

Table View

Programming Guide for iOS

Contents

About Table Views in iOS-Based Applications 7

At a Glance 8

Table Views are Composed From Cells in a Particular Style 8

There Are Several Ways to Create a Table View But the Basic Pattern is the Same 8

Responding to Selections of Rows 9

In Editing Mode You Can Add, Delete, and Reorder Rows 9

How to Use This Document 9

See Also 10

Table View Styles and Accessory Views 11

Table View Styles 11

Standard Styles for Table-View Cells 15

Accessory Views 19

Overview of the Table View API 21

Table View 21

Data Source and Delegate 21

Extension to the NSIndexPath Class 22

Table View Cells 22

Navigating a Data Hierarchy With Table Views 23

Hierarchical Data Models and Table Views 23

The Data Model as a Hierarchy of Model Objects 23

Table Views and the Data Model 24

View Controllers and Navigation-Based Applications 26

Overview of View Controllers and Navigation Controllers 26

Table-View Controllers 28

Managing Table Views In a Navigation-Based Application 30

Design Pattern for Navigation-Based Applications 33

Creating and Configuring a Table View 35

Basics of Table View Creation 35

Recommendations for Creating and Configuring Table Views 37

Creating a Table View Application the Easy Way 37

Examining the Project: How Things are Set Up	38
Adding Table Views to the Application	41
Creating a Table View Programmatically	42
Populating the Table View With Data	44
Populating an Indexed List	45
Optional Table-View Configurations	51
A Closer Look at Table-View Cells	54
Characteristics of Cell Objects	54
Using Cell Objects in Predefined Styles	55
Customizing Cells	59
Programmatically Adding Subviews to a Cell's Content View	60
Loading Custom Table-View Cells From Nib Files	64
Subclassing UITableViewCell	71
Cells and Table-View Performance	77
Managing Selections	78
Selections in Table Views	78
Responding to Selections	78
Programmatically Selecting and Scrolling	82
Inserting and Deleting Rows and Sections	84
Inserting and Deleting Rows in Editing Mode	85
When a Table View is Edited	85
An Example of Deleting a Table-View Row	88
An Example of Adding a Table-View Row	89
Batch Insertion, Deletion, and Reloading of Rows and Sections	91
An Example of Batched Insertion and Deletion Operations	92
Ordering of Operations and Index Paths	93
Managing the Reordering of Rows	95
What Happens When a Row is Relocated	96
Examples of Moving a Row	97
Document Revision History	99

Figures and Listings

About Table Views in iOS-Based Applications 7

Figure I-1 Table views of various kinds 7

Table View Styles and Accessory Views 11

Figure 1-1 A table view in the plain style (no section header or footer) 11

Figure 1-2 A table view configured as an section index 12

Figure 1-3 A table view configured as a selection list 13

Figure 1-4 A table view in the grouped style 14

Figure 1-5 Header and footer of a section 15

Figure 1-6 Default table row style (no subtitle) 16

Figure 1-7 Table row style with subtitle under title 17

Figure 1-8 Table row style with right-aligned subtitle 18

Figure 1-9 Table row style in Contacts format 19

Navigating a Data Hierarchy With Table Views 23

Figure 3-1 Mapping levels of the data model to table views 25

Figure 3-2 Navigation bars and common control items 27

Figure 3-3 Navigation controller and view controllers in a navigation-based application 28

Listing 3-1 Setting up the root view controller—window in nib file 30

Listing 3-2 Setting up the root view controller—window created programmatically 31

Listing 3-3 Setting the title of the navigation bar in `init` 32

Listing 3-4 Setting the buttons of a navigation bar in `loadView` 32

Listing 3-5 Creating and pushing the next table-view controller on the stack 33

Creating and Configuring a Table View 35

Figure 4-1 Calling sequence for creating and configuring a table view 36

Figure 4-2 A table-view application project when just created 38

Figure 4-3 The contents of the main window's nib file 39

Figure 4-4 Setting the nib-name property of the table-view controller 40

Figure 4-5 Connections in the root view controller's nib file 40

Figure 4-6 Specifying a custom subclass of `UITableViewController` 41

Listing 4-1 The application delegate displaying the initial user interface 39

Listing 4-2 Adopting the data source and delegate protocols 43

Listing 4-3 Creating a table view 43

- Listing 4-4 Populating a table view with data 44
- Listing 4-5 Defining the model-object interface 46
- Listing 4-6 Loading the table-view data and initializing the model objects 47
- Listing 4-7 Preparing the data for the indexed list 48
- Listing 4-8 Providing section-index data to the table view 49
- Listing 4-9 Populating the rows of an indexed list 50
- Listing 4-10 Adding a title to the table view 51
- Listing 4-11 Returning a header title for a specific section 51
- Listing 4-12 Custom indentation of a row 52
- Listing 4-13 Varying row height 52

A Closer Look at Table-View Cells 54

- Figure 5-1 Parts of a table-view cell 54
- Figure 5-2 Parts of a table-view cell—editing mode 55
- Figure 5-3 Default cell content in a `UITableViewCell` object 56
- Figure 5-4 A table view with rows showing both images and text 57
- Figure 5-5 Cells with custom content as subviews 61
- Figure 5-6 Table-view rows drawn with a cell from a nib file 64
- Figure 5-7 Table view rows drawn with multiple cells from a nib file 68
- Figure 5-8 A custom table-view cell 71
- Listing 5-1 Configuring a `UITableViewCell` object with both image and text 57
- Listing 5-2 Alternating the background color of cells in
 `tableView:willDisplayCell:forRowAtIndexPath:` 59
- Listing 5-3 Adding subviews to a cell's content view 61
- Listing 5-4 Defining an outlet for the cell 65
- Listing 5-5 Loading a cell from a nib file and assigning it content 66
- Listing 5-6 Defining outlet properties for the cells in the nib file 68
- Listing 5-7 Passing nib-file cells to the table view 70
- Listing 5-8 Dynamically changing the content of a nib-file cell 71
- Listing 5-9 Declaring the properties and methods of the `TimeZoneCell` class 72
- Listing 5-10 Initializing an instance of `TimeZoneCell` 73
- Listing 5-11 Declaring the interface of the `TimeZoneView` class 74
- Listing 5-12 Setting the time-zone wrapper and related values 74
- Listing 5-13 Drawing the custom table-view cell 75
- Listing 5-14 Returning an initialized instance of the custom table-view cell 77

Managing Selections 78

- Listing 6-1 Responding to a row selection 79
- Listing 6-2 Setting a switch object as an accessory view and responding to its action message 79

- Listing 6-3 Managing a selection list—exclusive list 81
- Listing 6-4 Managing a selection list—inclusive list 81
- Listing 6-5 Programmatically selecting a row 82

Inserting and Deleting Rows and Sections 84

- Figure 7-1 Calling sequence for inserting or deleting rows in a table view 86
- Figure 7-2 Deletion of section and row and insertion of row 94
- Listing 7-1 View controller responding to `setEditing:animated:` 88
- Listing 7-2 Customizing the editing style of rows 88
- Listing 7-3 Updating the data-model array and deleting the row 89
- Listing 7-4 Adding an Add button to the navigation bar 89
- Listing 7-5 Responding to a tap on the Add button 90
- Listing 7-6 Adding the new item to the data-model array 90
- Listing 7-7 Batch insertion and deletion methods 91
- Listing 7-8 Inserting and deleting a block of rows in a table view 92

Managing the Reordering of Rows 95

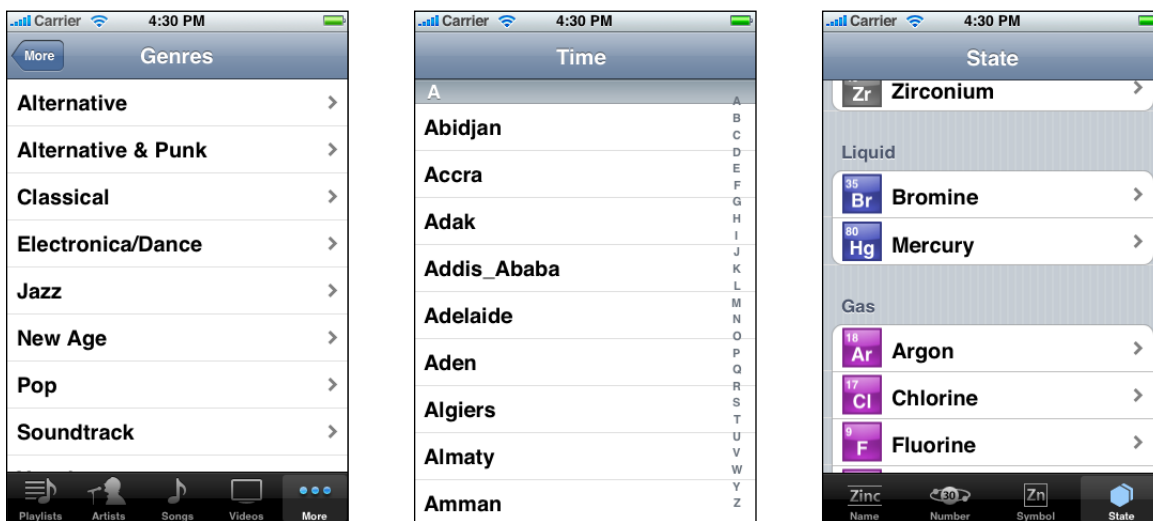
- Figure 8-1 Reordering a row 95
- Figure 8-2 Calling sequence for reordering a row in a table view 96
- Listing 8-1 Excluding a row from relocation 97
- Listing 8-2 Updating the data-model array for the relocated row 98
- Listing 8-3 Retargeting the destination row of a move operation 98

About Table Views in iOS-Based Applications

A table view presents a scrollable list of items that may be divided into sections. Table views are versatile user-interface objects frequently found in iOS-based applications, particularly productivity applications. They have many purposes:

- To let users navigate through hierarchically structured data
- To present an indexed list of items
- To display detail information and controls in visually distinct groupings
- To present a selectable list of options

Figure I-1 Table views of various kinds



A table view has only one column and allows vertical scrolling only. It consists of rows in sections. Each section can have a header and a footer that displays text or an image. However, many table views have only one section with no visible header or footer. Programmatically, UIKit identifies rows and sections through their index number: Sections are numbered 0 through $n-1$ from the top of a table view to the bottom; rows are numbered 0 through $n-1$ within a section. A table view can have its own header and footer, distinct from any section; the table header appears before the first row of the first section, and the table footer appears after the last row of the last section.

Programmatically, a table view is an instance of `UITableView` in one of two basic styles, plain or grouped. A plain-style table view is an unbroken list; a grouped table view has visually distinct sections. A table view has a data source and might have a delegate. These objects provide the data for populating the sections and rows of the table view and customize its appearance and behavior.

Related chapters [“Table View Styles and Accessory Views”](#) (page 11)

At a Glance

Table Views are Composed From Cells in a Particular Style

A table view draws its visible rows using cells—that is, `UITableViewCell` objects. Cells are views that can display text, images, or other kinds of content. They can have background views for both normal and selected states. Cells can also have accessory views, which function as controls for selection or setting an option.

The UIKit framework defines four standard cell styles, each with its own layout of the three default content elements: main label, detail label, and image. You may also create your own custom cells to acquire a distinctive style for your application’s table views.

Related Chapters [“Table View Styles and Accessory Views”](#) (page 11), [“A Closer Look at Table-View Cells”](#) (page 54)

There Are Several Ways to Create a Table View But the Basic Pattern is the Same

You create an instance of `UITableView` in one of the two styles and assign it a data source. Immediately after it’s created, the table view asks its data source for the number of sections, the number of rows in each section, and the table-view cell to use to draw each row. The data source manages the application data used for populating the sections and rows of the table view.

The easiest and recommended way to create a table view is to create a custom `UITableViewController` object. `UITableViewController` creates the table view and assigns itself as delegate and data source. If your application is largely based on table views, use the Navigation-based Application template when you create the application project; this template includes an initial custom `UITableViewController` class and a separate Interface Builder nib file for the table view.

You can create a table view with the help of Interface Builder or entirely in code. You can use a custom `UIViewController` object or even a non-controller object to manage a table view.

Related Chapters [“Navigating a Data Hierarchy With Table Views”](#) (page 23), [“Creating and Configuring a Table View”](#) (page 35)

Responding to Selections of Rows

When users select a row by tapping it, the delegate of the table view is informed via a message. The delegate is passed the indexes of the row and the section the row is in. It uses this information to locate the corresponding item in the application’s data model. This item might be an intermediate level in the hierarchy of data or a “leaf node” in the hierarchy. If the former, the application displays a new table view; if the latter, the application displays detail about the selected item in a grouped-style table view or some other kind of view.

In table views that list a series of options, tapping a row simply selects its associated option. No subsequent view of data is displayed.

Related Chapters [“Navigating a Data Hierarchy With Table Views”](#) (page 23), [“Managing Selections”](#) (page 78)

In Editing Mode You Can Add, Delete, and Reorder Rows

Table views can enter an editing mode wherein the user can insert or delete rows, or relocate them within the table. In editing mode, rows that are marked for insertion or deletion display a green plus sign (insertion) or a red minus sign (deletion) near the left edge of the row. If the user touches a deletion control or, in some table views, swipes across a row, a red Delete button appears to prompt the user for deletion of that row. Rows that can be relocated display near their right edge an image consisting of several horizontal lines. When the table view leaves editing mode, the insertion, deletion, and reordering controls disappear.

When users attempt to insert, delete, or reorder rows, the table view sends a sequence of messages to its data source and delegate to allow them to manage these operations.

Related Chapters [“Inserting and Deleting Rows and Sections”](#) (page 84), [“Managing the Reordering of Rows”](#) (page 95)

How to Use This Document

To get a concise overview of the programmatic interfaces involved in table-view creation and management, see [“Overview of the Table View API”](#) (page 21).

See Also

The information presented in this introduction and in [“Table View Styles and Accessory Views”](#) (page 11) summarizes prescriptive information on table views presented in *iOS Human Interface Guidelines*. It is recommended that you read this for a complete description of the styles and characteristics of table views, as well as their recommended uses.

Before reading this book, you should read *iOS App Programming Guide* to understand the basic process for developing iOS-based applications. You should also consider reading *View Controller Programming Guide for iOS* for general information about view controllers and for more detailed information about navigation controllers, which are frequently used in conjunction with table views.

You will find the following sample-code projects to be instructive models for your own table view implementations:

- *TableViewSuite* project
- *TheElements* project
- *SimpleDrillDown* project

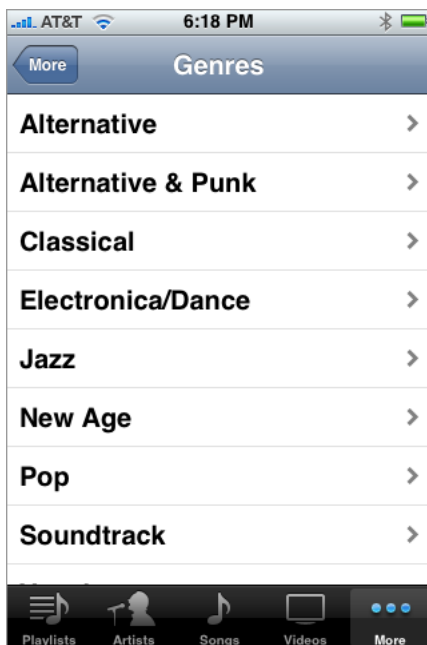
Table View Styles and Accessory Views

Table views come in distinctive styles that are suitable for specific purposes. In addition, UIKit provides standard styles for the cells used to draw the rows of table views. It also gives you standard accessory views (that is, controls) that you can include in cells.

Table View Styles

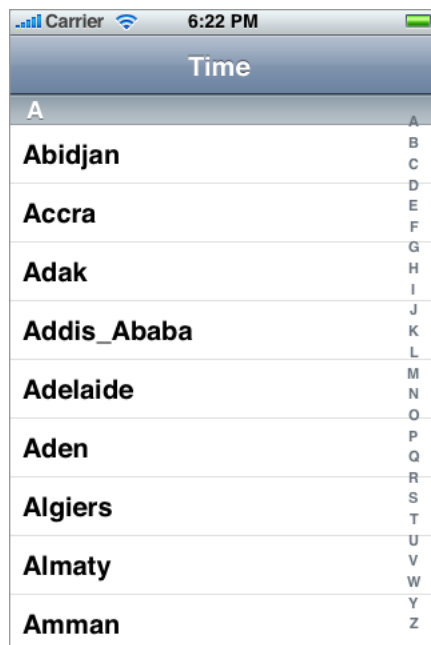
There are two major styles of table views: plain and grouped. A table view in the plain (or regular) style displays rows that stretch across the screen and have a white background (see Figure 1-1). The table view can have one or more sections, sections can have one or more rows, and each section can have its own header or footer title. (A header or footer may also have a custom view, for instance one containing an image). When the user scrolls through a section with many rows, the header of the section floats to the top of the table view and the footer of the section floats to the bottom.

Figure 1-1 A table view in the plain style (no section header or footer)



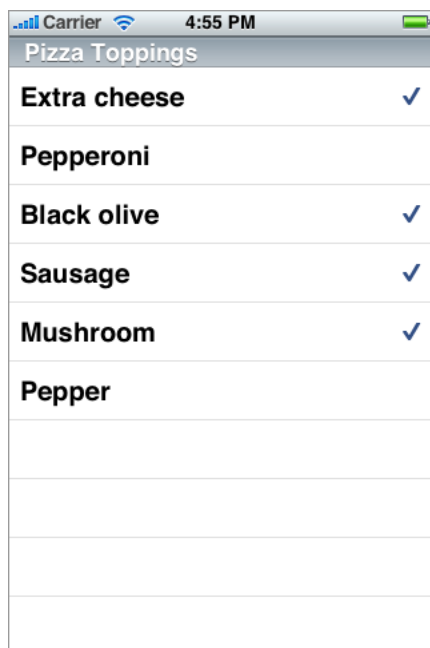
A variation of plain-style table views associates an index with sections for quick navigation; Figure 1-2 shows an example of this kind of table view, which is called an indexed list. The index runs down the right edge of the table view with entries in the index corresponding to section header titles. Touching an item in the index scrolls the table view to the associated section. For example, the section headings could be two-letter state abbreviations and the rows for a section could be the cities in that state; touching at a certain spot in the index displays the cities for the selected state. The rows in indexed section lists should not have disclosure indicators or detail disclosure buttons, because these interfere with the index.

Figure 1-2 A table view configured as an section index



The simplest kind of table view is a selection (or radio) list (see Figure 1-3). A selection list is a plain-style table view that presents a menu of options that users can select. It can limit the selection to one row or allow multiple selections. A selection list marks a selected row with a checkmark (see Figure 1-3).

Figure 1-3 A table view configured as a selection list



A grouped table view also displays a list of information, but it groups related rows in visually distinct sections. As shown in Figure 1-4, each section has rounded corners and by default appears against a bluish-gray background. Each section may have text or an image for its header or footer to provide some context or summary for the section. A grouped table works especially well for displaying the most detailed information

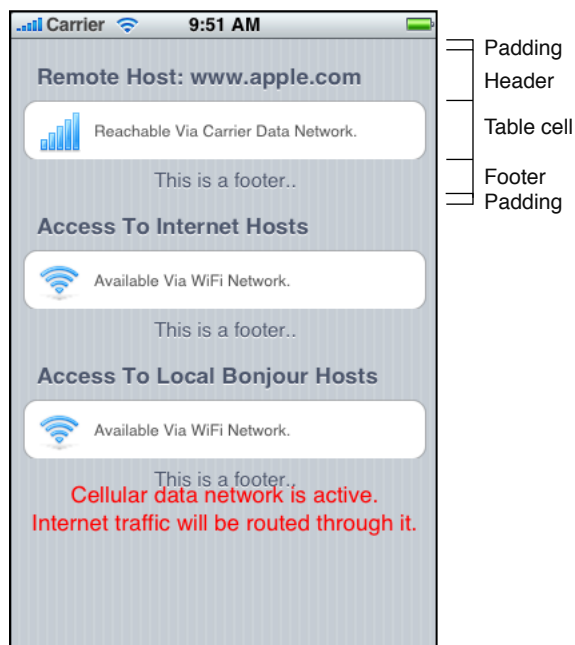
in a data hierarchy. It allows you to separate details into conceptual groups and provide contextual information to help users understand it quickly. For example, the information about a contact in the contacts list is grouped into phone information, email addresses, street addresses, and other sections.

Figure 1-4 A table view in the grouped style



The headers and footers of sections in a grouped table view have relative locations and sizes as indicated in Figure 1-5.

Figure 1-5 Header and footer of a section



On iPad devices, table views in the grouped style automatically get wider margins when the table views themselves are wide.

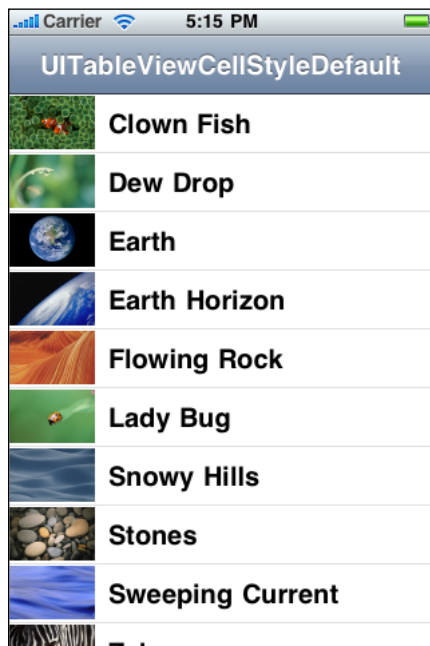
Standard Styles for Table-View Cells

In addition to the two styles of table views, UIKit defines four styles for the cells that a table view uses to draw its rows. You may create custom table-view cells with different appearances if you wish, but these four predefined cell styles are suitable for most purposes. The techniques for creating table-view cells in a predefined style and for creating custom cells are described in [“A Closer Look at Table-View Cells”](#) (page 54).

Note Table-view cell styles were introduced in iOS 3.0.

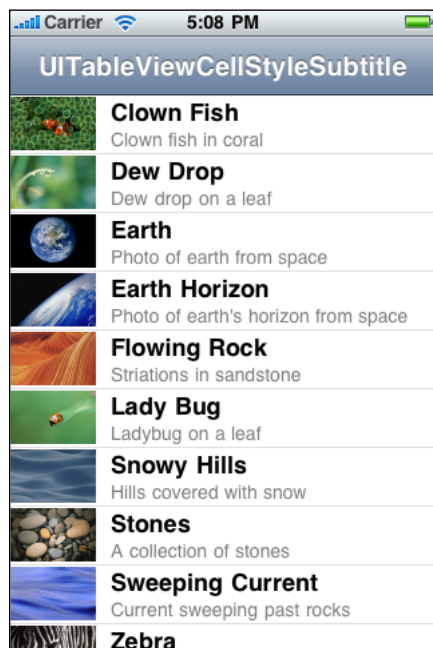
The default style for table-view rows uses a simple cell style that has a single title and permits an image (Figure 1-6). This style is associated with the `UITableViewCellStyleDefault` constant.

Figure 1-6 Default table row style (no subtitle)



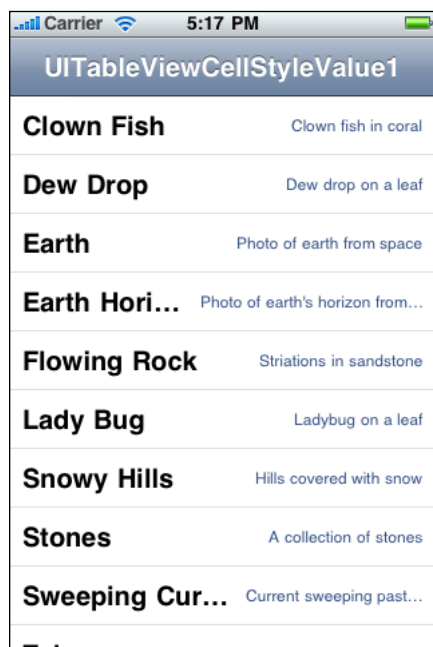
The cell style for the rows in Figure 1-7 left-aligns the main title and puts a gray subtitle right under it. It also permits an image in the default image location. This style is used in the iPod application and is associated with the `UITableViewCellStyleSubtitle` constant.

Figure 1-7 Table row style with subtitle under title



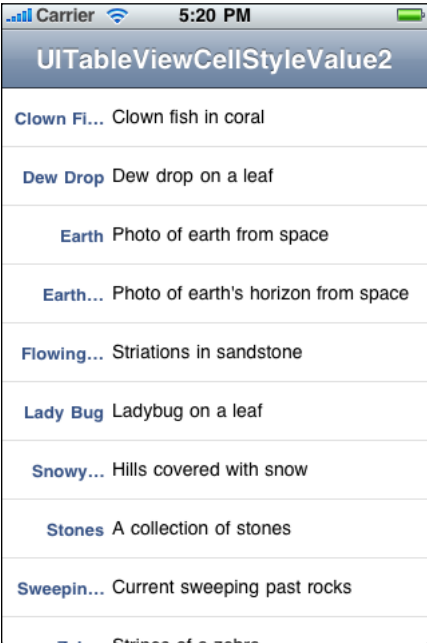
The cell style for the rows in Figure 1-8 left-aligns the main title and puts the subtitle in blue text and right-aligns it on the right side of the row. Images are not permitted. This style is used in the Settings application, where the subtitle indicates the current setting for a preference. It is associated with the `UITableViewCellStyleValue1` constant.

Figure 1-8 Table row style with right-aligned subtitle





The cell style for the rows in Figure 1-9 puts the main title in blue and right-aligns it at a point that’s indented from the left side of the row. The subtitle is left-aligned at a short distance to the right of this point. This style does not allow images. It is used in the Contacts part of the Phone application and is associated with the `UITableViewCellStyleValue2` constant.


Figure 1-9 Table row style in Contacts format



Accessory Views

There are three standard kinds of accessory views (shown with their accessory-type constant):

Standard accessory views	Description
	Disclosure indicator — <code>UITableViewCellAccessoryDisclosureIndicator</code> . You use the disclosure indicator when selecting a cell results in the display of another table view reflecting the next level in the data-model hierarchy.
	Detail disclosure button — <code>UITableViewCellAccessoryDetailDisclosureButton</code> . You use the detail disclosure button when selecting a cell results in a detail view of that item (which may or may not be a table view).

Standard accessory views	Description
	Check mark — <code>UITableViewCellAccessoryCheckmark</code> . You use a checkmark when a touch on a row results in the selection of that item. This kind of table view is known as a selection list, and it is analogous to a pop-up list. Selection lists can limit selections to one row, or they can allow multiple rows with checkmarks.

Instead of the standard accessory-view types, you may specify a control object (for example, a switch) or a custom view as the accessory view.

Overview of the Table View API

The programmatic interface for table views is expressed through several classes, two formal protocols, and a category added to a Foundation framework class. You have to deal with all of these API components when implementing a table view.

Table View

A table view itself is an instance of the `UITableView` class. This class declares methods that allow you to configure the appearance of the table view—for example, specifying the default height of rows or providing a view used as the header for the table. Other methods give you access to the currently selected row as well as specific rows or cells. You can call other methods of `UITableView` to manage selections, scroll the table view, and insert or delete rows and sections.

`UITableView` inherits from `UIScrollView`, which defines scrolling behavior for views with content larger than the size of the window. `UITableView` redefines the scrolling behavior to allow vertical scrolling only.

Data Source and Delegate

A `UITableView` object must have a delegate and a data source. Following the Model-View-Controller design pattern, the data source mediates between the application's data model (that is, its model objects) and the table view; the delegate, on the other hand, manages the appearance and behavior of the table view. The data source and the delegate are often (but not necessarily) the same object, and that object is frequently a custom subclass of `UITableViewController`. (See [“Navigating a Data Hierarchy With Table Views”](#) (page 23) for further information.)

The data source adopts the `UITableViewDataSource` protocol. `UITableViewDataSource` has two required methods. `tableView:numberOfRowsInSection:` tells the table view how many rows to display in each section, and `tableView:cellForRowAtIndexPath:` provides the cell to display for each row in the table. Optional methods allow the data source to configure multiple sections, provide headers and/or footers, and support adding, removing, and reordering rows in the table.

The delegate adopts the `UITableViewDelegate` protocol. `UITableViewDelegate` has no required methods. It declares methods that allow the delegate to modify visible aspects of the table view, manage selections, support an accessory view, and support editing of individual rows in the table.

An application can make use of a convenience class, `UILocalizedIndexedCollation`, to help the data source organize the data for indexed lists and display the proper section when users tap an item in the index. The `UILocalizedIndexedCollation` class also localizes section titles.

Extension to the NSIndexPath Class

Many methods of `UITableView`, `UITableViewDataSource`, and `UITableViewDelegate` have as parameters or return value objects representing index paths. An index path identifies a path to a specific node in a tree of nested arrays, and in the Foundation framework it is represented by an `NSIndexPath` object. UIKit declares a category on `NSIndexPath` (see *NSIndexPath UIKit Additions*) with methods that return key paths, locate rows in sections, and construct `NSIndexPath` objects from row and section indexes.

Table View Cells

As noted in [“Data Source and Delegate”](#) (page 21), the data source must return a cell object for each visible row that a table view displays. These cell objects must inherit from the `UITableViewCell` class. This class includes methods that let you manage cell selection and editing, manage accessory views, and configure the cell. You can directly instantiate cells in the standard styles defined by the `UITableViewCell` class and give these cells content consisting of one or two strings of text and, in some styles, both image and text as content. Or, instead of using a cell in a standard style, you can put your own custom subviews in the content view of an “off the shelf” cell object. You may also subclass `UITableViewCell` to radically customize the appearance and behavior of table-view cells. [“A Closer Look at Table-View Cells”](#) (page 54) discusses all of these approaches.

Navigating a Data Hierarchy With Table Views

A common use of table views—and one to which they’re ideally suited—is to navigate hierarchies of data. A table view at a top level of the hierarchy lists categories of data at the most general level. Users select a row to “drill down” to the next level in the hierarchy. At the bottom of the hierarchy is a view (often a table view) that presents details about a specific item (for example, an address-book record) and may allow users to edit the item. This section explains how you can map the levels of the data-model hierarchy to a succession of table views and describes how you can use the facilities of the UIKit framework to help you implement such navigation-based applications.

Hierarchical Data Models and Table Views

For a navigation-based application, you typically design your application data as a graph of model objects that is sometimes referred to as the application’s data model. You can then implement the model layer of your application using various mechanisms or technologies, including Core Data, property lists, or archives of custom objects. Regardless of the approach, the traversal of your application’s data model follows patterns that are common to all navigation-based applications. The data model has hierarchical depth, and objects at various levels of this hierarchy should be the source for populating the rows of a table view.

Note Core Data was introduced as a supported technology in iOS 3.0. To learn about the Core Data technology and framework, see *Core Data Starting Point*.

The Data Model as a Hierarchy of Model Objects

A well-designed application factors its classes and objects in a way that conforms to the Model-View-Controller (MVC) design pattern. The application’s data model consists of the model objects in this pattern. You can describe model objects (using the terminology provided by the object modeling pattern) in terms of their properties. These properties are of two general kinds: attributes and relationships.

Note The notion of “property” here is abstractly related to, but not identical with, the declared property feature of Objective-C. A class definition typically represents properties programmatically through instance variables and declared properties. For more on declared properties, see *The Objective-C Programming Language*. To find out more about MVC and object modeling, read “Cocoa Design Patterns” in *Cocoa Fundamentals Guide*.

Attributes represent elements of model-object data. Attributes can range from an instance of a primitive class (for example, an `NSString`, `NSDate`, or `UIColor` object) to a C structure or a simple scalar value. Attributes are generally what you use to populate a table view that represents a “leaf node” of the data hierarchy and presents a detail view of that item.

A model object may also have relationships with other model objects, and it is through these relationships that a data model acquires hierarchical depth by composing an object graph. Relationships are of two general kinds in terms of cardinality: to-one and to-many. To-one relationships define an object’s relationship with another object (for example, a parent relationship). A to-many relationship on the other hand defines an object’s relationship with multiple objects of the same kind. The to-many relationship is characterized by containment and can be programmatically represented by collections such as `NSArray` objects (or, simply, arrays). An array might contain other arrays, or it could contain multiple dictionaries, which are collections that identify their contained values through keys. Dictionaries, in turn, can contain one or more other collections, including arrays, sets, and even other dictionaries. By collections thus nesting other collections, your data model can acquire hierarchical depth.

Table Views and the Data Model

The rows of a table view in the regular (or plain) style are typically backed by collection objects of the application’s data model, and these objects are usually arrays. The array contains strings or other elements that the table view can use when displaying row content. When you create a table view (described in [“Creating and Configuring a Table View”](#) (page 35)), the table view immediately queries its data source for its dimensions—that is, the number of sections and the number of rows per section—and then asks for the content of each row. This content is fetched from an array in the appropriate level of the data-model hierarchy.

In many of the methods defined for a table view’s data source and delegate, the table view passes in an index path to identify the section and row that is the focus of the current operation—for example, fetching content for a row or indicating the row the user tapped. An index path is an instance of the Foundation framework’s `NSIndexPath` class that you can use to identify an item in a tree of nested arrays. The UIKit framework extends `NSIndexPath` to add a `section` and a `row` property to the class. The data source should use these properties to map a section and row of the table view to a value at the corresponding index of the array used as the source of data for the table view.

Note The UIKit framework extension of the `NSIndexPath` class is described in *NSIndexPath UIKit Additions*.

Consider the sequence of table views in Figure 3-1. The top level of the data hierarchy in this example is an array of four arrays, with each inner array containing objects representing the trails for a particular region. When the user selects one of these regions, the next table view lists names identifying the trails within the selected array. When the user selects a particular trail, the next table view displays detail about that trail in a table view in the grouped style.

Figure 3-1 Mapping levels of the data model to table views



For the purpose of illustration, the above diagram shows a sequence of three table views navigating three levels of a data hierarchy. You could easily redesign this application so that there are only two table views. The first table view could be a list (plain style) in which each region is a section of the table view whose rows name the trails for that region. The data model could reflect this arrangement as an array nesting multiple arrays. When the table view asks its data source for the content for a particular row, it passes an `NSIndexPath` object that the data source can use to locate, first, the inner array (`section` property) and then the object within that array (`row` property).

View Controllers and Navigation-Based Applications

The UIKit framework provides a number of view-controller classes for managing common user-interface patterns in iOS. View controllers are controller objects that inherit from the `UIViewController` class. They are an essential tool for view management, especially when an application uses those views to present successive levels of its data hierarchy. For managing table views, the UIKit provides the `UITableViewController` class, a subclass of `UIViewController` that is described in “[Table-View Controllers](#)” (page 28). This section explains what view controllers are and describes how they present and manage a succession of views, including table views.

Overview of View Controllers and Navigation Controllers

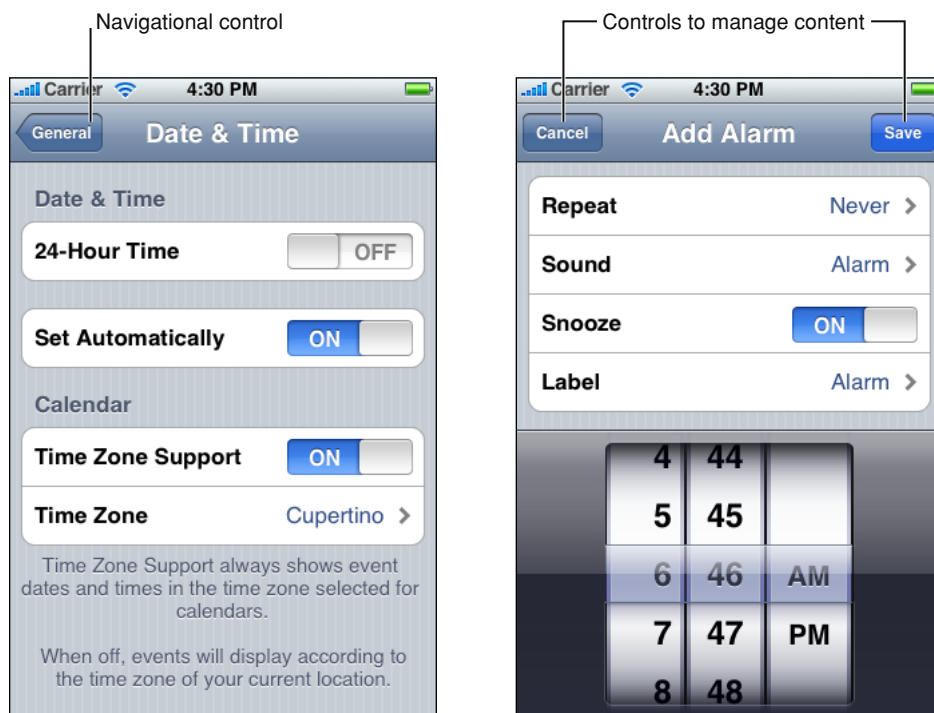
Navigation bars are a user-interface device that enables users to navigate a hierarchy of data. Users start with general, top-level items and “drill down” the hierarchy to detail views showing specific properties of leaf-node items. The view below the navigation bar presents the current level of data. A navigation bar includes a title for the current view and, if that view is lower in the hierarchy than the top level, a back button on the left side of the bar; the back button is a navigation control that the user taps to return to the previous level. (The back button by default displays the title for the previous view.) A navigation bar may also have an Edit button—used to enter editing mode for the current view—or custom buttons for functions that manage content (see [Figure 3-2](#) (page 27)).

Note This section gives an overview of view controllers and navigation controllers to provide some background for the coding tasks discussed throughout this document. To learn about view controllers and navigation controllers in depth, see *View Controller Programming Guide for iOS*.

Navigation bars are `UINavigationController` objects. Their per-managed-view components, such as the bar title, back button, and any Edit buttons or custom button, are instances of the `UINavigationControllerItem` class. If those components are buttons, they take `UIBarButtonItem` objects as their values. If you use a navigation

controller—an instance of `UINavigationController`—to manage the navigation items and the presented view, you never have to deal directly with either `UINavigationController` or `UINavigationControllerItem` except, in the latter case, to set a navigation item's value.

Figure 3-2 Navigation bars and common control items



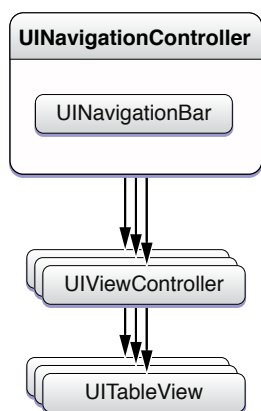
The `UINavigationController` class inherits from `UIViewController`, a base class that defines the common programmatic interface and behavior for controller objects that manage views in iOS. Through inheritance from this base class, a view controller acquires an interface for general view management. Once it implements parts of this interface, a view controller can autorotate its view, respond to low-memory notifications, overlay “modal” views, respond to taps on the Edit button, and otherwise manage the view.

A `UINavigationController` manages the navigation bar, including the items that are displayed in the bar for the view below it. A `UIViewController` object manages a view displayed below the navigation bar. For this view controller, you create a subclass of `UIViewController` or a subclass of a view-controller class that the UIKit framework provides for managing a particular type of view. For table views, this view-controller class is `UITableViewController`. For a navigation controller that displays a sequence of table views reflecting levels within a data hierarchy, you need to create a separate custom table-view controller for each table view.

A navigation controller manages a navigation bar that traverses a sequence of table views by maintaining a stack of view controllers, one for each of the table views displayed (see Figure 3-3). It begins with what's known as the root view controller. When the user taps a row of the table view (often on a disclosure indicator or a detail disclosure button), the root view controller pushes the next view controller onto the stack; the new view

controller's table view visually slides into place from the right and the navigation bar items are updated appropriately. When users tap the back button in the navigation bar, the current view controller is popped off the stack. As a consequence, the navigation controller displays the table view managed by the view controller now at the top of the stack.

Figure 3-3 Navigation controller and view controllers in a navigation-based application



The `UIViewController` includes methods that allow view controllers to access and set the navigation items (which are `UINavigationControllerItem` objects) displayed in the navigation bar for the currently displayed table view. It also declares a `title` property through which you can set the title of the navigation bar for the current table view.

Not all applications need to use the navigation controller and view controller architectures. If your application is displaying a single table view and not a sequence of them, you don't need to use navigation controllers unless you want to have an Edit button or other specialized buttons appear in the navigation bar. Also, if an application shows only one table view, it doesn't even need a custom view controller to manage it. For example, an application that uses a table view to present a list of selectable options can have an object other than a view controller assume the roles of data source and delegate for the table view.

Table-View Controllers

Although you could manage a table view within the navigation-controller architecture using a direct subclass of `UIViewController`, you save yourself a lot of work if instead you subclass `UITableViewController`. The `UITableViewController` class takes care of many of the details you would have to implement if you were to create a direct subclass of `UIViewController` to manage a table view.

You create a table-view controller by allocating memory for it and “Initialization” it with the `initWithStyle:` method, passing in either `UITableViewStylePlain` or `UITableViewStyleGrouped` for the required type of table view. Once you create a table-view controller, it either creates its table view or loads it from a nib file. In either case, the behavior is slightly different:

- If a nib file is specified, the `UITableViewController` object loads the `UITableView` object archived in the nib file. (The nib file is usually specified as attribute of the `UITableViewController`, which must be the File's Owner of the nib file.) The table view's attributes, size, and autosizing characteristics are usually set in the nib file. The data source and delegate of the table view become those objects defined in the nib file, if any.
- If there is no nib file, the `UITableViewController` object allocates and initializes an unconfigured `UITableView` object with the correct dimensions and autoresize mask. It sets itself as the data source and the delegate of the table view; it also does this if the nib file defines no data source or delegate.

When the table view is about to appear for the first time, the table-view controller sends `reloadData` to the table view, which prompts it to request data from its data source. The data source tells the table view how many sections and rows-per-section it wants, then gives the table view the data to display in each row. This process is described in [“Creating and Configuring a Table View”](#) (page 35).

The `UITableViewController` class also performs other common tasks. It clears selections when the table view is about to be displayed and flashes the scroll indicators when the table finishes displaying. In addition, it responds properly when users tap the Edit button by putting the table view into editing mode (or taking it out of editing mode if users tap Done). The class exposes one property, `tableView`, which gives you access to the managed table view.

Note `UITableViewController` has new capabilities in iOS 3.0. A table-view controller supports inline editing of table-view rows; if, for example, rows have embedded text fields in editing mode, it scrolls the row being edited above the virtual keyboard that is displayed. In addition, it now supports the `NSFetchedResultsController` class for managing the results returned from a Core Data fetch request.

The `UITableViewController` class implements the foregoing behavior by overriding `loadView`, `viewWillAppear:`, and other methods inherited from `UIViewController`. In your subclass of `UITableViewController`, you may also override these methods to acquire specialized behavior. If you do override these methods, be sure to invoke the superclass implementation of the method, usually as the first method call, to get the default behavior.

Note You should use a `UIViewController` subclass rather than a subclass of `UITableViewController` to manage a table view if the view to be managed is composed of multiple subviews, one of which is a table view. The default behavior of the `UITableViewController` class is to make the table view fill the screen between the navigation bar and the tab bar (if either are present).

If you decide to use a `UIViewController` subclass rather than a subclass of `UITableViewController` to manage a table view, you should perform a couple of the tasks mentioned above to conform to the human-interface guidelines. To clear any selection in the table view before it's displayed, implement the `viewWillAppear:` method to clear the selected row (if any) by calling `deselectRowAtIndexPath:animated:`. After the table view has been displayed, you should flash the scroll view's scroll indicators by sending a `flashScrollIndicators` message to the table view; you can do this in an override of the `viewDidAppear:` method of `UIViewController`.

Managing Table Views In a Navigation-Based Application

A `UITableViewController` object—or any other object that assumes the roles of data source and delegate for a table view—must respond to messages sent by the table view in order to populate its rows, configure it, respond to selections, and manage editing sessions. The subsequent chapters in this document describe how to do these things. However, there are certain other things you need to do to ensure the proper display of a sequence of table views in a navigation-based application.

Note This section summarizes view-controller and navigation-controller tasks with a focus on table views. For a thorough discussion of view controllers and navigation controllers, including the complete details of their implementation, see *View Controller Programming Guide for iOS*.

The sequence typically starts with the application delegate in its implementation of the `applicationDidFinishLaunching:` (or `application:didFinishLaunchingWithOptions:`) method. The delegate creates an instance of the `UITableViewController` subclass that is to be the root view controller. Then it allocates an instance of `UINavigationController` and initializes it with the just-created table-view controller with the `initWithRootViewController:` method. This initializer has the side effect of making the root view controller the first object on the stack of view controllers managed by the navigation controller. After creating the navigation controller, the delegate adds the navigation controller's view to the window and makes the window visible. Listing 3-1 illustrates this series of calls.

Listing 3-1 Setting up the root view controller—window in nib file

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
```

```
RootViewController *rootViewController = [[RootViewController alloc]
initWithStyle:UITableViewStylePlain];

NSArray *timeZones = [NSTimeZone knownTimeZoneNames];

rootViewController.timeZoneNames = [timeZones
sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];

UINavigationController *aNavigationController = [[UINavigationController alloc]
initWithRootViewController:rootViewController];

self.navigationController = aNavigationController;
[aNavigationController release];
[rootViewController release];

[window addSubview:[navigationController view]];
[window makeKeyAndVisible];
}
```

The window in this example is unarchived from a nib file and assigned to an outlet of the application delegate. If you were to create the window programmatically, the code would look similar to Listing 3-2. (In this case, `window` is a property of the application delegate.)

Listing 3-2 Setting up the root view controller—window created programmatically

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Create the window
    window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];

    // Create the navigation and view controllers
    RootViewController *rootViewController = [[RootViewController alloc]
initWithStyle:UITableViewStylePlain];
    navigationController = [[UINavigationController alloc]
initWithRootViewController:rootViewController];
    [rootViewController release];

    // Configure and show the window
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

```
}
```

When you initialize a table-view controller with `initWithStyle:`, that “Initialization” is invoked as well as the initializers inherited from superclasses:

- `initWithNibName:bundle:`—inherited from `UIViewController`

If your table view is defined in a nib file, instead of setting the Nib Name attribute of the table-view controller (as File’s Owner) in Interface Builder, you can override this method to call `super`, supplying the superclass with the name of the nib file.

- `init`—inherited from `NSObject`

You can override any of these initializers to perform set-up tasks common to view controllers—for example, setting the title of the navigation bar, as shown in [Listing 3-3](#) (page 32).

Listing 3-3 Setting the title of the navigation bar in `init`

```
- init {
    if (self = [super init]) {
        self.title = NSLocalizedString(@"List", @"List title");
    }
    return self;
}
```

You can perform the same set-up tasks in the `loadView` and `viewDidLoad` methods, which are invoked right after initialization. You can also in these methods set various `UINavigationController` properties through the inherited `navigationItem` property. These properties generally are button items. Listing 3-4 shows how one table-view controller class places an Edit|Done and an add (“+”) button on its navigation bar.

Listing 3-4 Setting the buttons of a navigation bar in `loadView`

```
- (void)loadView {
    [super viewDidLoad];
    self.tableView.allowsSelectionDuringEditing = YES;
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButtonItem = [[UIBarButtonItem alloc]
```



```
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self  
action:@selector(addItem:));  
self.navigationItem.rightBarButtonItem = addButtonItem;  
[addButtonItem release];  
}
```

At this point, let's assume the table view managed by our root table-view controller is presented to users. It is a list (or regular-style) table view. How does the application display the next table view in the sequence?

When a user taps a row of the table view, the table view calls the `tableView:didSelectRowAtIndexPath:` or `tableView:accessoryButtonTappedForRowWithIndexPath:` method implemented by the delegate. (That latter method is invoked if the user taps a row's detail-disclosure indicator.) The delegate in the example shown in Listing 3-5 creates the table-view controller managing the next table view in the sequence, sets the data it needs to populate its table view, and pushes this new view controller onto the navigation controller's stack of view controllers.

Listing 3-5 Creating and pushing the next table-view controller on the stack

```
- (void)tableView:(UITableView *)tableView  
accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {  
    BATDetailViewController *trailDetailController = [[BATDetailViewController  
alloc] initWithStyle:UITableViewStyleGrouped];  
    trailDetailController.curTrail = [trails objectAtIndex:indexPath.row];  
    [[self navigationController] pushViewController:trailDetailController  
animated:YES];  
    [trailDetailController release];  
}
```

This last code example is provided to give you a general idea of how a navigation-based application displays a sequence of table views. For complete details about handling selections in table views, see [“Managing Selections”](#) (page 78).

Design Pattern for Navigation-Based Applications

A good design for a navigation-based application with table views should be consistent with the following pattern:

- A view controller object (typically a `UITableViewController` object), acting in the role of data source, populates its table view with data from an object representing a level of the data hierarchy.

The object is typically an array when the table view displays a list of items; when the table view displays item detail (that is, a leaf node of the data hierarchy), the object can be a custom model object, a Core Data managed object, a dictionary, or something similar.

- The view controller stores the data it needs for populating its table view.

The view controller can use this data directly for populating the table view, or it can use it to fetch or otherwise obtain the necessary data. When you design your view-controller subclass, you should define an instance variable (whose access is mediated via a declared property or accessor methods) to hold this data.

View controllers should *not* obtain the data for their table view through a global variable or a singleton object such as the application delegate. Such direct dependencies make your code less reusable and more difficult to test and debug.

- The current view controller on top of the navigation-controller stack creates the next view controller in the sequence and, before it pushes it onto the stack, sets the data this view controller, acting as data source, needs to populate its table view.

Creating and Configuring a Table View

Your application must present a table view to users before it can manage it in response to taps on rows and other actions. This chapter shows what you must do to create a table view, configure it, and populate it with data.

Most of the code examples shown in this chapter come from the example projects *TableViewSuite* and *TheElements*.

Basics of Table View Creation

The creation of a table view requires the interaction of several entities in an application: the client, the table view itself, and the table view's data source and delegate. The client, delegate, and data source are often the same object, but can be separate objects. The client starts the calling sequence, diagrammed in [Figure 4-1](#) (page 36).

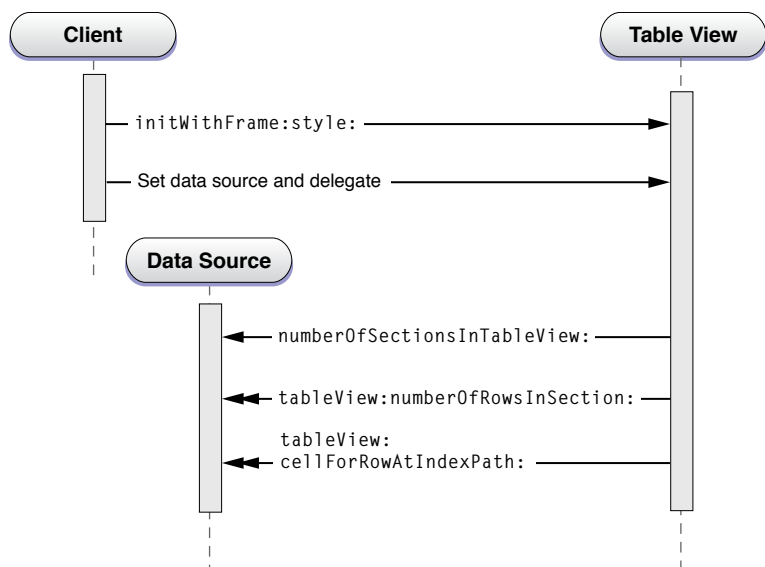
1. The client creates a `UITableView` instance in a certain frame and style. It can do this either programmatically or in Interface Builder. The frame is usually set to the screen frame, minus the height of the status bar or, in a navigation-based application, to the screen frame minus the heights of the status bar and the navigation bar. The client may also set global properties of the table view at this point, such as its autosizing behavior or a global row height.

See [“Creating a Table View Application the Easy Way”](#) (page 37) and [“Creating a Table View Programmatically”](#) (page 42) for information on creating table views using Interface Builder and programmatically, respectively.

2. The client sets the data source and delegate of the table view and sends `reloadData` to it. The data source must adopt the `UITableViewDataSource` protocol and the delegate must adopt the `UITableViewDelegate` protocol.
3. The data source receives a `numberOfSectionsInTableView:` message from the `UITableView` object and returns the number of sections in the table view. Although this is an optional protocol method, the data source must implement it if the table view has more than one section.
4. For each section, the data source receives a `tableView:numberOfRowsInSection:` message and responds by returning the number of rows for the section.

5. The data source receives a `tableView:cellForRowAtIndexPath:` message for each visible row in the table view. It responds by configuring and returning a `UITableViewCell` object for each row. The `UITableView` object uses this cell to draw the row.

Figure 4-1 Calling sequence for creating and configuring a table view



The diagram in Figure 4-1 shows the required protocol methods as well as the `numberOfSectionsInTableView:` method. (Note that zero is a valid value to return for number of sections.) Populating the table view with data occurs in steps 3 through 5; [“Populating the Table View With Data”](#) (page 44) describes how you implement the methods mentioned in these steps.

The data source and the delegate may implement other optional methods of their protocols to further configure the table view. For example, the data source might want to provide titles for each of the sections in the table view by implementing `tableView:titleForHeaderInSection:`. [“Optional Table-View Configurations”](#) (page 51) describes some of these optional table-view customizations.

You create a table view in either the plain style (`UITableViewStylePlain`) or the grouped style (`UITableViewStyleGrouped`). (You specify the style when you initialize the table view by calling, directly or indirectly, the `initWithFrame:style:` method. Although, the procedure for creating a table view in either styles is identical, you may want to perform different kinds of configurations. For example, because a grouped table view generally presents item detail, you may also want to add custom accessory views (for example, switches and sliders) or custom content (for example, text fields) to cells in the delegate’s `tableView:cellForRowAtIndexPath:` method; [“A Closer Look at Table-View Cells”](#) (page 54) gives an example of this.

Recommendations for Creating and Configuring Table Views

There are many ways to put together a table-view application. For example, you could use an instance of a custom `NSObject` subclass to create, configure, and manage a table view. However, you will find the task much easier if you adopt the classes, techniques, and design patterns that the UIKit offers for this purpose. The following approaches are recommended:

- Use an instance of a subclass of `UITableViewController` to create and manage a table view.

Most applications use a custom `UITableViewController` object to manage a table view. As described in [“Navigating a Data Hierarchy With Table Views”](#) (page 23), `UITableViewController` automatically creates a table view, assigns itself as both delegate and data source (and adopts the corresponding protocols), and initiates the procedure for populating the table view with data. It also takes care of several other “housekeeping” details of behavior. The behavior of `UITableViewController` (a subclass of `UIViewController`) within the navigation-controller architecture is described in [“Table-View Controllers”](#) (page 28).

- If your application is largely based on table views (a “list” application), select the Navigation-based Application template defined by Xcode when you create your project.

As described in [“Creating a Table View Application the Easy Way”](#) (page 37), the template includes stub code and nib files defining an application delegate, the navigation controller, and the root view controller (which is an instance of a custom subclass of `UITableViewController`).

- For successive table views, you should implement custom `UITableViewController` objects. You can either create the associated table views programmatically or you can load them from separate nib files.

Although either option is possible, you may find the purely programmatic route easier. These options are discussed in [“Adding Table Views to the Application”](#) (page 41).

If the view to be managed is a composite view in which a table view is one of multiple subviews, you must use a custom subclass of `UIViewController` to manage the table view (and other views). Do not use a `UITableViewController` object because this controller class sizes the table view to fill the screen between the navigation bar and the tab bar (if either are present).

Creating a Table View Application the Easy Way

Creating a table-view application in Xcode is very simple. When you create your project you select a template with stub code and nib files that supply the structure for setting up and managing table views. To create an application that is structured around table views, complete the following steps:

1. In Xcode, choose New Project from the File menu.
2. Select the Navigation-based Application template project and click Choose.

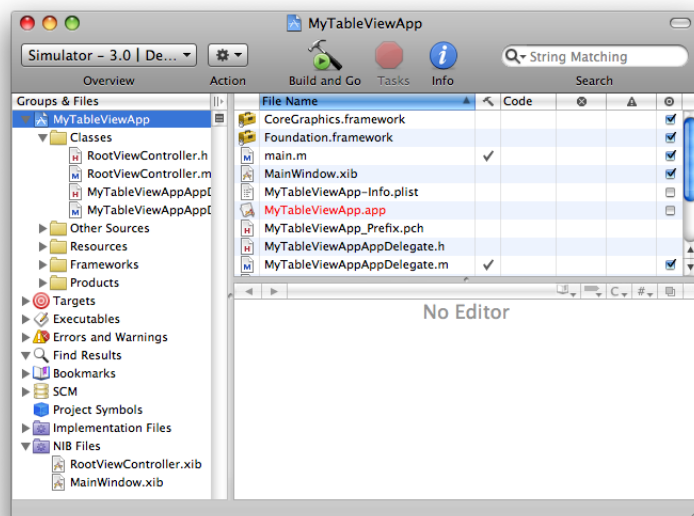
3. Specify a name and location for the project and click Save.

Xcode creates the table view in a nib file and creates an instance of a custom subclass of `UITableViewController` that, at runtime, loads the table view from the nib file, populates it, and manages it.

Examining the Project: How Things are Set Up

Before you begin any coding or nib work, it's useful to examine the key components of the project you just created: the application-delegate and root view-controller classes and the nib files `MainWindow.xib` and `RootViewController.xib`. Figure 4-2 shows where these items are located in the Groups & Files view of the Xcode project window.

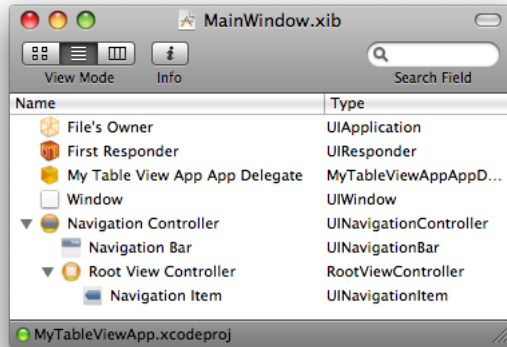
Figure 4-2 A table-view application project when just created



Double-click `MainWindow.xib` to open this nib file in Interface Builder. In the document window for this nib file you can see the objects it contains (shown in Figure 4-3). The main top-level objects are File's Owner (a proxy for the application object itself), the application delegate, the application's window, and a `UINavigationController` object. The last object is the root of an object graph that is significant for the table view. The direct children of the navigation controller are the navigation bar that runs across the screen above the table view and a placeholder for an instance of the `RootViewController` class of the project.

Because it is a child of the navigation controller, this view controller is made the root view controller, the first one on the stack of view controllers managed by the navigation controller (explained in [“Navigating a Data Hierarchy With Table Views”](#) (page 23)).

Figure 4-3 The contents of the main window’s nib file



When the application is launched, the `MainWindow.xib` nib file is loaded into memory. The application delegate displays the initial user interface in the two lines of code in Listing 4-1. By asking the navigation controller for its view, the application delegate obtains the navigation bar, the title in the navigation bar (the navigation item), and the table view associated with the root view controller.

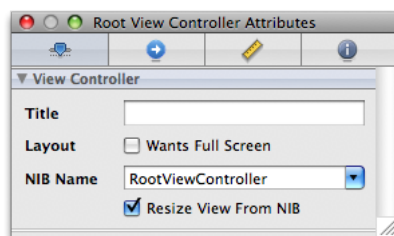
Listing 4-1 The application delegate displaying the initial user interface

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {  
  
    [window addSubview:[navigationController view]];  
    [window makeKeyAndVisible];  
}
```

But (you may have noticed) the Root View Controller object in the nib document window in [Figure 4-3](#) (page 39) does not show a subordinate table view. Where does the table view come from?

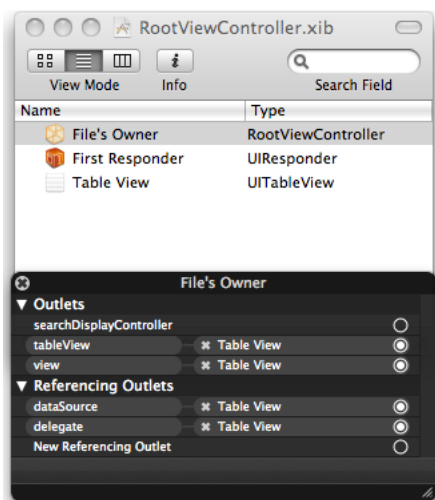
Select the Root View Controller object in the document window and display the attributes inspector for that object (Command-1). Here you see that the Nib Name field is set to RootViewController (as shown in Figure 4-4). When the delegate asks the navigation controller for its views, the navigation controller asks its root view controller for its view, and the root view controller loads the nib file specified through this attribute.

Figure 4-4 Setting the nib-name property of the table-view controller



The `RootViewController.xib` nib file contains the table view and sets the `RootViewController` object to be File's Owner. To see the connections between `RootViewController` and its table view, right-click (or Control-click) File's Owner. This action displays a heads-up display as depicted in Figure 4-5.

Figure 4-5 Connections in the root view controller's nib file



The `RootViewController` object keeps two references to the table view (one is via the `view` property inherited from `UIViewController`); it is also set to be the data source and delegate of the table view.

You can also change the characteristics of the table view by selecting it in the nib document window and then going to the Attributes pane of the Interface Builder inspector. For example, you could change the style of the table view from plain (the default) to grouped.

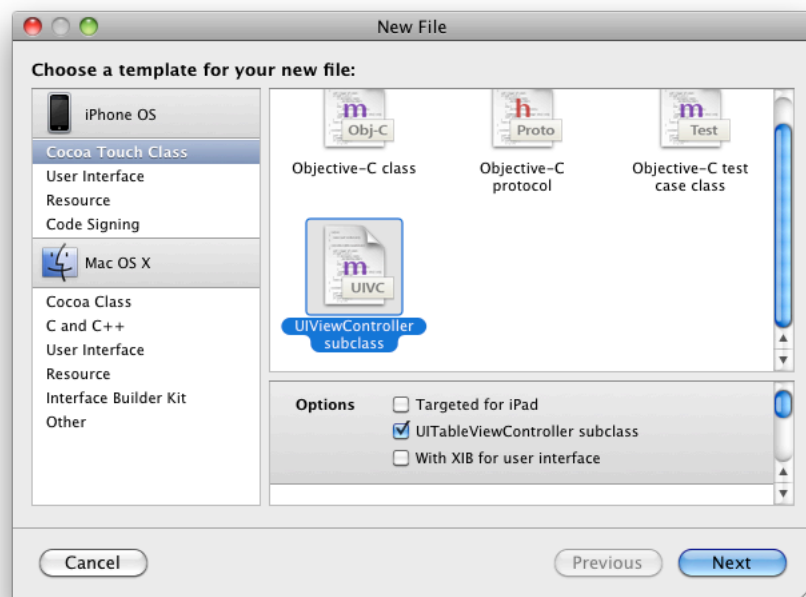
The table view managed by `RootViewController` is empty at runtime until the the table-view controller populates it with data; see [“Populating the Table View With Data”](#) (page 44) to learn the procedure for doing this. You can also programmatically configure the table view or the navigation bar, such as specifying the title of the navigation bar or the title of the table view. (Common sites for this configuration code are the `initWithStyle:` or `viewDidLoad` methods.) [“Optional Table-View Configurations”](#) (page 51) discusses some of the things you can do to customize the appearance or behavior of a table view.

Adding Table Views to the Application

If your application displays and manages more than one table view, you need to add those table views to your project. You typically add a table view by adding a `UITableViewController` object, which creates the table view it manages. After adding a custom table-view controller class to your project, you can configure the table view programmatically; or you can configure it and any related views in a nib file that is loaded at runtime. For situations where a table view is the sole view, the programmatic approach is easier. The nib-file approach is appropriate when the managed view is more complex—for example, a view with a table view as only one of its subviews.

To add a custom table-view controller class, open your Xcode project and choose New File from the File menu. This command displays the New File window, as shown in Figure 4-6. Select Cocoa Touch Class in the left column and then select the “UIViewController subclass” image. Finally, select the “UITableViewController subclass” check box and click Next. In the subsequent window, supply a suitable name for the header and implementation files of your custom class and click Finish.

Figure 4-6 Specifying a custom subclass of `UITableViewController`



As with the `RootViewController` class that comes with the Navigation-based Application template, you must implement the methods of the custom table-view controller class that populate the table view with data. You may also programmatically set properties of the table view or the navigation bar (typically in the `initWithStyle:` or `viewDidLoad` methods). When users select an item in the table view managed by the `RootViewController` class, you allocate and initialize an instance of the second table-view controller and push it onto the stack managed by the application's navigation controller; [“Managing Selections”](#) (page 78) describes this procedure in detail.

Note Populating a table view with data and configuring a table view are discussed in [“Populating the Table View With Data”](#) (page 44) and [“Optional Table-View Configurations”](#) (page 51), respectively.

If you prefer to load the table view managed by a custom table-view controller from a nib file, you must do the following:

1. In Interface Builder, create an empty Cocoa Touch nib file (File > New).
2. Drag a `UITableViewController` object from the Interface Builder Library into the nib document window.
3. Save the nib file in your project directory under an appropriate name and, when prompted, select your project to have the nib file added to it.
4. Select Table View Controller in the nib document window and open the Identity pane of the inspector. Set the class to your custom table-view controller class.
5. Select File's Owner in the nib document window and set its class identity to the custom table-view controller class.
6. Customize the table view in Interface Builder.
7. Select the table-view controller in the nib document window, open the Attributes pane of the inspector, and enter (or select) the name of the nib file in the Nib Name field.

When at runtime the current table-view controller allocates and initializes an instance of the new table-view controller, the associated nib file is loaded.

Creating a Table View Programmatically

If you choose not to use `UITableViewController` for table-view management, you must replicate what this class gives you “for free.” The class creating the table view (a `UIViewController` subclass in the examples below) typically makes itself the data source and delegate by adopting the `UITableViewDataSource` and `UITableViewDelegate` protocols. The adoption syntax appears just after the superclass in the `@interface` directive, as shown in [“Creating a Table View Programmatically.”](#)

Listing 4-2 Adopting the data source and delegate protocols

```
@interface RootViewController : UIViewController <UITableViewDelegate,
UITableViewDataSource> {
    NSArray *timeZoneNames;
}

@property (nonatomic, retain) NSArray *timeZoneNames;
@end
```

The next step is for the client to allocate and initialize an instance of the `UITableView` class. “Creating a Table View Application the Easy Way” gives an example of a client that creates a `UITableView` object in the plain style, specifies its autosizing characteristics, and then sets itself to be both data source and delegate. Again, keep in mind that the `UITableViewController` does all of this for you automatically.

Listing 4-3 Creating a table view

```
- (void)loadView
{
    UITableView *tableView = [[UITableView alloc] initWithFrame:[UIScreen
 mainScreen] applicationFrame]
                                style:UITableViewStylePlain];

    tableView.autoresizingMask =
    UIViewAutoresizingFlexibleHeight|UIViewAutoresizingFlexibleWidth;
    tableView.delegate = self;
    tableView.dataSource = self;
    [tableView reloadData];

    self.view = tableView;
    [tableView release];
}
```

Because in this example the class creating the table view is a subclass of `UIViewController`, it assigns the created table view to its `view` property, which it inherits from that class. It also sends a `reloadData` message to the table view, causing the table view to initiate the procedure for populating its sections and rows with data.

Populating the Table View With Data

Just after it is created, a table-view object receives a `reloadData` message, which tells it to start querying the data source and delegate for the information it needs for the sections and rows it displays. The table view immediately asks the data source for its logical dimensions—that is, the number of sections and the number of rows in each section. It then repeatedly invokes the `tableView:cellForRowAtIndexPath:` to get a cell object for each visible row; it uses this `UITableViewCell` object to draw the content of the row. (Scrolling a table view also causes an invocation of `tableView:cellForRowAtIndexPath:` for each newly visible row.)

“Populating the Table View With Data” shows how the data source and the delegate implement the required protocol methods for configuring a table view.

Listing 4-4 Populating a table view with data

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [regions count];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    // Number of rows is the number of time zones in the region for the specified
    section.
    Region *region = [regions objectAtIndex:section];
    return [region.timeZoneWrappers count];
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    // The header for the section is the region name -- get this from the region
    at the section index.
    Region *region = [regions objectAtIndex:section];
    return [region name];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *MyIdentifier = @"MyIdentifier";
```

```
UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:MyIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:MyIdentifier] autorelease];
    }
    Region *region = [regions objectAtIndex:indexPath.section];
    TimeZoneWrapper *timeZoneWrapper = [region.timeZoneWrappers
objectAtIndex:indexPath.row];
    cell.textLabel.text = timeZoneWrapper.localeName;
    return cell;
}
```

The data source in its implementation of the `tableView:cellForRowAtIndexPath:` method returns a configured cell object that the table view can use to draw a row. For performance reasons, the data source tries to reuse cells as much as possible. It first asks the table view for a specific reusable cell object by sending it `dequeueReusableCellWithIdentifier:.` message. If no such object exists, the data source creates it, assigning it a reuse identifier. It sets the cell's content (its text in this example) and returns it. [“A Closer Look at Table-View Cells”](#) (page 54) discusses this data-source method and `UITableViewCell` objects in more detail.

The implementation of `tableView:cellForRowAtIndexPath:` in Listing 4-4 illustrates an aspect of the table view API that you'll frequently encounter. The method includes an `NSIndexPath` argument that identifies the table-view section and row for the cell that the data source is to provide. (In this case, the section index is not needed because the table view has only one section.) UIKit declares a category of the `NSIndexPath` class, which is defined in the Foundation framework. This category extends the class to enable the identification of table-view rows by section and row index numbers. For information on this category, see *NSIndexPath UIKit Additions*.

Populating an Indexed List

An indexed list (also known as a section index table view) is ideally suited for navigating large amounts of data organized by a conventional ordering scheme such as an alphabet. An indexed list is a table view in the plain style that is specially configured through three `UITableViewDataSource` methods:

`sectionIndexTitlesForTableView:`, `tableView:titleForHeaderInSection:`, and `tableView:sectionForSectionIndexTitle:atIndex:.` The first method returns an array of the strings to use as the index entries (in order), the second method maps these index strings to the titles of the table-view's sections (they don't have to be the same), and the third method returns the section index related to the entry the user tapped in the index.

The data that you use to populate an indexed list should be organized to reflect this indexing model. Specifically, you need to build an array of arrays. Each inner array corresponds to a section in the table; section arrays are sorted (or collated) within the outer array according to the prevailing ordering scheme, which is often an alphabetical scheme (for example, A through Z). Additionally, the items in each section array are sorted. You could build and sort this array of arrays yourself, but fortunately the `UICollectionIndexingWrapper` class makes the tasks of building and sorting these data structures and providing data to the table view much easier. The class also collates items in the arrays according to the current localization.

Note The `UICollectionIndexingWrapper` class was added to the UIKit framework in iOS 3.0.

However you internally manage this array-of-arrays structure is up to you. The objects to be collated should have a property or method that returns a string value that the `UICollectionIndexingWrapper` class uses in collation; if it is a method, it should have no parameters. You might find it convenient to define a custom model class whose instances represent the rows in the table view. These model objects not only return a string value but define a property that holds the index of the section array to which the object is assigned. Listing 4-5 illustrates the definition of a class that declares a `name` property for the former purpose and a `sectionNumber` property for the latter purpose.

Listing 4-5 Defining the model-object interface

```
@interface State : NSObject {
    NSString *name;
    NSString *capitol;
    NSString *population;
    NSInteger sectionNumber;
}

@property(nonatomic,copy) NSString *name;
@property(nonatomic,copy) NSString *capitol;
@property(nonatomic,copy) NSString *population;
@property NSInteger sectionNumber;
@end
```

Before your table-view controller is asked to populate the table view, load the data to be used from whatever source and create instances of your model class from this data. The example in Listing 4-6 loads data defined in a property list and creates the model objects from that. It also obtains the shared instance of `UILocalizedIndexedCollation` and initializes the mutable array (`states`) that will contain the section arrays.

Listing 4-6 Loading the table-view data and initializing the model objects

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

    UILocalizedIndexedCollation *theCollation = [UILocalizedIndexedCollation
currentCollation];

    self.states = [NSMutableArray arrayWithCapacity:1];

    NSString *thePath = [[NSBundle mainBundle] pathForResource:@"States"
ofType:@"plist"];
    NSArray *tempArray;
    NSMutableArray *statesTemp;
    if (thePath && (tempArray = [NSArray arrayWithContentsOfFile:thePath]) ) {
        statesTemp = [NSMutableArray arrayWithCapacity:1];
        for (NSDictionary *stateDict in tempArray) {
            State *aState = [[State alloc] init];
            aState.name = [stateDict objectForKey:@"Name"];
            aState.population = [stateDict objectForKey:@"Population"];
            aState.capitol = [stateDict objectForKey:@"Capitol"];
            [statesTemp addObject:aState];
            [aState release];
        }
    } else {
        return;
    }
}
```

Once the data source has this “raw” array of model objects, it can process it with the facilities of the `UILocalizedIndexedCollation` class. The code snippet in Listing 4-7 marks the significant parts with numbers.

Listing 4-7 Preparing the data for the indexed list

```
// viewDidLoad continued...
// (1)
for (State *theState in statesTemp) {
    NSInteger sect = [theCollation sectionForObject:theState
collationStringSelector:@selector(name)];
    theState.sectionNumber = sect;
}
// (2)
NSInteger highSection = [[theCollation sectionTitles] count];
NSMutableArray *sectionArrays = [NSMutableArray arrayWithCapacity:highSection];
for (int i=0; i<=highSection; i++) {
    NSMutableArray *sectionArray = [NSMutableArray arrayWithCapacity:1];
    [sectionArrays addObject:sectionArray];
}
// (3)
for (State *theState in statesTemp) {
    [(NSMutableArray *)[sectionArrays objectAtIndex:theState.sectionNumber]
addObject:theState];
}
// (4)
for (NSMutableArray *sectionArray in sectionArrays) {
    NSArray *sortedSection = [theCollation sortedArrayFromArray:sectionArray
collationStringSelector:@selector(name)];
    [self.states addObject:sortedSection];
}
} // end of viewDidLoad
```

1. The data source enumerates the array of model objects and sends `sectionForObject:collationStringSelector:` to the collation manager on each iteration. This method takes as arguments a model object and a property or method of the object that it uses in collation. Each call returns the index of the section array to which the model object belongs, and that value is assigned to the `sectionNumber` property.
2. The data source source then creates a (temporary) outer mutable array and mutable arrays for each section; it adds each created section array to the outer array.
3. It then enumerates the array of model objects and adds each object to its assigned section array.

4. The data source enumerates the array of section arrays and calls `sortedArrayFromArray:collationStringSelector:` on the collation manager to sort the items in each array. It passes in a section array and a property or method that is to be used in sorting the items in the array. Each sorted section array is added to the final outer array.

Now the data source is ready to populate its table view with data. It implements the methods specific to indexed lists as shown in Listing 4-8. In doing this it calls two `UILocalizedIndexedCollation` methods: `sectionIndexTitles` and `sectionForSectionIndexTitleAtIndex:`. Also note that in `tableView:titleForHeaderInSection:` it suppresses any headers from appearing in the table view when the associated section does not have any items.

Listing 4-8 Providing section-index data to the table view

```
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView {
    return [[UILocalizedIndexedCollation currentCollation] sectionIndexTitles];
}

- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    if ([[self.states objectAtIndex:section] count] > 0) {
        return [[[UILocalizedIndexedCollation currentCollation] sectionTitles]
objectAtIndex:section];
    }
    return nil;
}

- (NSInteger)tableView:(UITableView *)tableView sectionForSectionIndexTitle:(NSString
*)title atIndex:(NSInteger)index
{
    return [[[UILocalizedIndexedCollation currentCollation]
sectionForSectionIndexTitleAtIndex:index];
}
}
```

Finally, the data source should implement the `UITableViewDataSource` methods that are common to all table views. Listing 4-9 gives examples of these implementations, and illustrates how to use the `section` and `row` properties of the table view–specific category of the `NSIndexPath` class described in *NSIndexPath UIKit Additions*.

Listing 4-9 Populating the rows of an indexed list

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [self.states count];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [[self.states objectAtIndex:section] count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"StateCell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }
    State *stateObj = [[self.states objectAtIndex:indexPath.section]
        objectAtIndex:indexPath.row];
    cell.textLabel.text = stateObj.name;
    return cell;
}
```

After initially populating the table view following the procedure outlined above, you can thereafter reload the contents of the index by calling `reloadSectionIndexTitles`, a method introduced in iOS 3.0.

For table views that are indexed lists, when the delegate assigns cells for rows in `tableView:cellForRowAtIndexPath:`, it should ensure that the `accessoryType` property of the cell is set to `UITableViewCellAccessoryNone`. You can force a redisplay of the section titles of an indexed list by calling the `reloadSectionIndexTitles` method (available in iOS 3.0).

Optional Table-View Configurations

The table view API allows you to configure various visual and behavioral aspects of a table view, including specific rows and sections. The following examples serve to give you some idea of the options available to you.

In the same block of code that creates the table view, you can apply global configurations using certain methods of the `UITableView` class. The code example in Listing 4-10 adds a custom title for the table view (using a `UILabel` object).

Listing 4-10 Adding a title to the table view

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

    CGRect titleRect = CGRectMake(0, 0, 300, 40);
    UILabel *tableTitle = [[UILabel alloc] initWithFrame:titleRect];
    tableTitle.textColor = [UIColor blueColor];
    tableTitle.backgroundColor = [self.tableView backgroundColor];
    tableTitle.opaque = YES;
    tableTitle.font = [UIFont boldSystemFontOfSize:18];
    tableTitle.text = [curTrail objectForKey:@"Name"];
    self.tableView.tableHeaderView = tableTitle;
    [self.tableView reloadData];
    [tableTitle release];
}
```

The example in Listing 4-11 returns a title string for a section header.

Listing 4-11 Returning a header title for a specific section

```
- (NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section {
    // Returns section title based on physical state: [solid, liquid, gas,
    artificial]
    return [[[PeriodicElements sharedPeriodicElements] elementPhysicalStatesArray]
    objectAtIndex:section];
}
```

The code in Listing 4-12 moves a specific row to the next level of indentation.

Listing 4-12 Custom indentation of a row

```
- (NSInteger)tableView:(UITableView *)tableView
indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath {
    if ( indexPath.section==TRAIL_MAP_SECTION && indexPath.row==0 ) {
        return 2;
    }
    return 1;
}
```

The example in Listing 4-13 varies the height of a specific row based on its index value.

Listing 4-13 Varying row height

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
*)indexPath
{
    CGFloat result;

    switch ([indexPath row])
    {
        case 0:
        {
            result = kUIRowHeight;
            break;
        }
        case 1:
        {
            result = kUIRowLabelHeight;
            break;
        }
    }
    return result;
}
```

You can also affect the appearance of rows by returning custom `UITableViewCell` objects with specially formatted subviews for content in `tableView:cellForRowAtIndexPath:`. [“A Closer Look at Table-View Cells”](#) (page 54) discusses cell customization.

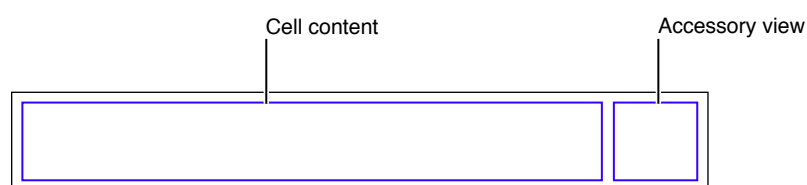
A Closer Look at Table-View Cells

A table view uses cell objects to draw its visible rows and caches those objects as long as the rows are visible. These objects inherit from the `UITableViewCell` class. The table view's data source provides the cell objects to the table view by implementing the `tableView:cellForRowAtIndexPath:` method, a required method of the `UITableViewDataSource` protocol. The following sections describe the characteristics of table-view cell objects, explain how to use the default capabilities of `UITableViewCell` for setting cell content, and show how to create custom `UITableViewCell` objects.

Characteristics of Cell Objects

A cell object has various parts which can change depending on the mode of the table view. Normally, most of a cell object is reserved for its content: text, image, or any other kind of distinctive identifier. As shown in Figure 5-1, a smaller area on the right side of the cell is reserved for accessory views: disclosure indicators, detail disclosure controls, control objects such as sliders or switches, and custom views. Figure 5-1 shows the major parts of a cell.

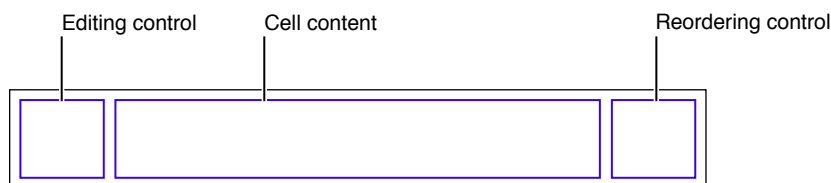
Figure 5-1 Parts of a table-view cell



When the table view goes into editing mode, the editing control for each cell object (if it's configured to have such a control) appears on its left side in the area shown in Figure 5-2; the editing control can be either a deletion control (a red minus sign inside a circle) or an insertion control (a green plus sign inside a circle). The cell's content is pushed toward the right to make room for the editing control. If the cell object is configured

for reordering (that is, relocation within the table view), the reordering control appears in the right side of the cell, next to any accessory view specified for editing mode. The reordering control is a stack of horizontal lines; to relocate a row within its table view, users press on the reordering control and drag the cell.

Figure 5-2 Parts of a table-view cell—editing mode



If a cell object is reusable—the typical case—the data source assigns the cell object a reuse identifier (an arbitrary string) when it creates the cell. The table view stores the cell object in an internal queue. When the table view subsequently requests another cell object, the data source can access the queued object by sending a `dequeueReusableCellWithIdentifier:` message to the table view, passing in a reuse identifier. The data source simply sets the content of the cell and any special properties before returning it. This reuse of cell objects is a performance enhancement because it eliminates the overhead of cell creation. Having multiple cell objects in a queue, each with its own identifier, makes it possible to have table views constructed from cell objects of different types. For example, some rows of a table view could have content based on the image and text properties of a `UITableViewCell` in a predefined style while other rows could be based on a customized `UITableViewCell` that defines a special format for its content.

When providing cells for the table view, there are three general approaches you can take. You can either use ready-made cell objects in a range of styles; you can add your own subviews to the cell object's content view (which can be done in the Interface Builder application); or you can use cell objects created from a custom subclass of `UITableViewCell`. Note that the content view is only a container of other views and displays no content itself.

Using Cell Objects in Predefined Styles

Using the `UITableViewCell` class directly, you can create “off the shelf” cell objects in a range of predefined styles. “[Standard Styles for Table-View Cells](#)” (page 15) describes these standard cells and provides examples of how they look in a table view. These cells are associated with the following `enum` constants, declared in `UITableViewCell.h`:

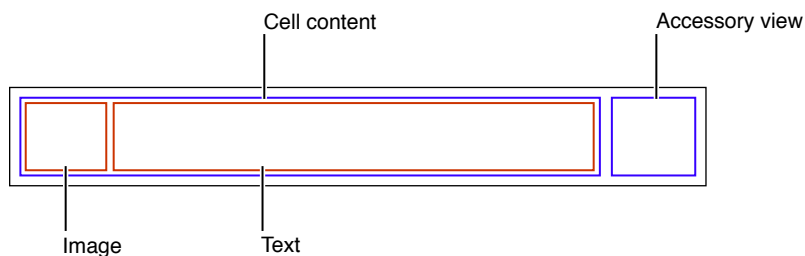
```
typedef enum {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
```

```
UITableViewCellStyleSubtitle  
{ UITableViewCellStyle;
```

Note The predefined cell styles and their corresponding content properties were introduced in iOS 3.0.

These cell objects have two kinds of content: one or more titles (text strings) and, in some cases, an image. Figure 5-3 shows the approximate areas for image and text. As an image expands to the right, it pushes the text in the same direction.

Figure 5-3 Default cell content in a `UITableViewCell` object



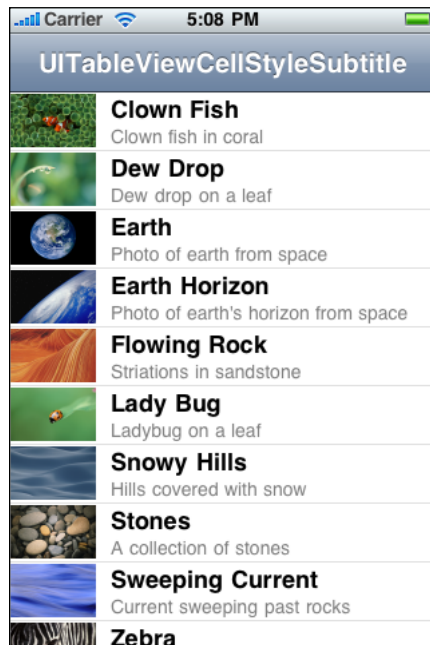
The `UITableViewCell` class defines properties for cell content in these predefined cell styles:

- `textLabel`—The main label for text in the cell (a `UILabel` object)
- `detailTextLabel`—The secondary label for text in the cell when there is additional detail (a `UILabel` object)
- `imageView`—An image view to hold an image (a `UIImageView` object)

Because the first two of these properties are labels, you can set the font, alignment, line-break mode, and color of the associated text through the properties defined by the `UILabel` class (including the color of text when the row is highlighted). For the image-view property, you can also set an alternate image for when the cell is highlighted using the `highlightedImage` property of the `UIImageView` class.

Figure 5-4 gives an example of a table view whose rows are drawn using a `UITableViewCell` object in the `UITableViewCellStyleSubtitle` style; it includes both an image and, for textual content, a title and a subtitle.

Figure 5-4 A table view with rows showing both images and text



Listing 5-1 shows the table view data source implementation of `tableView:cellForRowAtIndexPath:` that creates the table view in [Figure 5-4](#) (page 57). Typically, the first thing the data source should do is send `dequeueReusableCellWithIdentifier:` to the table view, passing in a reuse-identifier string. If the table view does not return a reusable cell object, the data source creates one, assigning the object a reuse identifier in the final parameter of `initWithStyle:reuseIdentifier:`. At this point it also sets general properties of the cell object for the table view (in this case, its selection style). Then it sets the cell object's content, both text and image.

Listing 5-1 Configuring a `UITableViewCell` object with both image and text

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"MyIdentifier"];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
reuseIdentifier:@"MyIdentifier"] autorelease];
```

```
        cell.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    NSDictionary *item = (NSDictionary *)[self.content objectAtIndex:indexPath.row];
    cell.textLabel.text = [item objectForKey:@"mainTitleKey"];
    cell.detailTextLabel.text = [item objectForKey:@"secondaryTitleKey"];
    NSString *path = [[NSBundle mainBundle] pathForResource:[item
objectForKey:@"imageKey"] ofType:@"png"];
    UIImage *theImage = [UIImage imageWithContentsOfFile:path];
    cell.imageView.image = theImage;
    return cell;
}
```

The table view's data source implementation of `tableView:cellForRowAtIndexPath:` should *always* reset all content when reusing a cell.

When you configure a `UITableViewCell` object, you also can set various other properties including (but not limited to) the following:

- `selectionStyle`—Controls the appearance of the cell when selected.
- `accessoryType` and `accessoryView`—Allows you to set one of the standard accessory views (disclosure indicator or detail disclosure control) or a custom accessory view for a cell in normal (non-editing) mode. For a custom view, you may provide any `UIView` object, such as a slider, a switch, or a custom view.
- `editingAccessoryType` and `editingAccessoryView`—Allows you to set one of the standard accessory views (disclosure indicator or detail disclosure control) or a custom accessory view for a cell in editing mode. For a custom view, you may provide any `UIView` object, such as a slider, a switch, or a custom view. (These properties were introduced in iOS 3.0.)
- `showsReorderControl`—Specifies whether it shows a reordering control when in editing mode. The related but read-only `editingStyle` property specifies the type of editing control the cell has (if any). The delegate returns the value of the `editingStyle` property in its implementation of the `tableView:editingStyleForRowAtIndexPath:` method.
- `backgroundView` and `selectedBackgroundView`—Provides a background view (when a cell is unselected and selected) to display behind all other views of the cell.
- `indentationLevel` and `indentationWidth`—Specifies the indentation level for cell content and the width of each indentation level.

Because a table-view cell inherits from `UIView`, you can also affect its appearance and behavior by setting the properties defined by that superclass. For example, to affect the background color of a cell, you could set its `backgroundColor` property. Listing 5-2 shows how you might alternate the background color of rows (via their backing cells) in a table view.

Listing 5-2 Alternating the background color of cells in `tableView:willDisplayCell:forRowAtIndexPath:`

```
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (indexPath.row == 0 || indexPath.row%2 == 0) {
        UIColor *altCellColor = [UIColor colorWithWhite:0.7 alpha:0.1];
        cell.backgroundColor = altCellColor;
    }
}
```

Listing 5-2 also illustrates an important aspect of the table-view API. A table view sends a `tableView:willDisplayCell:forRowAtIndexPath:` message to its delegate just before it draws a row. If the delegate chooses to implement this method, it can make last-minute changes to the cell object before it is displayed. In this method the delegate should change state-based properties set earlier by the table view, such as selection and background color, and not content.

Customizing Cells

`UITableViewCell` objects in their various predefined styles suffice for most of the rows that table views display. With these ready-made cell objects, rows can include one or two styles of text, often an image, and an accessory view of some sort. The application can modify the text in its font, color, and other characteristics, and it can supply an image for the row in its selected state as well as its normal state.

However, as flexible and useful as this cell content is, it might not satisfy the requirements of all applications. For example, the labels permitted by a native `UITableViewCell` object are pinned to specific locations within a row, and the image must appear on the left side of the row. If you want the cell to have different content components and to have these laid out in different locations, or if you want different behavioral characteristics for the cell, you have two alternatives. You can add subviews to the `contentView` property of the cell object or you can create a custom subclass of `UITableViewCell`.

- You should add subviews to a cell's content view when your content layout can be specified entirely with the appropriate autosizing settings and when you don't need to modify the default behavior of the cell.
- You should create a custom subclass when your content requires custom layout code or when you need to change the default behavior of the cell, such as in response to editing mode.

The following sections discuss both approaches.

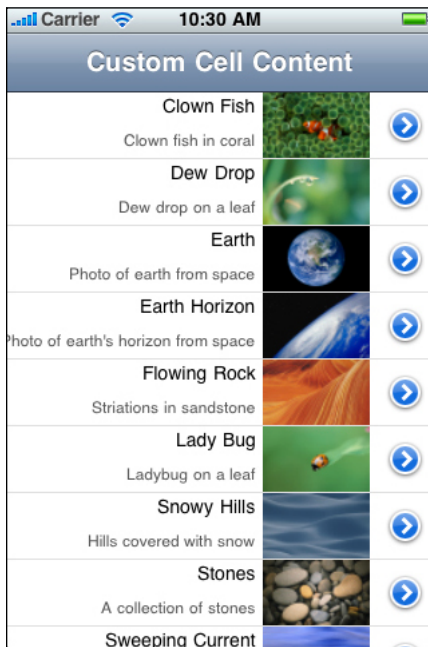
Programmatically Adding Subviews to a Cell's Content View

A cell that a table view uses for displaying a row is a view (`UITableViewCell` inherits from `UIView`). As a view, a cell has a content view—a superview for cell content—that it exposes as a property. To customize the appearance of rows in a table view, you can add subviews to the cell's content view, which is accessible through its `contentView` property, and lay them out in the desired locations in their superview's coordinates. You may configure and lay them out programmatically or in Interface Builder. (The approach using Interface Builder is discussed in [“Loading Custom Table-View Cells From Nib Files”](#) (page 64).)

One advantage of this approach is its relative simplicity; it doesn't require you to create a custom subclass of `UITableViewCell` and handle all of the implementation details required for custom views. However, if you do take this approach, avoid making the views transparent, if you can. Transparent subviews affect scrolling performance because of the increased compositing cost. Subviews should be opaque, and typically should have the same background color as the cell. And if the cell is selectable, make sure that the cell content is highlighted appropriately when selected; this happens automatically if the subview implements (if appropriate) the accessor methods for the `highlighted` property.

Let's say you wanted a cell with text and image content in entirely different locations than those provided by the standard cell styles. For example, you want the image on the right side of the cell and the title and subtitle of the cell right-aligned against the left side of the image. Figure 5-5 show how a table view with rows drawn with such a cell might look. (This example is for illustration only, and is not intended as a human-interface model.)

Figure 5-5 Cells with custom content as subviews



The code example in Listing 5-3 illustrates how the data source programmatically composes the cell with which this table view draws its rows. In `tableView:cellForRowAtIndexPath:`, it first checks to see the table view already has a cell object with the given reuse identifier. If there is no such object, the data source creates two label objects and one image view with specific frames within the coordinate system of their superview (the content view). It also sets attributes of these objects. Having acquired an appropriate cell to use, the data source sets the cell's content before returning the cell.

Listing 5-3 Adding subviews to a cell's content view

```
#define MAINLABEL_TAG 1
#define SECONDLABEL_TAG 2
#define PHOTO_TAG 3

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
static NSString *CellIdentifier = @"ImageOnRightCell";

UILabel *mainLabel, *secondLabel;
UIImageView *photo;

UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;

    mainLabel = [[[UILabel alloc] initWithFrame:CGRectMake(0.0, 0.0, 220.0,
15.0)] autorelease];
    mainLabel.tag = MAINLABEL_TAG;
    mainLabel.font = [UIFont systemFontOfSize:14.0];
    mainLabel.textAlignment = UITextAlignmentRight;
    mainLabel.textColor = [UIColor blackColor];
    mainLabel.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
    [cell.contentView addSubview:mainLabel];

    secondLabel = [[[UILabel alloc] initWithFrame:CGRectMake(0.0, 20.0, 220.0,
25.0)] autorelease];
    secondLabel.tag = SECONDLABEL_TAG;
    secondLabel.font = [UIFont systemFontOfSize:12.0];
    secondLabel.textAlignment = UITextAlignmentRight;
    secondLabel.textColor = [UIColor darkGrayColor];
    secondLabel.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
    [cell.contentView addSubview:secondLabel];

    photo = [[[UIImageView alloc] initWithFrame:CGRectMake(225.0, 0.0, 80.0,
45.0)] autorelease];
    photo.tag = PHOTO_TAG;
    photo.autoresizingMask = UIViewAutoresizingFlexibleLeftMargin |
UIViewAutoresizingFlexibleHeight;
    [cell.contentView addSubview:photo];
```

```
    } else {
        mainLabel = (UILabel *)[cell.contentView viewWithTag:MAINLABEL_TAG];
        secondLabel = (UILabel *)[cell.contentView viewWithTag:SECONDLABEL_TAG];
        photo = (UIImageView *)[cell.contentView viewWithTag:PHOTO_TAG];
    }
    NSDictionary *aDict = [self.list objectAtIndex:indexPath.row];
    mainLabel.text = [aDict objectForKey:@"mainTitleKey"];
    secondLabel.text = [aDict objectForKey:@"secondaryTitleKey"];
    NSString *imagePath = [[NSBundle mainBundle] pathForResource:[aDict
objectForKey:@"imageKey"] ofType:@"png"];
    UIImage *theImage = [UIImage imageWithContentsOfFile:imagePath];
    photo.image = theImage;

    return cell;
}
```

Note that when the data source creates the cells, it assigns each subview a tag. A tag is an identifier of a view; it allows you to locate a view in its view hierarchy by calling the `viewWithTag:` method. If later the delegate gets the designated cell from the table view's queue, it uses the tags to obtain references to the three subviews prior to assigning them content.

Also note that this code creates a `UITableViewCell` object in the predefined default style (`UITableViewCellStyleDefault`). However, because the content properties of the standard cells—`textLabel`, `detailTextLabel`, and `imageView`—are `nil` until assigned content, you may use any predefined cell as the template for customization.

Note An alternative, and perhaps easier, approach for programmatically composing the same cells shown in [Figure 5-5](#) (page 61) is to subclass `UITableViewCell` and create instances in the `UITableViewCellStyleSubtitle` style. Then override the `layoutSubviews` method to reposition the `textLabel`, `detailTextLabel`, and `imageView` subviews (after calling `super`).

A common technique for achieving “attributed string” effects with textual content is to lay out `UILabel` subviews of the `UITableViewCell` content view. The text of each label can have its own font, color, size, alignment, and other characteristics. There can be no variation of textual attributes within a label object, so you must create multiple labels and lay them out relative to each other if you want that kind of variation within a cell.

Loading Custom Table-View Cells From Nib Files

You can easily do the same subview customizations of table-view cells in Interface Builder that you can do programmatically. Cells in nib files require you to take one of two approaches based on whether the cells are for static or dynamic row content. With dynamic content, the table view is a list with a large (and potentially unbounded) number of rows. With static content, the number of rows is a finite, known quantity; a table view that presents a detail view of an item typically has static content. These two content types have different implications for nib files. With dynamic content, each `UITableViewCell` object should be in its own nib file. With static content, the `UITableViewCell` objects used in the table view can be in the same nib file as the table view.

The following sections demonstrate different ways to load nib files containing custom-configured table-view cells. They also show different techniques for access the subviews of the cells; one technique uses tags (the `tag` property defined by `UIView`) and the other uses outlet properties declared by the custom table-view controller class. With the former technique there is a small cost associated with searching for the tagged view; with the latter technique you can access the subview directly through the outlet connection.

The Technique for Dynamic Row Content

In this section you compose a simple table-view cell in Interface Builder and save it to a nib file. At runtime, the data source loads the nib file, prepares the cell, and gives it to its table view for drawing the rows depicted in Figure 5-6.

Figure 5-6 Table-view rows drawn with a cell from a nib file



0	300	>
1	299	>
2	298	>
3	297	>
4	296	>
5	295	>
6	294	>
7	293	>
8	292	>
9	291	>

In the view controller that manages the table view (typically a `UITableViewController` object), define an outlet property for the customized table-view cell you are going to load from the nib file, as shown in Listing 5-4. Make sure you synthesize accessor methods for the property in the implementation file.

Listing 5-4 Defining an outlet for the cell

```
@interface TVController : UITableViewController {
    UITableViewCell *tvCell;
}

@property (nonatomic, assign) IBOutlet UITableViewCell *tvCell;
@end
```

In Interface Builder complete the following steps:

1. Create a nib file.

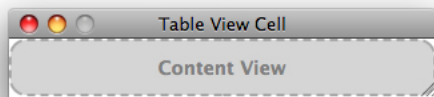
To this, choose New from the File menu, select the Empty template, and click Choose.

2. Save the nib file under an appropriate name and, when prompted, add it to the project.

This name is what you specify as the first argument of the `loadNibNamed:owner:options:` method call that loads the nib file from the application's main bundle. See [Listing 5-5](#) (page 66) for a code example.

3. Drag a Table View Cell object from the Library into the nib document window.

The cell object indicates where its content view is:



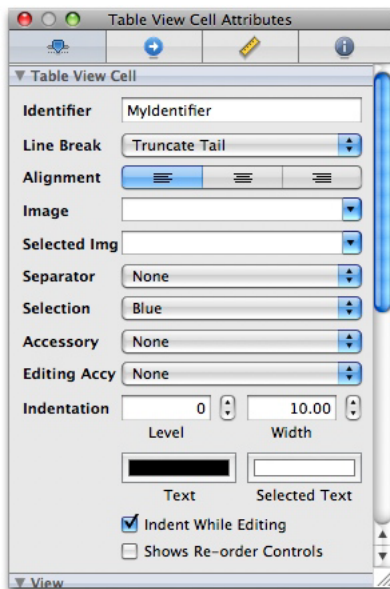
4. Drag objects from the Library onto the content view.

For this example, drag two label objects and position them near the ends of each cell (leaving room for the accessory view).

5. Select the objects on the content view and set their attributes, sizes, and autosizing characteristics.

An important attribute to set for the programmatic portion of this procedure is each object's `tag` property. Find this property in the View section of the Attributes pane and assign each object a unique integer.

6. Select the cell itself and set any general attributes you want it to have, such as alignment, font size, color, line-break mode, and so on.



Always set a string as an identifier of the cell; the table view requires the identifier to fetch a cell from its cache, if it's present. Setting the identifier is especially important if you have more than one kind of cell for a table view. You may also set the height and autosizing characteristics of the cell at this time.

7. Select File's Owner in the nib document window, open the Identity pane of the inspector, and set the class of File's Owner to your custom view controller class.
8. Connect the cell outlet of File's Owner (now the placeholder instance of your custom subclass) to the table-view cell object in the nib-file document.

Now save the nib file and return to the Xcode project. Write all the code you normally would to obtain the table-view's data and set up the table view. Implement the data source methods as you normally would, except for `tableView:cellForRowAtIndexPath:`. Implement that in a manner similar to the example in Listing 5-5.

Listing 5-5 Loading a cell from a nib file and assigning it content

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *MyIdentifier = @"MyIdentifier";

    UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:MyIdentifier];

    if (cell == nil) {
```

```
        [[NSBundle mainBundle] loadNibNamed:@"TVCell" owner:self options:nil];
        cell = tvCell;
        self.tvCell = nil;
    }
    UILabel *label;
    label = (UILabel *)[cell viewWithTag:1];
    label.text = [NSString stringWithFormat:@"%d", indexPath.row];

    label = (UILabel *)[cell viewWithTag:2];
    label.text = [NSString stringWithFormat:@"%d", NUMBER_OF_ROWS - indexPath.row];

    return cell;
}
```

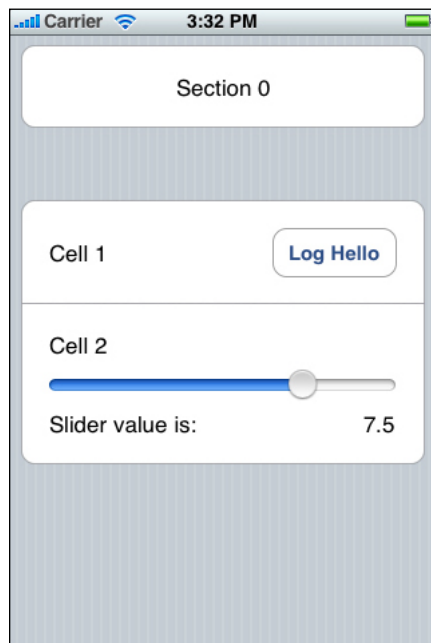
There are a few useful aspects of this code to note:

- The string identifier you assigned to the cell in Interface Builder is the same string passed to the table view in `dequeueReusableCellWithIdentifier:`. The cell is loaded from the nib file as required to fill the table view. If rows are scrolled out of view, the associated cells become available in the table view's cache. Keep in mind that the table view may request additional cells—that is, more than are present in the reuse queue—at any time for a variety of reasons. For example, if you insert a hundred new rows, the table view might ask for a hundred new cells at once, none of which will be kept in the reuse queue until after the insertion operation completes.
- The nib file is loaded using the `NSBundle` method `loadNibNamed:owner:options:`, which passes `self` as the owner (remember, File's Owner refers to your table-view controller).
- Because of the outlet connection you made, the `tvCell` outlet now has a reference to the cell loaded from the nib file. Immediately assign the cell to the passed-in `cell` variable and set the outlet to `nil`.
- The code gets the labels in the cell by calling `viewWithTag:`, passing in their tag integers. It can then set the textual content of the labels.

The Technique for Static Row Content

In this section you compose several table-view cells in Interface Builder and save them to the same nib file that contains the table view. At runtime, when the table view is loaded from its nib file, the data source has immediate access to these cells and composes the sections and rows of the table view with them, as depicted in Figure 5-7.

Figure 5-7 Table view rows drawn with multiple cells from a nib file



As with the procedure for dynamic content, start by adding a subclass of `UITableViewController` to your project. (See [“Adding Table Views to the Application”](#) (page 41) for details.) Define outlet properties for each of the three cells in the nib file plus an outlet property for the slider-value label in the last cell, as shown in Listing 5-6.

Listing 5-6 Defining outlet properties for the cells in the nib file

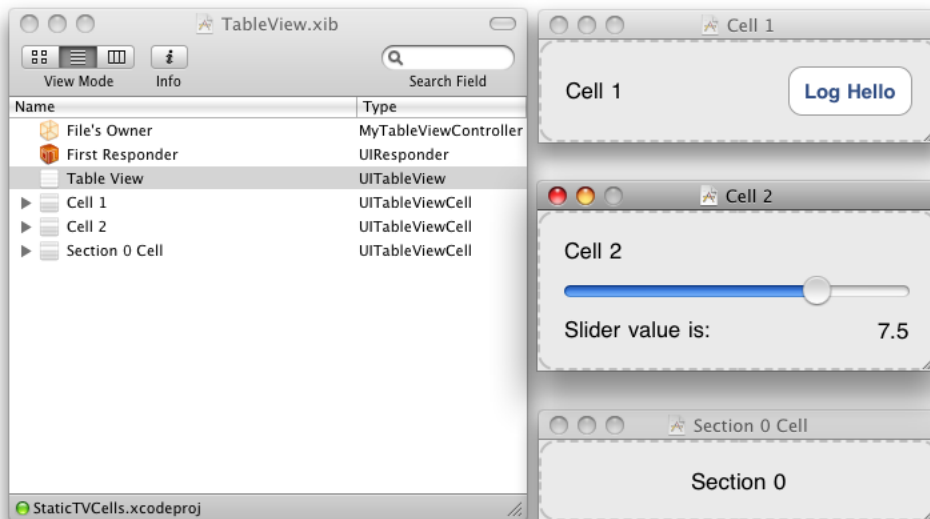
```
@interface MyTableViewController : UITableViewController {

    UITableViewCell *cell0;
    UITableViewCell *cell1;
    UITableViewCell *cell2;
    UILabel *cell2Label;
}
```

```
@property (nonatomic, retain) IBOutlet UITableViewCell *cell0;  
@property (nonatomic, retain) IBOutlet UITableViewCell *cell1;  
@property (nonatomic, retain) IBOutlet UITableViewCell *cell2;  
@property (nonatomic, retain) IBOutlet UILabel *cell2Label;  
  
- (IBAction)logHello;  
- (IBAction)sliderValueChanged:(UISlider *)slider;  
  
@end
```

Start by creating a nib file that contains a table view and make your custom table-view controller File's Owner of the nib file. Connect the view outlet of the controller to the table view and change the style of the table view to Grouped in the Attributes pane of the inspector. ("[Creating a Table View Application the Easy Way](#)" (page 37) describes how to do these things.) Then for the cells, complete the following steps:

1. Drag three Table View Cell objects from the Interface Builder Library into the nib document window.
2. Drag objects from the Library to compose the subviews of each cell as depicted here:

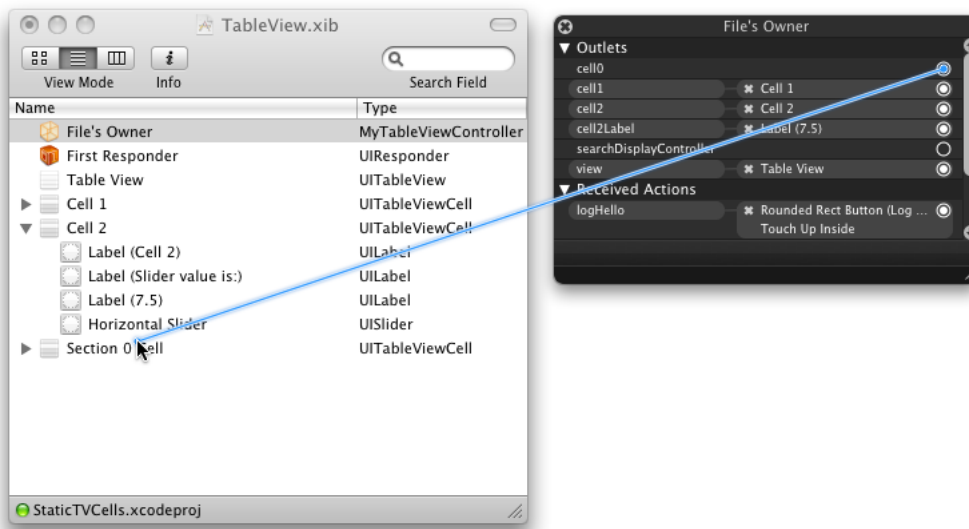


It is not necessary to assign identifiers as attributes of these cells because they are single-use cells.

3. Set any desired attributes of these objects.

For example, the slider should have a range of values from 0 to 10 with an initial value of 7.5.

4. Right-click (or Control-click) File's Owner in the nib document window to display the connections window; make outlet connections between File's Owner (your table-view controller) and each of the cell objects.



Also connect the `cell2Label` outlet to the slider-value label. Note that giving views like these tags is not necessary with static content because you can make outlet connections to them.

5. While you're at it, implement the two action methods declared in [Listing 5-6](#) (page 68) and make target-action connections as shown in the above illustration.

Save the nib file, return to the Xcode project, and implement the data source methods for the table view. When the application delegate or previous table-view controller instantiates the current table-view controller, the nib file containing the table view and the table-view cells is loaded into application memory. Because the cells in the nib file are single-use cells, you need only return them to the table view (via their outlets) when it asks for them in the `tableView:cellForRowAtIndexPath:` method, as shown in [Listing 5-7](#).

Listing 5-7 Passing nib-file cells to the table view

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    if (indexPath.section == 0) {
        return cell0;
    }
    // section 1
    if (indexPath.row == 0) {
        return cell1;
    }
}
```

```
        return cell2;  
    }  
}
```

Even though the cells in the nib file are for static content, with each cell used only once in the table view, you can still dynamically modify their contents. For example, you could easily change the text value of the label in the first cell to a property of the current data item; Listing 5-8 illustrates how this might look.

Listing 5-8 Dynamically changing the content of a nib-file cell

```
// ...  
if (indexPath.section == 0) {  
    UILabel *theLabel = (UILabel *)[cell0 viewWithTag:1];  
    theLabel.text = [self.currentAddress lastName];  
    return cell0;  
}  
// ...
```

Subclassing UITableViewCell

Another way of customizing the appearance of cell objects is to create a custom subclass of `UITableViewCell` that draws its own content. You then use instances of this class to populate the rows of a table view. This approach also gives you greater control over how the cells behave, for instance, when the table view enters editing mode. (In editing mode, the area for content shrinks.) Figure 5-8 gives an example of a custom table-view cell object.

Figure 5-8 A custom table-view cell



Before you write the first line of subclassing code, carefully consider some design aspects and performance constraints of `UITableViewCell` subclasses.

- **Draw the entire cell only when appropriate.** Your subclass of `UITableViewCell` could draw all of its content in its `drawRect:` method, but you should be aware of the potential drawbacks of this approach. Custom drawing applies to the cell's layer, which can be obscured by any views placed over it. For example, in table views in the grouped style, the background view (the `backgroundView` property) obscures any

drawing performed in `drawRect:`. The blue selection background will also obscure any drawing. Moreover, custom drawing that occurs during animation (such as when the table view enters and exits editing mode) drastically decreases performance.

An alternative is a subclass that composes the content of the cell from subviews, laying those views out in the desired way. Because those views are cached, they can simply be moved around (when, for instance, the cell goes into editing mode). [“Programmatically Adding Subviews to a Cell’s Content View”](#) (page 60) illustrates one such approach and notes another.

However, if the content of a cell is composed of more than three or four subviews, scrolling performance might suffer. In this case (and especially if the cell is not editable), consider drawing directly in one subview of the cell’s content view. The gist of this guideline is that, when implementing custom table-view cells, be aware that there is a tradeoff between optimal scrolling performance and optimal editing or reordering performance.

- **Avoid transparency.** Subviews of table-view cells have a compositing cost that you can largely mitigate by making the views opaque. Even one transparent subview per cell impacts scrolling performance. Always use opaque subviews if at all possible.
- **Mark the cell as needing display when viewable properties change.** If you have a custom reusable table cell and it displays a custom property as part of the cell content, you must be sure to send a `setNeedsDisplay` message to the cell if the value of the property changes. Otherwise UIKit doesn’t know that the cell is “dirty” and therefore won’t invoke the cell’s `drawRect:` method to have it redraw itself with the new value. A good place to call `setNeedsDisplay` is in a (non-synthesized) setter method associated with the property.

The remainder of this section takes you on a guided tour of the parts of the `CustomTableViewCell` project that implement a custom subclass of `UITableViewCell`. (This project is part of the *TableViewSuite* extended example.) This subclass implements a cell with complex content that, because it is complex, has a single custom view that draws itself. By examining how this project creates the custom cell object shown in [Figure 5-8](#) (page 71), you can gain a working understanding of how you might create your own custom subclasses of `UITableViewCell`.

The `CustomTableViewCell` project declares the interface of the `TimeZoneCell` subclass of `UITableViewCell` as shown in Listing 5-9. This interface is simple, consisting of a reference to a custom view class and two methods, one for setting the content that the custom view uses to draw and the other for redrawing the cell on demand.

Listing 5-9 Declaring the properties and methods of the `TimeZoneCell` class

```
@class TimeZoneWrapper;  
@class TimeZoneView;
```



```
@interface TimeZoneCell : UITableViewCell {
    TimeZoneView *timeZoneView;
}

@property (nonatomic, retain) TimeZoneView *timeZoneView;
- (void)setTimeZoneWrapper:(TimeZoneWrapper *)newTimeZoneWrapper;
- (void)redisplay;
@end
```

The method `setTimeZoneWrapper:` takes as an argument a custom model object that represents a time zone and lazily creates and caches derived properties that are expensive to compute. The `TimeZoneWrapper` class is important because an instance of that class is the source for each cell's content. (The implementation of the class is not shown here.)

In its implementation, the `TimeZoneCell` class overrides the `initWithStyle:reuseIdentifier:` initializer of its superclass, calling the superclass implementation as the first step. In this method it creates an instance of the `TimeZoneView` class sized to the bounds of the cell's content view. It sets the autosizing characteristics of the view and adds it as a subview of the cell's content view. Listing 5-10 shows the initialization code.

Listing 5-10 Initializing an instance of `TimeZoneCell`

```
- (id)initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseIdentifier {
    if (self = [super initWithStyle:UITableViewCellStyleDefault reuseIdentifier:reuseIdentifier]) {
        CGRect tzvFrame = CGRectMake(0.0, 0.0, self.contentView.bounds.size.width,
                                     self.contentView.bounds.size.height);
        timeZoneView = [[TimeZoneView alloc] initWithFrame:tzvFrame];
        timeZoneView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;
        [self.contentView addSubview:timeZoneView];
    }
    return self;
}
```

The `TimeZoneView` class has an interface as shown in Listing 5-11. In addition to the instance of `TimeZoneWrapper`, it encapsulates a date formatter, an abbreviation, and two flags for indicating whether the cell is highlighted and whether editing mode is effect.

Listing 5-11 Declaring the interface of the `TimeZoneView` class

```
@interface TimeZoneView : UIView {
    TimeZoneWrapper *timeZoneWrapper;
    NSDateFormatter *dateFormatter;
    NSString *abbreviation;
    BOOL highlighted;
    BOOL editing;
}
@property (nonatomic, retain) TimeZoneWrapper *timeZoneWrapper;
@property (nonatomic, retain) NSDateFormatter *dateFormatter;
@property (nonatomic, retain) NSString *abbreviation;
@property (nonatomic, getter=isHighlighted) BOOL highlighted;
@property (nonatomic, getter=isEditing) BOOL editing;

@end
```

Recall that the `TableViewCell` class declared a method for setting a `TimeZoneWrapper` object. This method simply invokes the identical setter method (non-synthesized) of the `TimeZoneView` class encapsulated by `TimeZoneCell`. That setter method is implemented as shown in Listing 5-12. In addition to providing the standard memory-management code, this setter method associates the time zone with the data formatter, creates an abbreviation for the time zone, and marks the view for redisplay.

Listing 5-12 Setting the time-zone wrapper and related values

```
- (void)setTimeZoneWrapper:(TimeZoneWrapper *)newTimeZoneWrapper {

    // If the time zone wrapper changes, update the date formatter and abbreviation
    string.
    if (timeZoneWrapper != newTimeZoneWrapper) {
        [timeZoneWrapper release];
        timeZoneWrapper = [newTimeZoneWrapper retain];
        [dateFormatter setTimeZone:timeZoneWrapper.timeZone];
    }
}
```

```
        NSString *string = [[NSString alloc] initWithFormat:@"%@" (%@)",
timeZoneWrapper.abbreviation, timeZoneWrapper.gmtOffset];
        self.abbreviation = string;
        [string release];
    }
    [self setNeedsDisplay];
}
```

After the `TimeZoneView` class is marked for redisplay its `drawRect:` method is invoked. Listing 5-13 shows representative sections of the `TimeZoneView` implementation, eliding other parts for brevity. One of these elided parts is the initial code that defines both constants for laying out the fields of the view and colors for drawn text that is conditional on whether the cell is in a normal or highlighted state. The implementation uses the `drawAtPoint:forWidth:withFont:fontSize:lineBreakMode:baselineAdjustment:` method of `NSString` to draw the text and the `drawAtPoint:` method of `UIImage` to draw the image.

Listing 5-13 Drawing the custom table-view cell

```
- (void)drawRect:(CGRect)rect {

    // set up #define constants and fonts here ...
    // set up text colors for main and secondary text in normal and highlighted
    cell states...

    CGRect contentRect = self.bounds;
    if (!self.editing) {
        CGFloat boundsX = contentRect.origin.x;
        CGPoint point;
        CGFloat actualFontSize;
        CGSize size;

        // draw main text
        [mainTextColor set];

        // draw time-zone locale string
        point = CGPointMake(boundsX + LEFT_COLUMN_OFFSET, UPPER_ROW_TOP);
        [timeZoneWrapper.timeZoneLocaleName drawAtPoint:point
        forWidth:LEFT_COLUMN_WIDTH withFont:mainFont minFontSize:MIN_MAIN_FONT_SIZE
```

```
actualFontSize:NULL lineBreakMode:UILineBreakModeTailTruncation
baselineAdjustment:UIBaselineAdjustmentAlignBaselines];

    // ... other strings drawn here...

    // draw secondary text
    [secondaryTextColor set];

    // draw the time-zone abbreviation
    point = CGPointMake(boundsX + LEFT_COLUMN_OFFSET, LOWER_ROW_TOP);
    [abbreviation drawAtPoint:point forWidth:LEFT_COLUMN_WIDTH
withFont:secondaryFont minFontSize:MIN_SECONDARY_FONT_SIZE actualFontSize:NULL
lineBreakMode:UILineBreakModeTailTruncation
baselineAdjustment:UIBaselineAdjustmentAlignBaselines];

    // ... other strings drawn here...

    // Draw the quarter image.
    CGFloat imageY = (contentRect.size.height -
timeZoneWrapper.image.size.height) / 2;
    point = CGPointMake(boundsX + RIGHT_COLUMN_OFFSET, imageY);
    [timeZoneWrapper.image drawAtPoint:point];
}
}
```

An important aspect of this `drawRect:` implementation is the `if` statement that checks whether the cell's table view is in editing mode. The view draws the content of its cell only if the cell is *not* in editing mode. If you wished, you could add an `else` clause to this statement that draws the cell when it is in editing mode; because the cell has a reduced content area in editing mode, you might have to move fields around, shrink font sizes, or even omit less-important fields. However, drawing in editing mode is not encouraged because, as you might recall, custom drawing while cells animate into and out of editing mode severely affects performance.

Finally, the data source provides its table view with the custom cell in the `tableView:cellForRowAtIndexPath:` method, as shown in Listing 5-14. For the cell content, it locates and sets the `TimeZoneWrapper` object that corresponds to the corresponding section and row of the table view.

Listing 5-14 Returning an initialized instance of the custom table-view cell

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"TimeZoneCell";

    TimeZoneCell *timeZoneCell = (TimeZoneCell *)[tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
    if (timeZoneCell == nil) {
        timeZoneCell = [[[TimeZoneCell alloc]
        initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
        autorelease];
        timeZoneCell.frame = CGRectMake(0.0, 0.0, 320.0, ROW_HEIGHT);
    }
    Region *region = [displayList objectAtIndex:indexPath.section];
    NSArray *regionTimeZones = region.timeZoneWrappers;
    [timeZoneCell setTimeZoneWrapper:[regionTimeZones objectAtIndex:indexPath.row]];
    return timeZoneCell;
}
```

A subclass of `UITableViewCell` may override the method `prepareForReuse` to reset attributes of the cell object. The table view invokes this method just before it returns a cell object to the data source in `dequeueReusableCellWithIdentifier:`. For performance reasons, you should only reset attributes of the cell that are not related to content, for example, alpha, editing, and selection state.

Cells and Table-View Performance

The proper use of table-view cells, whether off-the-shelf or custom cell objects, is a major factor in the performance of table views. You should ensure your application does the following three things:

- **Reuse cells.** Object allocation has a performance cost, especially if the allocation has to happen repeatedly over a short period—say, when the user scrolls a table view. If you reuse cells instead of allocating new ones, you greatly enhance table-view performance.
- **Avoid relayout of content.** When reusing cells with custom subviews, refrain from laying out those subviews each time the table view requests a cell. Lay out the subviews once, when the cell is created.
- **Use opaque subviews.** When customizing table view cells, make the subviews of the cell opaque, not transparent.

Managing Selections

When users tap a row of a table view, usually something happens as a result. Another table view could slide into place, the row could display a checkmark, or some other action could be performed. The following sections describe how to respond to selections and how to make selections programmatically.

Selections in Table Views

There are a few human-interface guidelines to keep in mind when dealing with cell selection in table views:

- You should never use selection to indicate state. Instead, use check marks and accessory views for showing state.
- When the user selects a cell, you should respond by deselecting the previously selected cell (by calling the `deselectRowAtIndexPath:animated:` method) as well as by performing any appropriate action, such as displaying a detail view.
- If you respond to the the selection of a cell by pushing a new view controller onto the navigation controller's stack, you should deselect the cell (with animation) when the view controller is popped off the stack.

You can control whether rows are selectable when the table view is in editing mode by setting the `allowsSelectionDuringEditing` property of `UITableView`. In addition, beginning with iOS 3.0, you can control whether cells are selectable when editing mode is not in effect by setting the `allowsSelection` property.

Responding to Selections

Users tap a row in a table view either to signal to the application that they want to know more about what that row signifies or to select what the row represents. In response to the user tapping a row, an application could do any of the following:

- Show the next level in a data-model hierarchy.
- Show a detail view of an item (that is, a leaf node of the data-model hierarchy).
- Show a checkmark in the row to indicate that the represented item is selected.

- If the touch occurred in a control embedded in the row, it could respond to the action message sent by the control.

To handle most selections of rows, the table view's delegate must implement the `tableView:didSelectRowAtIndexPath:` method. In sample method implementation shown in Listing 6-1, the delegate first deselects the selected row. Then it allocates and initializes an instance of the next table-view controller in the sequence. It sets the data this view controller needs to populate its table view and then pushes this object onto the stack maintained by the application's `UINavigationController` object.

Listing 6-1 Responding to a row selection

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    BATTrailsViewController *trailsController = [[BATTrailsViewController alloc]
initWithStyle:UITableViewStylePlain];
    trailsController.selectedRegion = [regions objectAtIndex:indexPath.row];
    [[self navigationController] pushViewController:trailsController animated:YES];
    [trailsController release];
}
```

If a row has a disclosure control—the white chevron over a blue circle—for an accessory view, clicking the control results in the delegate receiving a `tableView:accessoryButtonTappedForRowWithIndexPath:` message (instead of `tableView:didSelectRowAtIndexPath:`). The delegate responds to this message in the same general way as it does for other kinds of selections.

A row can also have a control object as its accessory view, such as a switch or a slider. This control object functions as it would in any other context: Manipulating the object in the proper way results in an action message being sent to a target object. Listing 6-2 illustrates a data source object that adds a `UISwitch` object as a cell's accessory view and then responds to the action messages sent when the switch is “flipped.”

Listing 6-2 Setting a switch object as an accessory view and responding to its action message

```
- (UITableViewCell *)tableView:(UITableView *)tv cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tv
dequeueReusableCellWithIdentifier:@"CellWithSwitch"];
    if (cell == nil) {
```

```
        cell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"CellWithSwitch"]
autorelease];

        cell.selectionStyle = UITableViewCellSelectionStyleNone;
        cell.textLabel.font = [UIFont systemFontOfSize:14];
    }
    UISwitch *switchObj = [[UISwitch alloc] initWithFrame:CGRectMake(1.0, 1.0,
20.0, 20.0)];
    switchObj.on = YES;
    [switchObj addTarget:self action:@selector(toggleSoundEffects:)
forControlEvents:(UIControlEventsValueChanged | UIControlEventTouchDragInside)];
    cell.accessoryView = switchObj;
    [switchObj release];

    cell.textLabel.text = @"Sound Effects";
    return cell;
}

- (void)toggleSoundEffects:(id)sender {
    [self.soundEffectsOn = [(UISwitch *)sender isOn];
    [self reset];
}
```

You may also define controls as accessory views of table-view cells created in Interface Builder. Drag a control object (switch, slider, and so on) into a nib document window containing a table-view cell. Then, using the connection window, make the control the accessory view of the cell. [“Loading Custom Table-View Cells From Nib Files”](#) (page 64) describes the procedure for creating and configuring table-view cell objects in nib files.

Selection management is also important with selection lists. There are two kinds of selection lists:

- Exclusive lists where only one row is permitted the checkmark
- Inclusive lists where more than one row can have a checkmark

Listing 6-3 illustrates one approach to managing an exclusive selection list. It first deselects the currently selected row and returns if the same row is selected; otherwise it sets the checkmark accessory type on the newly selected row and removes the checkmark on the previously selected row

Listing 6-3 Managing a selection list—exclusive list

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    [tableView deselectRowAtIndexPath:indexPath animated:NO];
    NSInteger catIndex = [taskCategories objectAtIndex:self.currentCategory];
    if (catIndex == indexPath.row) {
        return;
    }
    NSIndexPath *oldIndexPath = [NSIndexPath indexPathForRow:catIndex inSection:0];

    UITableViewCell *newCell = [tableView cellForRowAtIndexPath:indexPath];
    if (newCell.accessoryType == UITableViewCellAccessoryNone) {
        newCell.accessoryType = UITableViewCellAccessoryCheckmark;
        self.currentCategory = [taskCategories objectAtIndex:indexPath.row];
    }

    UITableViewCell *oldCell = [tableView cellForRowAtIndexPath:oldIndexPath];
    if (oldCell.accessoryType == UITableViewCellAccessoryCheckmark) {
        oldCell.accessoryType = UITableViewCellAccessoryNone;
    }
}
```

Listing 6-4 illustrates how to manage a inclusive selection list. As the comments in this example indicate, when the delegate adds a checkmark to a row or removes one, it typically also sets or unsets any associated model-object attribute.

Listing 6-4 Managing a selection list—inclusive list

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)newIndexPath {

    [tableView deselectRowAtIndexPath:[tableView indexPathForSelectedRow]
    animated:NO];
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:newIndexPath];
    if (cell.accessoryType == UITableViewCellAccessoryNone) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    }
}
```

```
        // Reflect selection in data model
    } else if (cell.accessoryType == UITableViewCellAccessoryCheckmark) {
        cell.accessoryType = UITableViewCellAccessoryNone;
        // Reflect deselection in data model
    }
}
```

In `tableView:didSelectRowAtIndexPath:` you should always deselect the currently selected row.

Programmatically Selecting and Scrolling

Occasionally the selection of a row originates within the application itself rather than from a tap in a table view. There could be an externally induced change in the data model. For example, the user adds a new person to an address book and then returns to the list of contacts; the application wants to scroll this list to the recently added person. For situations like these, you can use the `UITableView` methods `selectRowAtIndexPath:animated:scrollPosition:` and (if the row is already selected) `scrollToNearestSelectedRowAtScrollPosition:animated:`. You may also call `scrollToRowAtIndexPath:atScrollPosition:animated:` if you want to scroll to a specific row without selecting it.

The code in Listing 6-5 (somewhat whimsically) programmatically selects and scrolls to a row 20 rows away from the just-selected row using the `selectRowAtIndexPath:animated:scrollPosition:` method.



Listing 6-5 Programmatically selecting a row

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSIndexPath *scrollIndexPath;
    if (indexPath.row + 20 < [timeZoneNames count]) {
        scrollIndexPath = [NSIndexPath indexPathForRow:indexPath.row+20
inSection:indexPath.section];
    } else {
        scrollIndexPath = [NSIndexPath indexPathForRow:indexPath.row-20
inSection:indexPath.section];
    }
    [tableView selectRowAtIndexPath:scrollIndexPath animated:YES
                                scrollPosition:UITableViewScrollPositionMiddle];
}
```

```
}
```

Inserting and Deleting Rows and Sections

A table view has an editing mode as well as its normal (selection) mode. When a table view goes into editing mode, it displays the editing and reordering controls associated with its rows. The editing controls, which are in the left side of the row, allow the user to insert and delete rows in the table view. The editing controls have distinctive appearances:

	Deletion control
	Insertion control

When a table view enters editing mode and when users click an editing control, the table view sends a series of messages to its data source and delegate, but only if they implement these methods. These methods allow the data source and delegate to refine the appearance and behavior of rows in the table view; the messages also enable them to carry out the deletion or insertion operation.

Even if a table view is not in editing mode, you can insert or delete a number of rows or sections as a group and have those operations animated.

The first section below shows you how, when a table is in editing mode, to insert new rows and delete existing rows in a table view in response to user actions. The second section, [“Batch Insertion, Deletion, and Reloading of Rows and Sections”](#) (page 91), discusses how you can insert and delete multiple sections and rows animated as a group.

Note The procedure for reordering rows when in editing mode is described in [“Managing the Reordering of Rows”](#) (page 95).

Inserting and Deleting Rows in Editing Mode

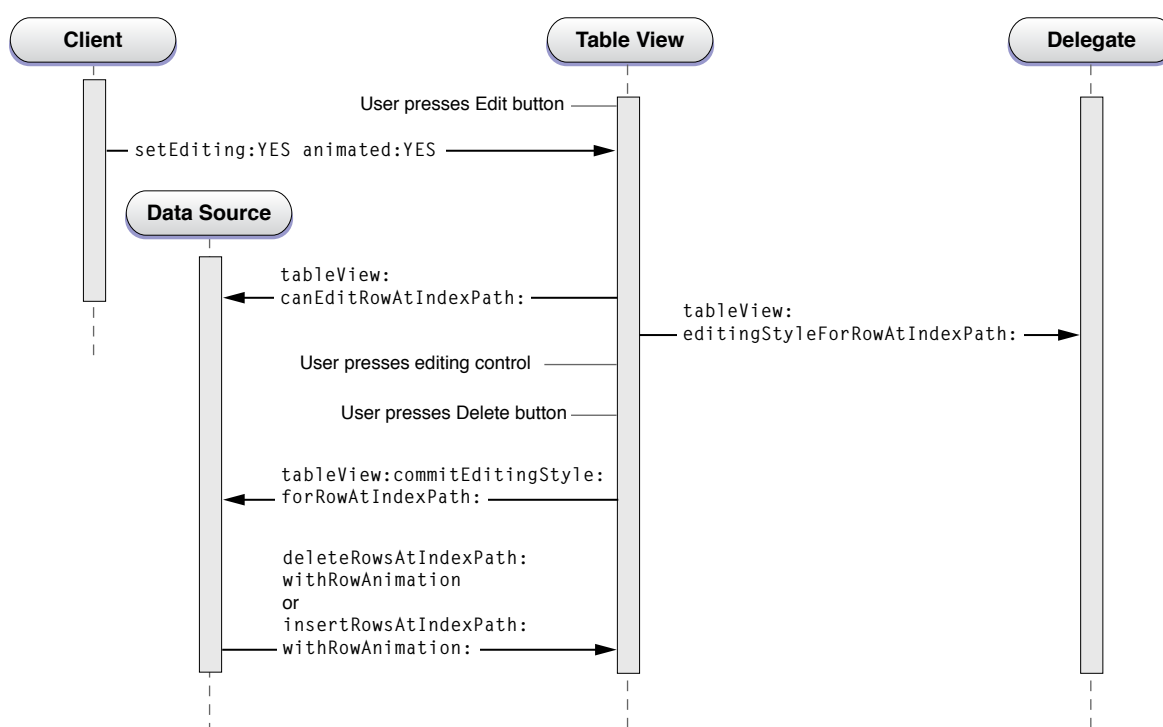
When a Table View is Edited

A table view goes into editing mode when it receives a `setEditing:animated:` message. Typically (but not necessarily) the message originates as an action message sent when the user taps an Edit button in the navigation bar. In editing mode, a table view displays any editing (and reordering) controls that its delegate has assigned to each row. The delegate assigns the controls as a result of returning the editing style for a row in the `tableView:editingStyleForRowAtIndexPath:` method.

Note If a `UIViewController` object is managing the table view, it automatically receives a `setEditing:animated:` message when the Edit button is tapped. In its implementation of this message, it can update button state or do any other kind of task before invoking the table view's version of the method.

When the table view receives `setEditing:animated:`, it sends the same message to the `UITableViewCell` object for each visible row. Then it sends a succession of messages to its data source and its delegate (if they implement the methods) as depicted in the diagram in Figure 7-1.

Figure 7-1 Calling sequence for inserting or deleting rows in a table view



After resending `setEditing:animated:` to the cells corresponding to the visible rows, the sequence of messages is as follows:

1. The table view invokes the `tableView:canEditRowAtIndexPath:` method if its data source implements it. This method allows the application to exclude rows in the table view from being edited even when their cell's `editingStyle` property indicates otherwise. Most applications do not need to implement this method.
2. The table view invokes the `tableView:editingStyleForRowAtIndexPath:` method if its delegate implements it. This method allows the application to specify a row's editing style and thus the editing control that the row displays.

At this point, the table view is fully in editing mode. It displays the insertion or deletion control for each eligible row.

3. The user taps an editing control (either the deletion control or the insertion control). If he or she taps a deletion control, a Delete button is displayed on the row. The user then taps that button to confirm the deletion.
4. The table view sends the `tableView:commitEditingStyle:forRowAtIndexPath:` message to the data source. Although this protocol method is marked as optional, the data source must implement it if it wants to insert or delete a row. It must do two things:
 - Send `deleteRowsAtIndexPaths:withRowAnimation:` or `insertRowsAtIndexPaths:withRowAnimation:` to the table view to direct it to adjust its presentation.
 - Update the corresponding data-model array by either deleting the referenced item from the array or adding an item to the array.

When the user swipes across a row to display the Delete button for that row, there is a variation in the calling sequence diagrammed in [Figure 7-1](#) (page 86). When the user swipes a row to delete it, the table view first checks to see if its data source has implemented the `tableView:commitEditingStyle:forRowAtIndexPath:` method; if that is so, it sends `setEditing:animated:` to itself and enters editing mode. In this “swipe to delete” mode, the table view does not display the editing and reordering controls. Because this is a user-driven event, it also brackets the messages to the delegate inside of two other messages: `tableView:willBeginEditingRowAtIndexPath:` and `tableView:didEndEditingRowAtIndexPath:`. By implementing these methods, the delegate can update the appearance of the table view appropriately.

Note The data source should not call `setEditing:animated:` from within its implementation of `tableView:commitEditingStyle:forRowAtIndexPath:`. If for some reason it must, it should invoke it after a delay by using the `performSelector:withObject:afterDelay:` method.

Although you can use an insertion control as the trigger to insert a new row in a table view, an alternative approach is to have an Add (or plus sign) button in the navigation bar. Tapping the button sends an action message to the view controller, which overlays the table view with a modal view for entering the new item. Once the item is entered, the controller adds it to the data-model array and reloads the table. “[An Example of Adding a Table-View Row](#)” (page 89) discusses this approach.

An Example of Deleting a Table-View Row

This section gives a guided tour through the parts of a project that work together to set up a table view for editing mode and delete rows from it. This project uses the navigation controller and view controller architecture to manage its table views. In its `loadView` method, the custom view controller creates the table view and sets itself to be the data source and delegate. It also sets the right item of the navigation bar to be the standard Edit button.

```
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

This button is preconfigured to send `setEditing:animated:` to the view controller when tapped; it toggles the button title (between Edit and Done) and the Boolean `editing` parameter on alternating taps. In its implementation of the method, as shown in Listing 7-1, the view controller invokes the superclass invocation of the method, sends the same message to the table view, and updates the enabled state of the other button in the navigation bar (a plus-sign button, for adding items).

Listing 7-1 View controller responding to `setEditing:animated:`

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated {
    [super setEditing:editing animated:animated];
    [tableView setEditing:editing animated:YES];
    if (editing) {
        addButton.enabled = NO;
    } else {
        addButton.enabled = YES;
    }
}
```

When its table view enters editing mode, the view controller specifies a deletion control for every row except the last, which has an insertion control. It does this in its implementation of the `tableView:editingStyleForRowAtIndexPath:` method (Listing 7-2).

Listing 7-2 Customizing the editing style of rows

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
*)[[UIApplication sharedApplication] delegate];
    if (indexPath.row == [controller countOfList]-1) {
```



```
        return UITableViewCellEditingStyleInsert;
    } else {
        return UITableViewCellEditingStyleDelete;
    }
}
```

The user taps the deletion control in a row and the view controller receives a `tableView:commitEditingStyle:forRowAtIndexPath:` message from the table view. As shown in Listing 7-3, it handles this message by removing the item corresponding to the row from a model array and sending `deleteRowsAtIndexPaths:withRowAnimation:` to the table view.

Listing 7-3 Updating the data-model array and deleting the row

```
- (void)tableView:(UITableView *)tv
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    // If row is deleted, remove it from the list.
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
*)[[UIApplication sharedApplication] delegate];
        [controller removeObjectFromListAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

An Example of Adding a Table-View Row

This section shows project code that inserts a row in a table view. Instead of using the insertion control as the trigger for inserting a row, it uses an Add button (visually a plus sign) in the navigation bar above the table view. This code also is based on the navigation controller and view controller architecture. In its `loadView` method implementation, the view controller assigns the Add button as the right-side item of the navigation bar using the code shown in Listing 7-4.

Listing 7-4 Adding an Add button to the navigation bar

```
addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(addItem:)];
```

```
self.navigationItem.rightBarButtonItem = addButton;
```

Note that the view controller sets the control states for the title as well as the action selector and the target object. When the user taps the Add button, the `addItem:` message is sent to the target (the view controller). It responds as shown in Listing 7-5. It creates a navigation controller with a single view controller whose view is put onscreen modally—it animates upward to overlay the table view. The `presentModalViewController:animated:` method to do this.

Listing 7-5 Responding to a tap on the Add button

```
- (void)addItem:sender {
    if (itemInputController == nil) {
        itemInputController = [[ItemInputController alloc] init];
    }
    UINavigationController *navigationController = [[UINavigationController alloc]
initWithRootViewController:itemInputController];
    [[self navigationController] presentModalViewController:navigationController
animated:YES];
    [navigationController release];
}
```

The modal, or overlay, view consists of a single custom text field. The user enters text for the new table-view item and then taps a Save button. This button sends a `save:` action message to its target: the view controller for the modal view. As shown in Listing 7-6, the view controller extracts the string value from the text field and updates the application's data-model array with it.

Listing 7-6 Adding the new item to the data-model array

```
- (void)save:sender {

    UITextField *textField = [(EditableTableViewTextField *)[tableView
cellForRowAtIndexPath:[NSIndexPath indexPathForRow:0 inSection:0]] textField];

    SimpleEditableListAppDelegate *controller = (SimpleEditableListAppDelegate
*)[[UIApplication sharedApplication] delegate];
    NSString *newItem = textField.text;
    if (newItem != nil) {
        [controller insertObject:newItem inListAtIndex:[controller countOfList]];
    }
}
```

```
    }  
    [self dismissModalViewControllerAnimated:YES];  
}
```

After the modal view is dismissed the table view is reloaded, and it now reflects the added item.

Batch Insertion, Deletion, and Reloading of Rows and Sections

The `UITableView` class allows you to insert, delete, and reload a group of rows or sections at one time, animating the operations simultaneously in specified ways. The eight methods shown in Listing 7-7 pertain to batch insertion and deletion. Note that you can call these insertion and deletion methods outside of an animation block (as you do in the data source method `tableView:commitEditingStyle:forRowAtIndexPath:` as discussed in [“Inserting and Deleting Rows in Editing Mode”](#) (page 85)).

Listing 7-7 Batch insertion and deletion methods

```
- (void)beginUpdates;  
- (void)endUpdates;  
  
- (void)insertSections:(NSIndexSet *)sections  
withRowAnimation:(UITableViewRowAnimation)animation;  
- (void)deleteSections:(NSIndexSet *)sections  
withRowAnimation:(UITableViewRowAnimation)animation;  
- (void)reloadSections:(NSIndexSet *)sections  
withRowAnimation:(UITableViewRowAnimation)animation;  
  
- (void)insertRowsAtIndexPaths:(NSArray *)indexPaths withRowAnimation:  
(UITableViewRowAnimation)animation;  
- (void)deleteRowsAtIndexPaths:(NSArray *)indexPaths withRowAnimation:  
(UITableViewRowAnimation)animation;  
- (void)reloadRowsAtIndexPaths:(NSArray *)indexPaths  
withRowAnimation:(UITableViewRowAnimation)animation;
```

Note The `reloadSections:withRowAnimation:` and `reloadRowsAtIndexPaths:withRowAnimation:` methods, which were introduced in iOS 3.0, allow you to request the table view to reload the data for specific sections and rows instead of loading the entire visible table view by calling `reloadData`.

To animate a batch insertion, deletion, and reloading of rows and sections, call the corresponding methods within an animation block defined by successive calls to `beginUpdates` and `endUpdates`. If you don't call the insertion, deletion, and reloading methods within this block, row and section indexes may be invalid. Calls to `beginUpdates` and `endUpdates` can be nested; all indexes are treated as if there were only the outer update block.

At the conclusion of a block—that is, after `endUpdates` returns—the table view queries its data source and delegate as usual for row and section data. Thus the collection objects backing the table view should be updated to reflect the new or removed rows or sections.

An Example of Batched Insertion and Deletion Operations

To insert and delete a group of rows and sections in a table view, first prepare the array (or arrays) that are the source of data for the sections and rows. After rows and sections are deleted and inserted, the resulting rows and sections are populated from this data store.

Next, call the `beginUpdates` method, followed by invocations of `insertRowsAtIndexPaths:withRowAnimation:`, `deleteRowsAtIndexPaths:withRowAnimation:`, `insertSections:withRowAnimation:`, or `deleteSections:withRowAnimation:`. Conclude the animation block by calling `endUpdates`. Listing 7-8 illustrates this procedure.

Listing 7-8 Inserting and deleting a block of rows in a table view

```
- (IBAction)insertAndDeleteRows:(id)sender {
    // original rows: Arizona, California, Delaware, New Jersey, Washington

    [states removeObjectAtIndex:4]; // Washington
    [states removeObjectAtIndex:2]; // Delaware
    [states insertObject:@"Alaska" atIndex:0];
    [states insertObject:@"Georgia" atIndex:3];
    [states insertObject:@"Virginia" atIndex:5];

    NSArray *deleteIndexPaths = [NSArray arrayWithObjects:
```

```
        [NSIndexPath indexPathForRow:2 inSection:0],
        [NSIndexPath indexPathForRow:4 inSection:0],
        nil];

    NSArray *insertIndexPaths = [NSArray arrayWithObjects:
        [NSIndexPath indexPathForRow:0 inSection:0],
        [NSIndexPath indexPathForRow:3 inSection:0],
        [NSIndexPath indexPathForRow:5 inSection:0],
        nil];

    UITableView *tv = (UITableView *)self.view;

    [tv beginUpdates];
    [tv insertRowsAtIndexPaths:insertIndexPaths
     withRowAnimation:UITableViewRowAnimationRight];
    [tv deleteRowsAtIndexPaths:deleteIndexPaths
     withRowAnimation:UITableViewRowAnimationFade];
    [tv endUpdates];

    // ending rows: Alaska, Arizona, California, Georgia, New Jersey, Virginia
}
```

This example removes two strings from an array (and their corresponding rows) and inserts three strings into the array (along with their corresponding rows). The next section, “Ordering of Operations and Index Paths,” explains particular aspects of the row (or section) insertion and deletion behavior.

Ordering of Operations and Index Paths

You might have noticed something in the code shown in Listing 7-8 that seems peculiar. The code calls the `deleteRowsAtIndexPaths:withRowAnimation:` method after it calls `insertRowsAtIndexPaths:withRowAnimation:`. However, this is not the order in which `UITableView` completes the operations. It defers any insertions of rows or sections until after it has handled the deletions of rows or sections. The table view behaves the same way with reloading methods called inside an update block—the reload takes place with respect to the indexes of rows and sections before the animation block is executed. This behavior happens regardless of the ordering of the insertion, deletion, and reloading method calls.

Deletion and reloading operations within an animation block specify which rows and sections in the original table should be removed or reloaded; insertions specify which rows and sections should be added to the resulting table. The index paths used to identify sections and rows follow this model. Inserting or removing

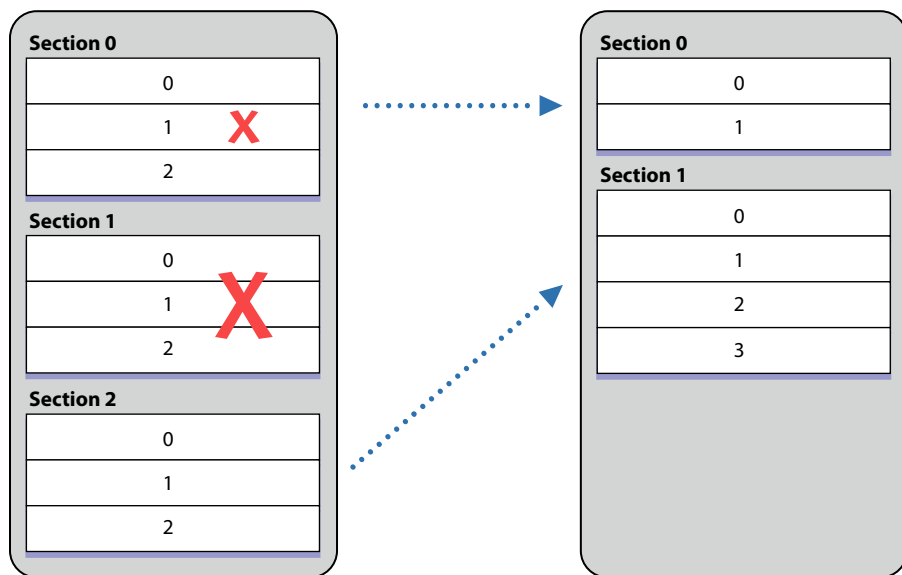
an item in a mutable array, on the other hand, may affect the array index used for the successive insertion or removal operation; for example, if you insert an item at a certain index, the indexes of all subsequent items in the array are incremented.

An example is useful here. Say you have a table view with three sections, each with three rows. Then you implement the following animation block:

1. Begin updates.
2. Delete row at index 1 of section at index 0.
3. Delete section at index 1.
4. Insert row at index 1 of section at index 1.
5. End updates.

Figure 7-2 illustrates what takes place after the animation block concludes.

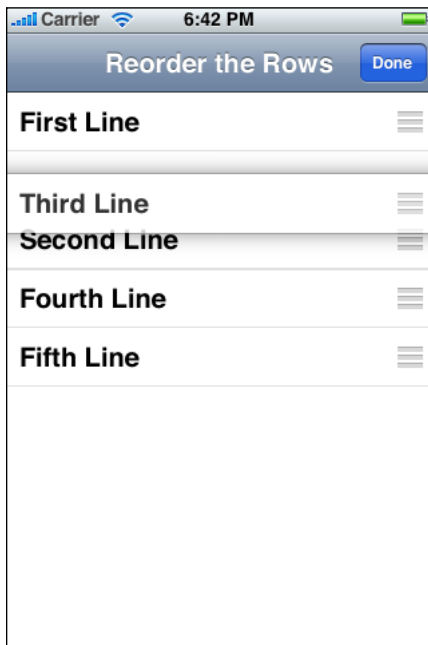
Figure 7-2 Deletion of section and row and insertion of row



Managing the Reordering of Rows

A table view has an editing mode as well as its normal (selection) mode. When a table view goes into editing mode, it displays the editing and reordering controls associated with its rows. The reordering control allows the user to move a row to a different location in the table. As shown in Figure 8-1, the reordering control appears on the right side of the row.

Figure 8-1 Reordering a row



When a table view enters editing mode and when users drag a reordering control, the table view sends a series of messages to its data source and delegate, but only if they implement these methods. These methods allow the data source and delegate to restrict whether and where a row can be moved as well to carry out the actual move operation. The following sections show you how to move rows around in a table view.

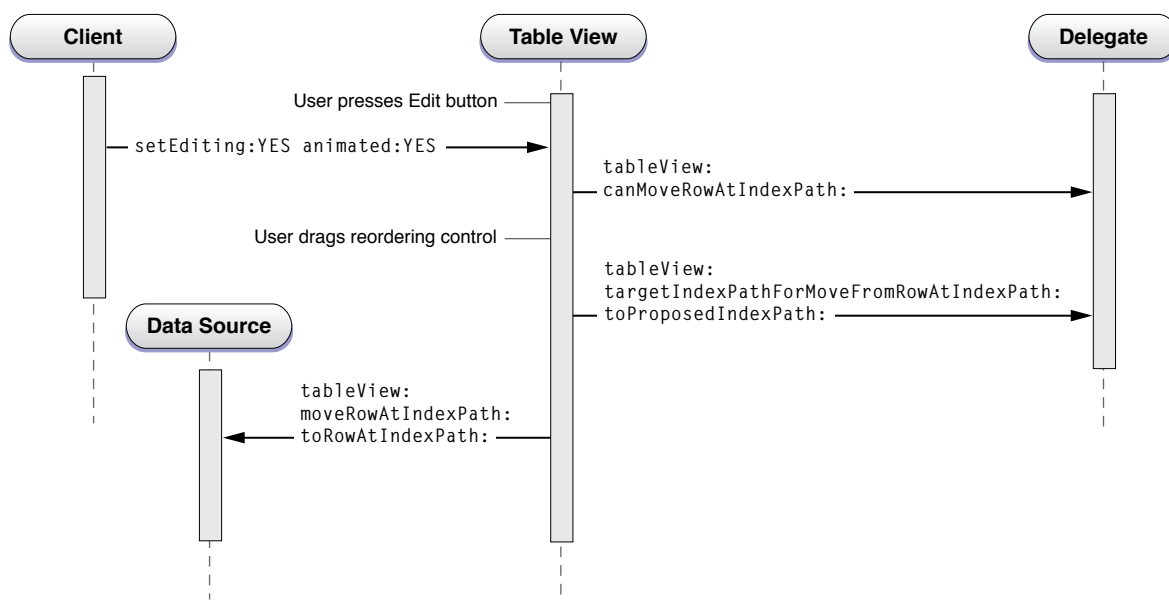
What Happens When a Row is Relocated

A table view goes into editing mode when it receives a `setEditing:animated:` message. Typically (but not necessarily) the message originates as an action message sent when the user taps an Edit button in the navigation bar. In editing mode, a table view displays any reordering (and editing) controls that its delegate has assigned to each row. The delegate assigns the controls in `tableView:cellForRowAtIndexPath:` by setting the `showsReorderControl` property of `UITableViewCell` objects to YES.

Note If a `UIViewController` object is managing the table view, it automatically receives a `setEditing:animated:` message when the Edit button is tapped. `UITableViewController`, a subclass of `UIViewController`, implements this method to update button state and invoke the table view's version of the method. If you are using `UIViewController` to manage a table view, you need to implement the same behavior.

When the table view receives `setEditing:animated:`, it sends the same message to the `UITableViewCell` object for each visible row. Then it sends a succession of messages to its data source and its delegate (if they implement the methods) as depicted in the diagram in Figure 8-2.

Figure 8-2 Calling sequence for reordering a row in a table view



When the table view receives the `setEditing:animated:` message, it resends the same message to the cell objects corresponding to its visible rows. After that, the sequence of messages is as follows:

1. The table view sends a `tableView:canMoveRowAtIndexPath:` message to its data source (if it implements the method). In this method the delegate may selectively exclude certain rows from showing the reordering control.
2. The user drags a row by its reordering control up or down the table view. As the dragged row hovers over a part of the table view, the underlying row slides downward to show where the destination would be.
3. Every time the dragged row is over a destination, the table view sends `tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:` to its delegate (if it implements the method). In this method the delegate may reject the current destination for the dragged row and specify an alternative one.
4. The table view sends `tableView:moveRowAtIndexPath:toIndexPath:` to its data source (if it implements the method). In this method the data source updates the data-model array that is the source of items for the table view, moving the item to a different location in the array.

Examples of Moving a Row

This section comments on some sample code that illustrates the reordering steps enumerated in [“What Happens When a Row is Relocated”](#) (page 96). Listing 8-1 shows an implementation of `tableView:canMoveRowAtIndexPath:` that excludes the first row in the table view from being reordered (this row does not have a reordering control).

Listing 8-1 Excluding a row from relocation

```
- (BOOL)tableView:(UITableView *)tableView canMoveRowAtIndexPath:(NSIndexPath *)indexPath {  
    if (indexPath.row == 0) // Don't move the first row  
        return NO;  
  
    return YES;  
}
```

When the user finishes dragging a row, it slides into its destination in the table view, which sends `tableView:moveRowAtIndexPath:toIndexPath:` to its data source. Listing 8-2 shows an implementation of this method. Note that it retains the data item fetched from the array (the item to be relocated) because the item is automatically released when it is removed from the array.

Listing 8-2 Updating the data-model array for the relocated row

```
- (void)tableView:(UITableView *)tableView moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath toIndexPath:(NSIndexPath *)destinationIndexPath {
    NSString *stringToMove = [[self.reorderingRows objectAtIndex:sourceIndexPath.row] retain];
    [self.reorderingRows removeObjectAtIndex:sourceIndexPath.row];
    [self.reorderingRows insertObject:stringToMove atIndex:destinationIndexPath.row];
    [stringToMove release];
}
```

The delegate can also retarget the proposed destination for a move to another row by implementing the `tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:` method. The following example restricts rows to relocation in their own group and prevents moves to the last row of a group (which is reserved for the add-item placeholder).

Listing 8-3 Retargeting the destination row of a move operation

```
- (NSIndexPath *)tableView:(UITableView *)tableView
    targetIndexPathForMoveFromRowAtIndexPath:(NSIndexPath *)sourceIndexPath
    toProposedIndexPath:(NSIndexPath *)proposedDestinationIndexPath {
    NSDictionary *section = [data objectAtIndex:sourceIndexPath.section];
    NSUInteger sectionCount = [[section valueForKey:@"content"] count];
    if (sourceIndexPath.section != proposedDestinationIndexPath.section) {
        NSUInteger rowInSourceSection =
            (sourceIndexPath.section > proposedDestinationIndexPath.section) ?
            0 : sectionCount - 1;
        return [NSIndexPath indexPathForRow:rowInSourceSection
            inSection:sourceIndexPath.section];
    } else if (proposedDestinationIndexPath.row >= sectionCount) {
        return [NSIndexPath indexPathForRow:sectionCount - 1
            inSection:sourceIndexPath.section];
    }
    // Allow the proposed destination.
    return proposedDestinationIndexPath;
}
```

Document Revision History

This table describes the changes to *Table View Programming Guide for iOS*.

Date	Notes
2011-01-05	Made some minor corrections.
2010-09-14	Made several minor corrections.
2010-08-03	States now that <code>beginUpdates...endUpdates</code> calls can be nested.
2010-07-08	Changed the title from "Table View Programming Guide for iPhone OS."
2010-05-20	Made many minor corrections in diagrams and text. Added description of reload behavior in batch updates.
2010-03-24	Made the introduction more informative.
2009-08-19	Explained how to set the background color of cells, emphasized purpose of <code>tableView:willDisplayCell:forRowAtIndexPath:</code> , and made minor corrections.
2009-05-28	Updated to describe 3.0 API, especially predefined cell styles and related properties. It also describes the use of nib files with table views and table-view cells and includes an updated chapter on view controllers and design patterns and strategies.
2008-10-15	Warned against calling <code>setEditing:animated:</code> when in <code>tableView:commitEditingStyle:forRowAtIndexPath:</code> . Updated illustration.
2008-09-09	Added section on batch insertions and deletions, added related classes to TOC frame, added guidelines on clearing selection, and made minor corrections.
2008-06-25	First version of this document.



Apple Inc.

© 2011 Apple Inc.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Apple Inc.

1 Infinite Loop

Cupertino, CA 95014

408-996-1010

Apple, the Apple logo, Cocoa, Cocoa Touch, iPad, iPhone, iPod, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.