSD (nonframework) 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, namely:

```
parseerr.h
platform.h
putil.h
uconfig.h
udraft.h
uintrnal.h
uiter.h
umachine.h
uregex.h
```

```
uréname.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.

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 the 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 subset of the available keyboards and languages are available to classes that adopt only the `UIKeyInput` protocol. For example, any multi-stage input method, such Chinese, Japanese, Korean, and Thai is excluded. If a class also adopts the `UITextInput` protocol, those input methods are then available.

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 35). 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 5-2 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 5-2 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];
}
```

To actually draw the text in the view, this code uses the `drawInRect:withFont:` from the `UIStringDrawing` category on `NSString`. See "Facilities for Text Drawing and Text Processing" (page 35) for more about `UIStringDrawing`.

Communicating with the Text Input System

The text input system of iOS manages the keyboard. It interprets taps as presses of specific keys in specific keyboards suitable for certain languages. It then sends the associated character to the target view for insertion. As explained in "Simple Text Input" (page 40), 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. For example, it 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 Kanji (Japanese) and Hanzi (Chinese), which 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.

The following section describes the general responsibilities of a custom text view that communicates with the text input system. "A Guided Tour of a UITextInput Implementation" (page 45) examines the most important classes and methods of a typical implementation of `UITextInput`.

Overview of the Client Side of Text Input

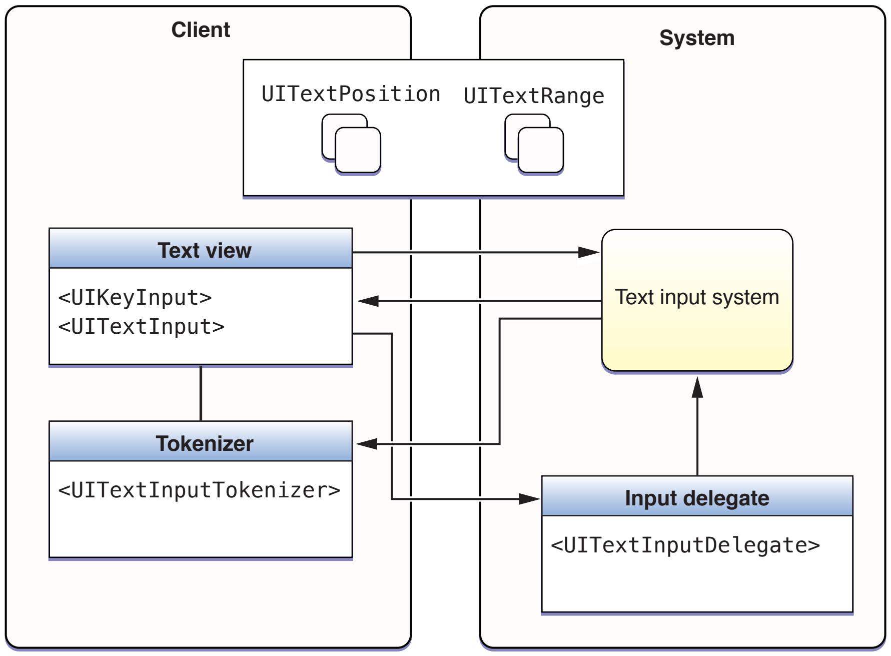
A class that wants to communicate with the text input system must adopt the `UITextInput` protocol. The class needs to inherit from `UIResponder` and is in most cases a custom view.

Note: The responder class that adopts `UITextInput` does not have to the the view that draws and manages text (as is the case in the sample code analyzed in "A Guided Tour of a UITextInput Implementation" (page 45)). However, if it isn't the view that draws and manages text, the class that does adopt `UITextInput` should be able to communicate directly with the view that does. For simplicity's sake, the following discussion refers to the responder class that adopts `UITextInput` as the *text view*.

The text view must do its own text layout and font management; for this purpose, the Core Text framework is recommended. ("Facilities for Text Drawing and Text Processing" (page 35) gives an overview of Core Text.) The class should also adopt and implement the `UIKeyInput` protocol and should set the necessary properties of the `UITextInputTraits` protocol.

The general architecture of the client and system sides of the text input system are diagrammed in Figure 5-2.

Figure 5-2 Paths of communication with the text input system



The text input system calls the `UITextInput` methods that the text view implements. Many of these methods request information about specific text positions and text ranges from the text view and pass the same information back to the class in other method calls. The reasons for these exchanges of text positions and text ranges are summarized in "Tasks of a UITextInput Object" (page 43).

Text positions and text ranges in the text input system are represented by instances of custom classes. "Text Positions and Text Ranges" (page 43) discusses these objects in more detail.

The text view also maintains references to a tokenizer and an input delegate. The text view calls methods declared by the `UITextInputDelegate` protocol to notify a system-provided input delegate about external changes in text and selection. The text input system 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. The text view includes a property (declared by `UITextInput`) that holds a reference to the tokenizer.

Text Positions and Text Ranges

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

Although `UITextPosition` itself declares no methods or properties, it is an essential part of the information exchanged between a text document and the text input system. The text input 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 can 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 can encapsulate a single offset or index 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 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 text view must assign text positions to properties marking the beginning and end of the displayed text. In addition, it must also 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 methods implemented by a `UITextInput` object can be divided into distinctive tasks:

- **Returning and replacing text by text range.** Given a range, either return the text in that range or replace that text with text provided by the text input system.

  ```
  textInRange:
  replaceRange:withText:
  ```

- **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.

  ```
  positionFromPosition:offset:
  positionFromPosition:inDirection:offset:
  textRangeFromPosition:toPosition:
  ```

- **Evaluating text positions.** Compare two text positions or return the offset from one text position to another.

  ```
  comparePosition:toPosition:
  offsetFromPosition:toPosition:
  ```

- **Answering layout questions.** Determine a text position or text range by extending in a given layout direction.

  `positionWithinRange:farthestInDirection:`
  `characterRangeByExtendingPosition:inDirection:`

- **Hit-testing.** Given a point, return the closest text position or text range.

  `closestPositionToPoint:`
  `closestPositionToPoint:withinRange:`
  `characterRangeAtPoint:`

- **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.

  `firstRectForRange:`
  `caretRectForPosition:`

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).

When changes occur in the text view due to external reasons—that is, they aren't caused by calls from the text input system—the `UITextInput` object should send `textWillChange:`, `textDidChange:`, `selectionWillChange:`, and `selectionDidChange:` messages to the input delegate (which it holds a reference to). For example, when users tap a text view and you set the range of selected text to place the insertion point under the finger, you would send `selectionWillChange:` before you change the selected range, and you send `selectionDidChange:` after you change the range.

Tokenizers

Tokenizers are objects that determine whether a text position is within or at the boundary of a text unit with a given granularity. When queried by the text input system, a tokenizer returns ranges of text units with a given granularity or the boundary text position for a text unit with a given 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.

The text input system uses the tokenizer in a variety of ways. For example, the keyboard might require the last sentence's worth of context to figure out what the user is trying to type. Or, if the user is pressing the Option-left arrow key (on an external keyboard), the text system queries the tokenizer to find the information it needs to move to the previous word.

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 all supported languages. 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. Otherwise you should subclass `UITextInputStringTokenizer` to provide application-specific information about layout directions.

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

A Guided Tour of a UITextInput Implementation

SimpleTextInput is a simple text-editing application based on Core Text. It has two custom subclasses of `UIView`. One view subclass, `SimpleCoreTextView`, provides text layout and editing support using the facilities of Core Text. The other view subclass, `EditableCoreTextView`, adopts the `UIKeyInput` protocol to enable text input; it also adopts the `UITextInput` protocol and creates and implements the related subclasses to communicate with the text input system. `EditableCoreTextView` embeds `SimpleCoreTextView` as an instance variable, instantiates it,and calls through to it in most `UITextInput` and `UIKeyInput` method implementations.

Note: For reasons of space, the guided tour shows implementations of the `UITextInput` methods that are most important or illustrative. However, it is possible to extrapolate from these chosen implementations to the others of the protocols. The code was taken from the *SimpleTextInput* sample code project.

Subclasses of UITextPosition and UITextRange

`EditableCoreTextView` creates a custom subclass of `UITextPosition` called `IndexedPosition` and a custom subclass of `UITextRange` called `IndexedRange`. These subclasses simply encapsulate a single index value and an `NSRange` value based on two of those indexes. Listing 5-3 shows the declaration of these classes.

Listing 5-3 Declaring the `IndexedPosition` and `IndexedRange` classes

```
@interface IndexedPosition : UITextPosition {
    NSUInteger _index;
    id <UITextInputDelegate> _inputDelegate;
}
@property (nonatomic) NSUInteger index;
+ (IndexedPosition *)positionWithIndex:(NSUInteger)index;
@end

@interface IndexedRange : UITextRange {
    NSRange _range;
}
@property (nonatomic) NSRange range;
+ (IndexedRange *)rangeWithNSRange:(NSRange)range;

@end
```

Both classes declare class factory methods to vend instances. Listing 5-4 shows the implementation of these methods as well as the methods declared by the `UITextRange` class.

Listing 5-4 Implementing the `IndexedPosition` and `IndexedRange` classes

```
@implementation IndexedPosition
@synthesize index = _index;

+ (IndexedPosition *)positionWithIndex:(NSUInteger)index {
    IndexedPosition *pos = [[IndexedPosition alloc] init];
    pos.index = index;
    return [pos autorelease];
}

@end

@implementation IndexedRange
@synthesize range = _range;

+ (IndexedRange *)rangeWithNSRange:(NSRange)nsrange {
    if (nsrange.location == NSNotFound)
        return nil;
    IndexedRange *range = [[IndexedRange alloc] init];
    range.range = nsrange;
    return [range autorelease];
}

- (UITextPosition *)start {
    return [IndexedPosition positionWithIndex:self.range.location];
}

- (UITextPosition *)end {
        return [IndexedPosition positionWithIndex:(self.range.location +
self.range.length)];
}

-(BOOL)isEmpty {
    return (self.range.length == 0);
}
@end
```

Inserting and Deleting Text

A text view that adopts the `UITextInput` protocol must also adopt the `UIKeyInput` protocol. That means it must implement the `insertText:`, `deleteBackward`, and `hasText` methods as discussed in "Simple Text Input" (page 40). Because the `EditableCoreTextView` class is adopting `UITextInput`, it must also maintain the selected and marked text ranges (that is, the current values of the `selectedTextRange` and `markedTextRange` properties) as text is entered and deleted.

Listing 5-5 illustrates how `EditableCoreTextView` does this when text is entered. If there is marked text when a character is entered, it replaces the marked text with the character by calling the `replaceCharactersInRange:withString:` method on the backing mutable string. If there is a selected range of text, it replaces the characters in that range with the input character. Otherwise, the method inserts the input character at the caret.

Listing 5-5 Inserting text input into storage and updating selected and marked ranges

```
- (void)insertText:(NSString *)text {
    NSRange selectedNSRange = _textView.selectedTextRange;
    NSRange markedTextRange = _textView.markedTextRange;

    if (markedTextRange.location != NSNotFound) {
        [_text replaceCharactersInRange:markedTextRange withString:text];
        selectedNSRange.location = markedTextRange.location + text.length;
        selectedNSRange.length = 0;
        markedTextRange = NSMakeRange(NSNotFound, 0);
    } else if (selectedNSRange.length > 0) {
        [_text replaceCharactersInRange:selectedNSRange withString:text];
        selectedNSRange.length = 0;
        selectedNSRange.location += text.length;
    } else {
        [_text insertString:text atIndex:selectedNSRange.location];
        selectedNSRange.location += text.length;
    }
    _textView.text = _text;
    _textView.markedTextRange = markedTextRange;
    _textView.selectedTextRange = selectedNSRange;
}
```

Even though the structure of the `deleteBackward` method implemented by `EditableCoreTextView` is identical to the `insertText:` method, there are appropriate differences in how the selected and marked text ranges are adjusted. Another difference is that the `deleteCharactersInRange:` method is called on the backing mutable string rather than `replaceCharactersInRange:withString:`.

Returning and Replacing Text by Range

Any text view that communicates with the text input system must, when requested, return a specified range of text and replace a range of text with a given string. The classes in our example, `EditableCoreTextView` and `SimpleCoreTextView`, maintain synchronized copies of the backing string object (`EditableCoreTextView` as a `NSMutableString` object). The implementations of `textInRange:` and `replaceRange:withText:` in Listing 5-6 call the appropriate `NSString` methods on the backing string to accomplish their essential functions.

Listing 5-6 Implementations of `textInRange:` and `replaceRange:withText:`

```
- (NSString *)textInRange:(UITextRange *)range
{
    IndexedRange *r = (IndexedRange *)range;
    return ([_text substringWithRange:r.range]);
}

- (void)replaceRange:(UITextRange *)range withText:(NSString *)text
{
    IndexedRange *r = (IndexedRange *)range;
    NSRange selectedNSRange = _textView.selectedTextRange;
    if ((r.range.location + r.range.length) <= selectedNSRange.location) {
        selectedNSRange.location -= (r.range.length - text.length);
    } else {
        // Need to also deal with overlapping ranges.
    }
    [_text replaceCharactersInRange:r.range withString:text];
    _textView.text = _text;
    _textView.selectedTextRange = selectedNSRange;
}
```

When the `text` property of `SimpleCoreTextView` changes (as shown in the implementation of `replaceRange:withText:`), `SimpleCoreTextView` lays out the text again and redraws it using Core Text functions.

Maintaining Selected and Marked Text Ranges

Because editing operations are performed on selected and marked text, the text input system frequently requests that the text view return and set the ranges of selected and marked text. Listing 5-7 shows how `EditableCoreTextView` returns the ranges of selected and marked text by implementing getter methods for the `selectedTextRange` and `markedTextRange` properties.

Listing 5-7 Returning ranges of selected and marked text

```
- (UITextRange *)selectedTextRange {
    return [IndexedRange rangeWithNSRange:_textView.selectedTextRange];
}

- (UITextRange *)markedTextRange {
    return [IndexedRange rangeWithNSRange:_textView.markedTextRange];
}
```

The setter method for the `selectedTextRange` in Listing 5-8 simply sets the selected-text range on the embedded text view. The `setMarkedText:selectedRange:` method is more complex because, as you may recall, the range of marked text contains within it the range of selected text (even if the range merely identifies the caret), and these ranges have to be reconciled to reflect the situation after the insertion of text.

Listing 5-8 Setting the range of selected text and setting the marked text

```
- (void)setSelectedTextRange:(UITextRange *)range
{
    IndexedRange *r = (IndexedRange *)range;
    _textView.selectedTextRange = r.range;
}

- (void)setMarkedText:(NSString *)markedText selectedRange:(NSRange)selectedRange
 {
    NSRange selectedNSRange = _textView.selectedTextRange;
    NSRange markedTextRange = _textView.markedTextRange;

    if (markedTextRange.location != NSNotFound) {
        if (!markedText)
            markedText = @"";
       [_text replaceCharactersInRange:markedTextRange withString:markedText];
        markedTextRange.length = markedText.length;
    } else if (selectedNSRange.length > 0) {
       [_text replaceCharactersInRange:selectedNSRange withString:markedText];
        markedTextRange.location = selectedNSRange.location;
        markedTextRange.length = markedText.length;
    } else {
        [_text insertString:markedText atIndex:selectedNSRange.location];
        markedTextRange.location = selectedNSRange.location;
        markedTextRange.length = markedText.length;
    }
    selectedNSRange = NSMakeRange(selectedRange.location +
markedTextRange.location,
        selectedRange.length);

    _textView.text = _text;
    _textView.markedTextRange = markedTextRange;
    _textView.selectedTextRange = selectedNSRange;
}
```

Note that `EditableCoreTextView` replaces the text by calling the `replaceCharactersInRange:withString:` method on its mutable string object, which it then assigns to the `text` property of the embedded text view.

Frequently Called UITextInput Methods

When users type characters on the keyboard and when those characters enter text storage and are laid out, the text input system requests information from the object adopting the `UITextInput` protocol. Three of the more frequently called methods are `textRangeFromPosition:toPosition:`, `offsetFromPosition:toPosition:`, and `positionFromPosition:offset:`.

The text input system calls `positionFromPosition:offset:` to get the position in the text that's a given offset from another position. Listing 5-9 shows how `EditableCoreTextView` implements this method (which includes range checking).

Listing 5-9 Implementing `positionFromPosition:offset:`

```
- (UITextPosition *)positionFromPosition:(UITextPosition *)position
offset:(NSInteger)offset {
    IndexedPosition *pos = (IndexedPosition *)position;
    NSInteger end = pos.index + offset;
    if (end > _text.length || end < 0)
        return nil;
    return [IndexedPosition positionWithIndex:end];
}
```

The `offsetFromPosition:toPosition:` method should satisfy the opposite request and return a value specifying the offset between two text positions. `EditableCoreTextView` implements it as shown in Listing 5-10.

Listing 5-10 Implementing `offsetFromPosition:toPosition:`

```
- (NSInteger)offsetFromPosition:(UITextPosition *)from toPosition:(UITextPosition
 *)toPosition {
    IndexedPosition *f = (IndexedPosition *)from;
    IndexedPosition *t = (IndexedPosition *)toPosition;
    return (t.index - f.index);
}
```

Finally, the text input system frequently asks a text view for a text range that falls between two text positions. Listing 5-11 shows an implementation of `textRangeFromPosition:toPosition:` that returns this range.

Listing 5-11 Implementing `textRangeFromPosition:toPosition:`

```
- (UITextRange *)textRangeFromPosition:(UITextPosition *)fromPosition
          toPosition:(UITextPosition *)toPosition {
    IndexedPosition *from = (IndexedPosition *)fromPosition;
    IndexedPosition *to = (IndexedPosition *)toPosition;
    NSRange range = NSMakeRange(MIN(from.index, to.index), ABS(to.index -
from.index));
    return [IndexedRange rangeWithNSRange:range];
}
```

Returning Rectangles

When a correction bubble appears and when the user types in Japanese, the text input system sends `firstRectForRange:` and `caretRectForPosition:` to the text view. The purpose of both of these methods is to return a rectangle enclosing either a range of text or the caret that marks the insertion point. The `EditableCoreTextView` class implements the first of these methods by calling a method of its embedded text view that maps the range to an enclosing rectangle (see Listing 5-12). Before returning the rectangle, it converts it to the local coordinate system.

Listing 5-12 An implementation of `firstRectForRange:`

```
- (CGRect)firstRectForRange:(UITextRange *)range {
    IndexedRange *r = (IndexedRange *)range;
    CGRect rect = [_textView firstRectForNSRange:r.range];
    return [self convertRect:rect fromView:_textView];
}
```

The embedded text view in this case performs the lion's share of the work. Using Core Text functions, it computes the rectangle that encloses the range of text and returns it, as shown in Listing 5-13.

Listing 5-13 Mapping text range to enclosing rectangle

```
- (CGRect)firstRectForNSRange:(NSRange)range; {
    int index = range.location;
    NSArray *lines = (NSArray *) CTFrameGetLines(_frame);
    for (int i = 0; i < [lines count]; i++) {
        CTLineRef line = (CTLineRef) [lines objectAtIndex:i];
        CFRange lineRange = CTLineGetStringRange(line);
        int localIndex = index - lineRange.location;
        if (localIndex >= 0 && localIndex < lineRange.length) {
            int finalIndex = MIN(lineRange.location + lineRange.length,
                range.location + range.length);
            CGFloat xStart = CTLineGetOffsetForStringIndex(line, index, NULL);
          CGFloat xEnd = CTLineGetOffsetForStringIndex(line, finalIndex, NULL);
            CGPoint origin;
            CTFrameGetLineOrigins(_frame, CFRangeMake(i, 0), &origin);
            CGFloat ascent, descent;
            CTLineGetTypographicBounds(line, &ascent, &descent, NULL);

            return CGRectMake(xStart, origin.y - descent, xEnd - xStart, ascent
 + descent);
        }
    }
    return CGRectNull;
}
```

For `caretRectForPosition:`, the approach you take would be somewhat different. Selection affinity (`selectionAffinity`) is a factor to consider; more importantly, keep in mind that the height and width of the caret rectangle can be different from the bounding rectangle returned from `firstRectForRange:`.

Hit Testing

Another area where the text input system asks the text view to map between the display of text and the storage of text is hit testing. Given a point in the text view (the text input system asks), what is the corresponding text position or text range? The `UITextInput` methods it calls for this information are `closestPositionToPoint:`, `closestPositionToPoint:withinRange:`, and `characterRangeAtPoint:`. Listing 5-14 illustrates how `EditableCoreTextView` implements the first of these methods.

Listing 5-14 An implementation of `closestPositionToPoint:`

```
- (UITextPosition *)closestPositionToPoint:(CGPoint)point {
    NSInteger index = [_textView closestIndexToPoint:point];
    return [IndexedPosition positionWithIndex:(NSUInteger)index];
}
```

Here, as with the methods that return rectangles for text ranges or text positions, `EditableCoreTextView` calls a method of its embedded view that uses Core Text to calculate the character index that corresponds to the point. Listing 5-15 illustrates how the embedded view accomplishes this.

Listing 5-15 Mapping a point to a character index

```
- (NSInteger)closestIndexToPoint:(CGPoint)point {
    NSArray *lines = (NSArray *) CTFrameGetLines(_frame);
    CGPoint origins[lines.count];
    CTFrameGetLineOrigins(_frame, CFRangeMake(0, lines.count), origins);

    for (int i = 0; i < lines.count; i++) {
        if (point.y > origins[i].y) {
            CTLineRef line = (CTLineRef) [lines objectAtIndex:i];
            return CTLineGetStringIndexForPosition(line, point);
        }
    }
    return  _text.length;
}
```

Informing the Text Input Delegate of Changes

When there is a change in text or a change in selection that is not initiated by the text input system, you should inform the text input delegate by sending it an appropriate “will-change” method. After making the change, send the delegate the corresponding “did-change” method.

The text input delegate is a system-provided object that adopts the `UITextInputDelegate` protocol. If the class adopting the `UITextInput` protocol defines an `inputDelegate` property, the text input system automatically assigns a delegate object to this property at runtime.

Listing 5-16 shows an action method that is invoked when the user taps in the text view. If the view is tapped but it isn’t the first responder, the text view makes itself the first responder and starts an editing session. If the view is subsequently tapped, the text view sends a `selectionWillChange:` message to the text input delegate. It then clears any marked text range and resets the selected text range so that the caret is at the point in the text where the tapped occurred. After this, it calls `selectionDidChange:`.

Listing 5-16 Sending messages to the text input delegate

```
- (void)tap:(UITapGestureRecognizer *)tap
{
    if (![self isFirstResponder]) {
        _textView.editing = YES;
        [self becomeFirstResponder];
    } else {
        [self.inputDelegate selectionWillChange:self];

        NSInteger index = [_textView closestIndexToPoint:[tap
locationInView:_textView]];
        _textView.markedTextRange = NSMakeRange(NSNotFound, 0);
        _textView.selectedTextRange = NSMakeRange(index, 0);

        [self.inputDelegate selectionDidChange:self];
    }
}
```

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 41).

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 5-17.

Listing 5-17 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 5-17 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 for an iPad application 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 5-18 shows how you might call this method and display a table view listing the completions in a popover view.

Listing 5-18 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];
}
```

Document Revision History

This table describes the changes to *Text and Web Programming Guide for iOS*.

Date	Notes
2010-07-07	Changed the title from "Text and Web Programming Guide for iPhone OS."
2010-05-11	First version of a document that describes the technologies and techniques for displaying and managing text and web content in iOS.