Table View Programming Guide for iOS

Developer

Contents

Figures and Listings

Table View Styles and Accessory Views

Table views come in distinctive styles that are suitable for specific purposes. In addition, the UIKit framework 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.

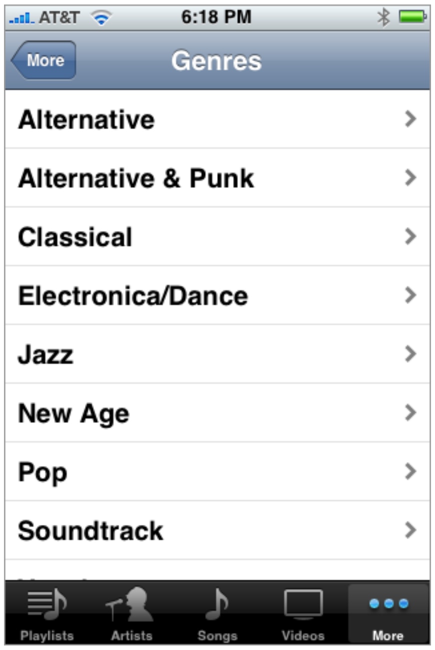
Table View Styles

There are two major styles of table views: plain and grouped. The two styles are distinguished mainly by appearance.

Plain Table Views

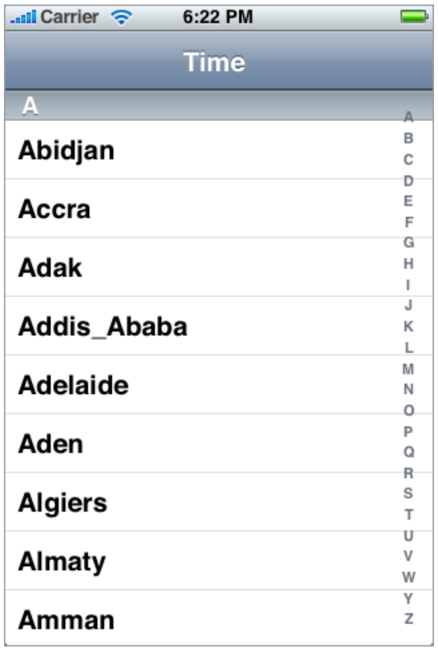
A table view in the plain (or regular) style displays rows that stretch across the screen and have a creamy white background (see Figure 1-1). A plain table view can have one or more sections, sections can have one or more rows, and each section can have its own header or footer title. (A header or footer 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



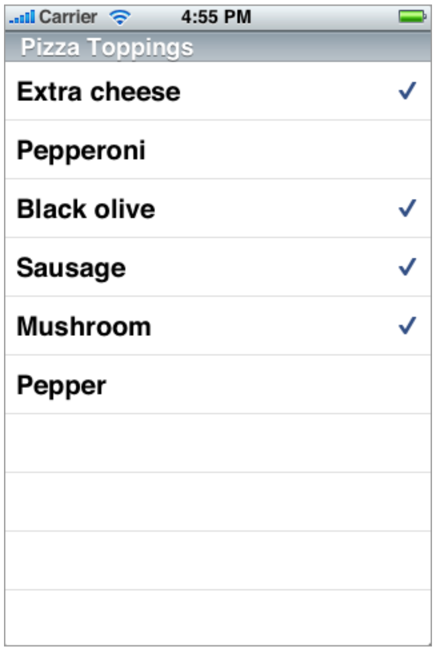
A variation of plain table views associates an index with sections for quick navigation; Figure 1-2 shows an example of this kind of table view, which is called an **indexed list**. The index runs down the right edge of the table view. Entries in the index correspond to section header titles. Touching 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 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 indexed list



The simplest kind of table view is a selection list (see Figure 1-3). A selection list is a plain 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).

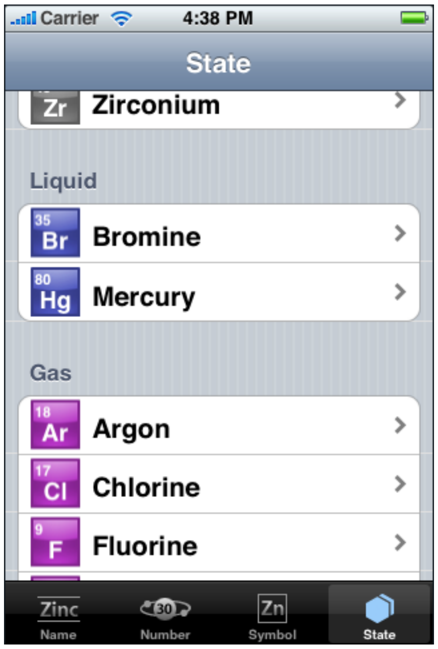
Figure 1-3 A table view configured as a selection list



Grouped Table Views

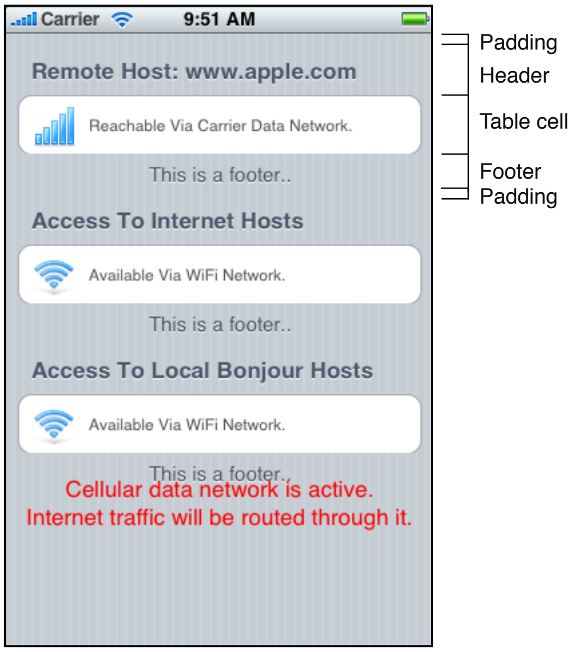
A grouped table view also displays a list of information, but it groups related rows in visually distinct sections. As shown in Figure 1-4, each section has rounded corners and by default appears against a bluish-gray background. Each section may have text or an image for its header or footer to provide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

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


Carrier
9:51 AM
Remote Host: www.apple.com
Reachable Via Carrier Data Network.
This is a footer..
Access To Internet Hosts
Available Via WiFi Network.
This is a footer..
Access To Local Bonjour Hosts
Available Via WiFi Network.
This is a footer..
Cellular data network is active.
Internet traffic will be routed through it.
Padding
Header
Table cell
Footer
Padding

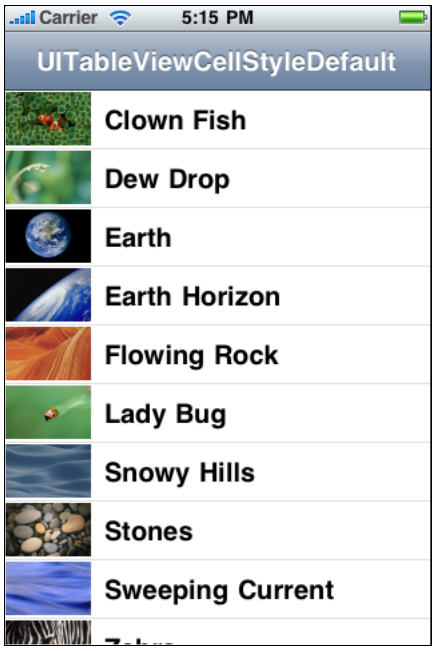
On iPad devices, a grouped table view automatically gets wider margins when the table view itself is wide.

Standard Styles for Table View Cells

In addition to defining two styles of table views, the UIKit framework 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 want, 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 51).

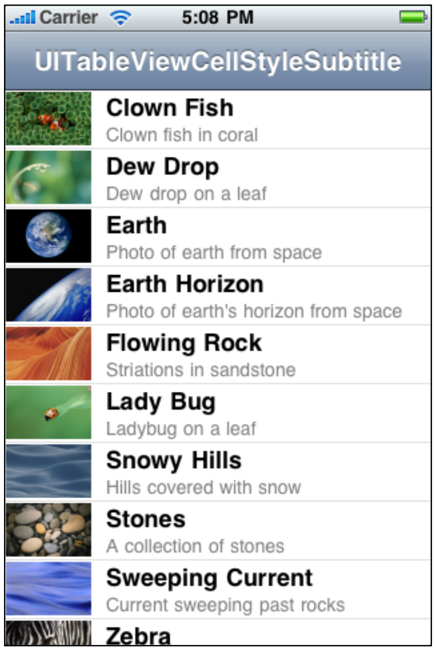
The default style for table view rows uses a simple cell style that has a single title and an optional image (Figure 1-6). This style is associated with the `UITableViewCellStyleDefault` constant.

Figure 1-6 Default table row style



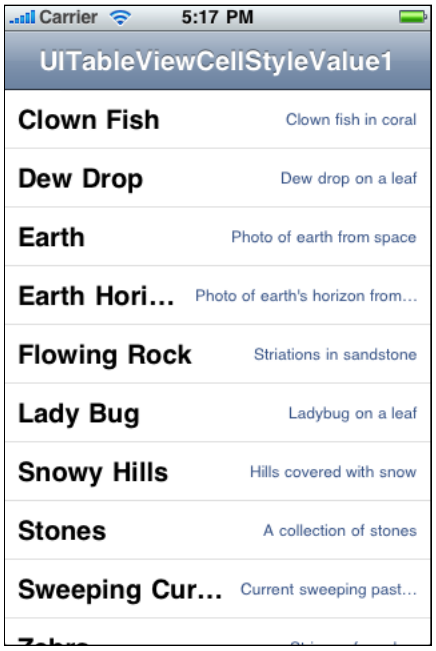
The cell style for the rows in Figure 1-7 left-aligns the main title and puts a gray subtitle under it. It also permits an image in the default image location. This style is associated with the `UITableViewCellStyleSubtitle` constant.

Figure 1-7 Table row style with a subtitle under the title



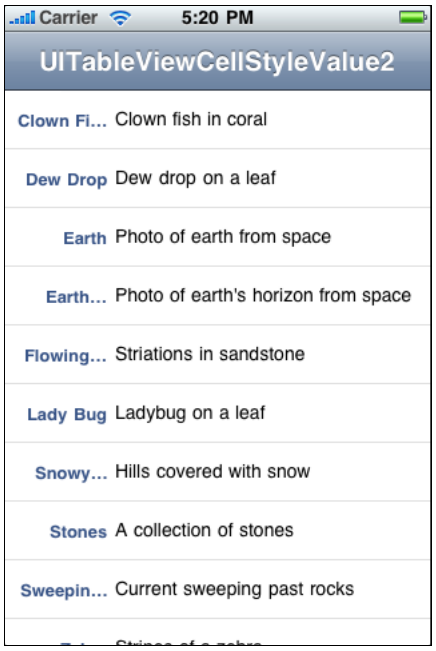
The cell style for the rows in Figure 1-8 left-aligns the main title. It puts the subtitle in blue text and right-aligns it on the right side of the row. Images are not permitted. This style is used in the Settings app, where the subtitle indicates the current setting for a preference. It is associated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

Figure 1-8 Table row style with a 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 app and is associated with the `UITableViewCellStyleValue2` constant.

Figure 1-9 Table row style in Contacts format



UITableViewCellStyleValue2

Clown Fi...	Clown fish in coral
Dew Drop	Dew drop on a leaf
Earth	Photo of earth from space
Earth...	Photo of earth's horizon from space
Flowing...	Striations in sandstone
Lady Bug	Ladybug on a leaf
Snowy...	Hills covered with snow
Stones	A collection of stones
Sweepin...	Current sweeping past rocks

Accessory Views

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

Standard accessory views	Description
›	**Disclosure indicator**—`UITableViewCellAccessoryDisclosure-Indicator`. 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**—`UITableViewCellAccessoryDetail-DisclosureButton`. 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
✓	**Checkmark**—`UITableViewCellAccessoryCheckmark`. 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 views, you may specify a control (for example, a switch) or a custom view as the accessory view.

Overview of the Table View API

The table view programming interface includes several UIKit classes, two formal protocols, and a category added to a Foundation framework class.

Table View

A table view itself is an instance of the `UITableView` class. You use its methods to configure the appearance of the table view—for example, specifying the default height of rows or providing a subview 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 the `UIScrollView` class, 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.

Table View Controller

The `UITableViewController` class manages a table view and adds support for many standard table-related behaviors such as selection management, row editing, table configuration, and others. This additional support is there to minimize the amount of code you have to write to create and initialize your table-based interface. You don't use this class directly—instead you subclass `UITableViewController` to add custom behaviors.

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 app'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 usually a custom subclass of `UITableViewController`. (See "Navigating a Data Hierarchy with Table Views" (page 21) for further information.)

The data source adopts the `UITableViewDataSource` protocol. `UITableViewDataSource` has two required methods. The `tableView:numberOfRowsInSection:` method tells the table view how many rows to display in each section, and the `tableView:cellForRowAtIndexPath:` method 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. This protocol 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 a table.

An app can make use of the 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 table view methods use index paths as parameters or return values. 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` with methods that return key paths, locate rows in sections, and construct `NSIndexPath` objects from row and section indexes. For more information, see *NSIndexPath UIKit Additions* .

Table View Cells

As noted in "Data Source and Delegate" (page 19), 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 for managing cell selection and editing, managing accessory views, and configuring the cell. You can instantiate cells directly 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. 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 customize the appearance and behavior of table view cells. These approaches are all discussed in "A Closer Look at Table View Cells" (page 51).

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 apps.

Hierarchical Data Models and Table Views

For a navigation-based app, you typically design your app data as a graph of model objects that is sometimes referred to as the app's **data model**. You can then implement the model layer of your app using various mechanisms or technologies, including Core Data, property lists, or archives of custom objects. Regardless of the approach, the traversal of your app's data model follows patterns that are common to all navigation-based apps. 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: 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 app factors its classes and objects in a way that conforms to the Model-View-Controller (MVC) design pattern. The app'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.

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 that presents a detail view of that item.

A model object may also have relationships with other model objects. 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. As collections nest in other collections, your data model can acquire hierarchical depth.

Table Views and the Data Model

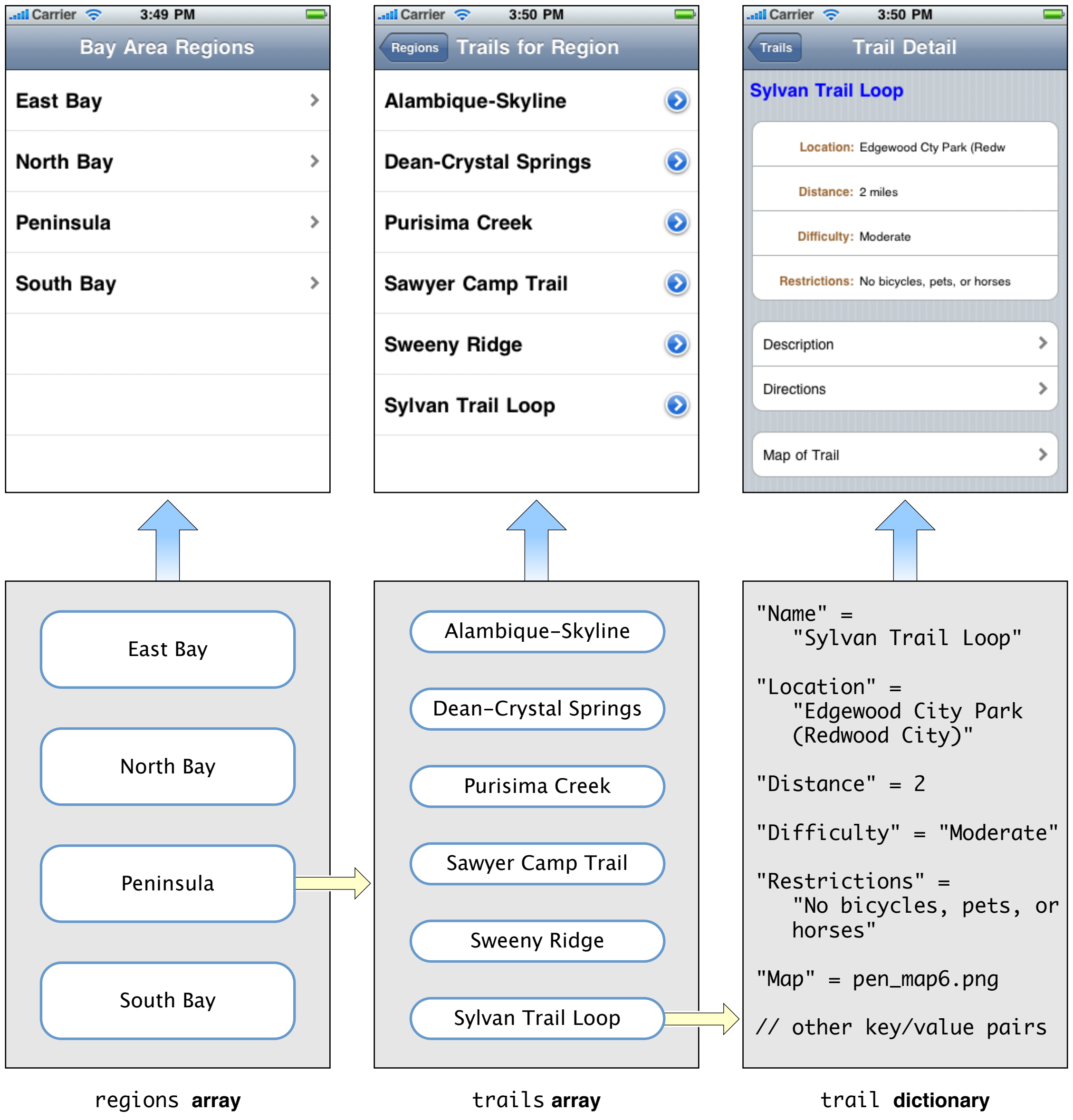
The rows of a plain table view are typically backed by collection objects of the app's data model; these objects are usually arrays. Arrays contain strings or other elements that a table view can use when displaying row content. When you create a table view (described in "Creating and Configuring a Table View" (page 32)), it immediately queries its data source for its dimensions—that is, it requests the number of sections and the number of rows per section—and then asks for the content of each row. The data source fetches this content 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 being used as the table view's source of data.

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

In the sequence of table views in Figure 3-1, the top level of the data hierarchy 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 describes that trail using a grouped table view.

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



```
"Name" =
   "Sylvan Trail Loop"

"Location" =
   "Edgewood City Park
   (Redwood City)"

"Distance" = 2

"Difficulty" = "Moderate"

"Restrictions" =
   "No bicycles, pets, or
   horses"

"Map" = pen_map6.png

// other key/value pairs
```

Note: You could easily redesign the app in Figure 3-1 (page 23) to have only two table views. The first table view would be an indexed list of trails by region. The second table view would display the detail for a selected trail.

View Controllers and Navigation-Based Apps

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 app uses those views to present successive levels of its data hierarchy. This section describes how two subclasses of `UIViewController`, navigation controllers and table view controllers, present and manage a succession of table views.

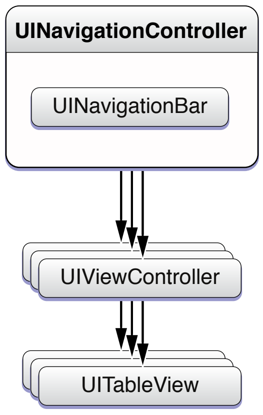
Note: This section gives an overview of view controllers to provide some background for the coding tasks discussed later in this document. To learn about view controllers in depth, see *View Controller Programming Guide for iOS*.

Navigation Controllers

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. After 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 navigation controller maintains a stack of view controllers, one for each of the table views displayed (see Figure 3-2). It begins with what’s known as the **root view controller**. When the user taps a row of the table view (often on 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 that is now at the top of the stack.

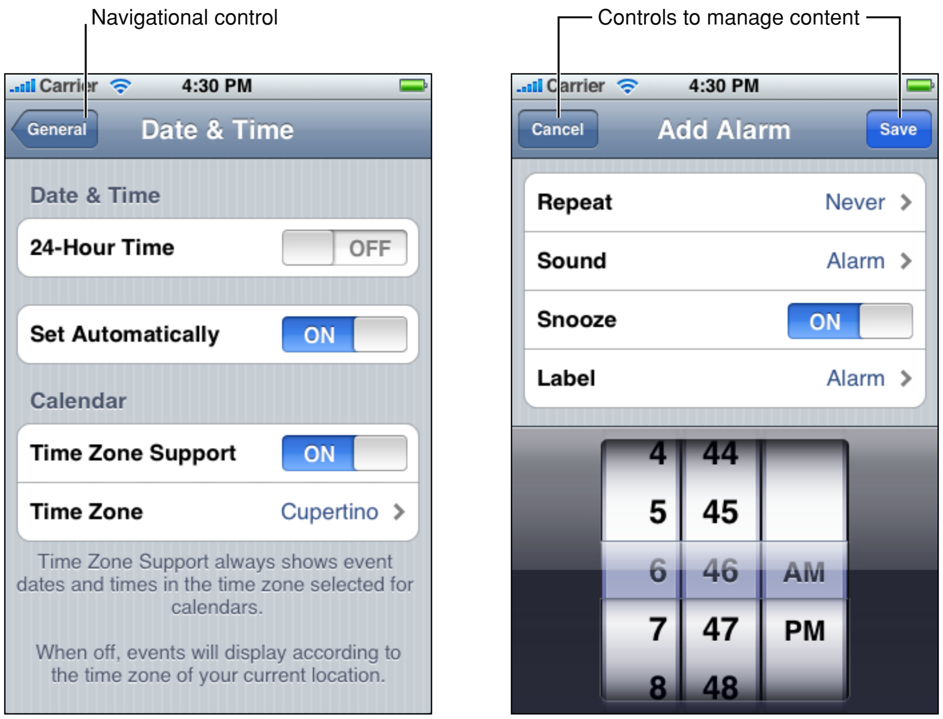
Figure 3-2 Navigation controller and view controllers in a navigation-based app



Navigation Bars

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 detailed 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 3-3).

Figure 3-3 Navigation bars and common control items



Date & Time
24-Hour Time OFF
Set Automatically ON
Calendar
Time Zone Support ON
Time Zone Cupertino
Time Zone Support always shows event dates and times in the time zone selected for calendars.
When off, events will display according to the time zone of your current location.

Repeat Never
Sound Alarm
Snooze ON
Label Alarm
4 44
5 45
6 46 AM
7 47 PM
8 48

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.

The `UIViewController` class includes methods that let view controllers access and set the navigation items displayed in the navigation bar for the currently displayed table view. This class also declares a `title` property through which you can set the title of the navigation bar for the current table view.

Table View Controllers

Although you could manage a table view 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 created a direct subclass of `UIViewController` to manage a table view.

The recommended way to create a table view controller is to specify it in a storyboard. The associated table view is loaded from the storyboard, along with the table view's attributes, size, and autoresizing characteristics. The table view controller sets itself as the data source and the delegate of the table view.

Note: You can create a table view controller programmatically by allocating memory for it and initializing it with the `initWithStyle:` method, passing in either `UITableViewStylePlain` or `UITableViewStyleGrouped` for the required table view style.

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, and then gives the table view the data to display in each row. This process is described in "Creating and Configuring a Table View" (page 32).

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: 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. It also 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, only 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 App

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. In the rest of this document, you learn 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 app.

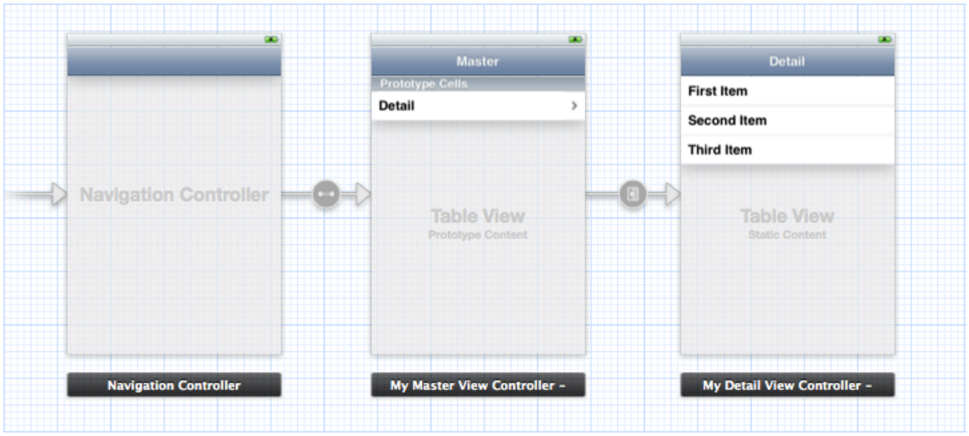
Note: This section summarizes view-controller and navigation-controller tasks, with a focus on table views. For a thorough discussion of view controllers and navigation controllers, including the complete details of their implementation, see *View Controller Programming Guide for iOS* and *View Controller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

At this point, let's assume that a table view managed by a table view controller presents a list to the user. How does the app 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 button.) The delegate 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. A storyboard provides the specification that allows UIKit to perform most of this work for you.

Storyboards represent the screens in an app and the transitions between them. The storyboard in a basic app may contain just a few screens, but a more complex app might have multiple storyboards, each of which represents a different subset of its screens. The storyboard example in Figure 3-4 presents a graphical representation of each scene, its contents, and its connections.

Figure 3-4 A storyboard with two table view controllers



A **scene** represents an onscreen content area that is managed by a view controller. (In the context of a storyboard, scene and view controller are synonymous terms.) The leftmost scene in the default storyboard represents a navigation controller. A navigation controller is a container view controller because, in addition to its views, it also manages a set of other view controllers. For example, the navigation controller in Figure 3-4 (page 29) manages the master and detail view controllers, in addition to the navigation bar and the back button that you see when you run the app.

A **relationship** is a type of connection between scenes. In Figure 3-4, there is a relationship between the navigation controller and the master scene. In this case, the relationship represents the containment of the master and detail scenes by the navigation controller. When the app runs, the navigation controller automatically loads the master scene and displays the navigation bar at the top of the screen.

A **segue** represents a transition from one scene (called the source) to the next scene (called the destination). For example, in Figure 3-4, the master scene is the source and the detail scene is the destination. When you select the Detail item in the master list, you trigger a segue from the source to the destination. In this case, the segue is a push segue, which means that the destination scene slides over the source scene from right to left. As the detail screen is revealed, a back button appears at the left end of the navigation bar, titled with the previous screen's title (in this case, "Master"). The back button is provided automatically by the navigation controller that manages the master-detail hierarchy.

Storyboards make it easy to pass data from one scene to another via the `prepareForSegue:sender:` method of the `UIViewController` class. This method is called when the first scene (the source) is about to transition to the next scene (the destination). The source view controller can implement `prepareForSegue:sender:` to perform setup tasks, such as passing information to the destination view controller about what it should display in its table view. Listing 3-1 shows one implementation of this method.

Listing 3-1 Passing data to a destination view controller

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([[segue identifier] isEqualToString:@"ShowDetails"]) {
   MyDetailViewController *detailViewController = [segue destinationViewController];
    NSIndexPath *indexPath = [self.tableView indexPathForSelectedRow];
    detailViewController.data = [self.dataController
objectInListAtIndex:indexPath.row];
  }
}
```

A segue represents a one-way transition from a source scene to a destination scene. One of the consequences of this design is that you can use a segue to pass data to a destination, but you can't use a segue to send data from a destination to its source. To solve this problem, you create a delegate protocol that declares methods that the destination view controller calls when it needs to pass back some data.

Listing 3-2 shows one implementation of a protocol for passing data back to a source view controller.

Listing 3-2 Passing data to a source view controller

```
@protocol MyAddViewControllerDelegate <NSObject>
- (void)addViewControllerDidCancel:(MyAddViewController *)controller;
- (void)addViewControllerDidFinish:(MyAddViewController *)controller data:(NSString
 *)item;
@end


- (void)addViewControllerDidCancel:(MyAddViewController *)controller {
  [self dismissViewControllerAnimated:YES completion:NULL];
}
```

```
- (void)addViewControllerDidFinish:(MyAddViewController *)controller data:(NSString
 *)item {
  if ([item length]) {
    [self.dataController addData:item];
    [[self tableView] reloadData];
  }
  [self dismissViewControllerAnimated:YES completion:NULL];
}
```

Note: The full details of creating storyboards are described in *Xcode Overview*. To learn more about using view controllers in storyboards, see *View Controller Programming Guide for iOS*.

Design Pattern for Navigation-Based Apps

A navigation-based app with table views should follow these design best practices:

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

  When the table view displays a list of items, the object is typically an array. 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 a property 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 app 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 that this view controller, acting as data source, needs to populate its table view.

Creating and Configuring a Table View

Your app 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 sample projects *TableView Fundamentals for iOS* and *TheElements*.

Basics of Table View Creation

To create a table view, several entities in an app must interact: the view controller, the table view itself, and the table view's data source and delegate. The view controller, data source, and delegate are usually the same object. The view controller starts the calling sequence, diagrammed in Figure 4-1 (page 33).

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

   To learn how to create table views in a storyboard and programmatically, see "Creating a Table View Using a Storyboard" (page 34) and "Creating a Table View Programmatically" (page 39).

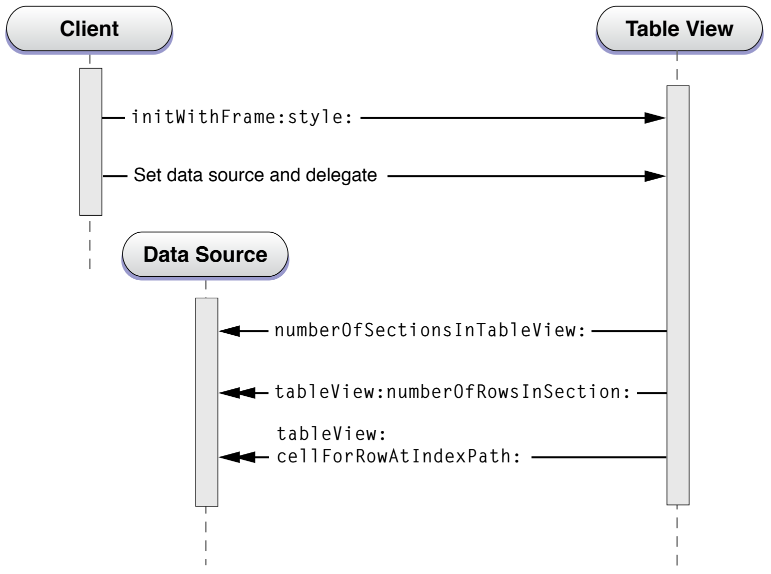
2. The view controller sets the data source and delegate of the table view and sends a `reloadData` message to it. The data source must adopt the `UITableViewDataSource` protocol, and the delegate must adopt the `UITableViewDelegate` protocol.

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


Client
Table View
initWithFrame:style:
Set data source and delegate
Data Source
numberOfSectionsInTableView:
tableView:numberOfRowsInSection:
tableView:
cellForRowAtIndexPath:

The diagram in Figure 4-1 shows the required protocol methods as well as the `numberOfSectionsInTableView:` method. Populating the table view with data occurs in steps 3 through 5. To learn how to implement the methods in these steps, see "Populating a Dynamic Table View with Data" (page 40).

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 the `tableView:titleForHeaderInSection:` method. For more on some of these optional table view customizations, see "Optional Table View Configurations" (page 48).

You create a table view in either the plain style (`UITableViewStylePlain`) or the grouped style (`UITableViewStyleGrouped`). (You specify the style in a storyboard.) 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). For an example, see "A Closer Look at Table View Cells" (page 51).

Recommendations for Creating and Configuring Table Views

There are many ways to put together a table view app. For example, you can 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 framework 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 apps use a custom `UITableViewController` object to manage a table view. As described in "Navigating a Data Hierarchy with Table Views" (page 21), `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 27).

- If your app is largely based on table views, select the Master-Detail Application template provided by Xcode when you create your project.

  As described in "Creating a Table View Using a Storyboard" (page 34), the template includes stub code and a storyboard defining an app delegate, the navigation controller, and the master 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 load them from a storyboard or create the associated table views programmatically.

  Although either option is possible, the storyboard route is generally easier.

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 `UITableViewController`, because this controller class sizes the table view to fill the screen between the navigation bar and the tab bar (if either is present).

Creating a Table View Using a Storyboard

Create an app with a table view using Xcode. When you create your project, select a template that contains stub code and a storyboard that, by default, supply the structure for setting up and managing table views.

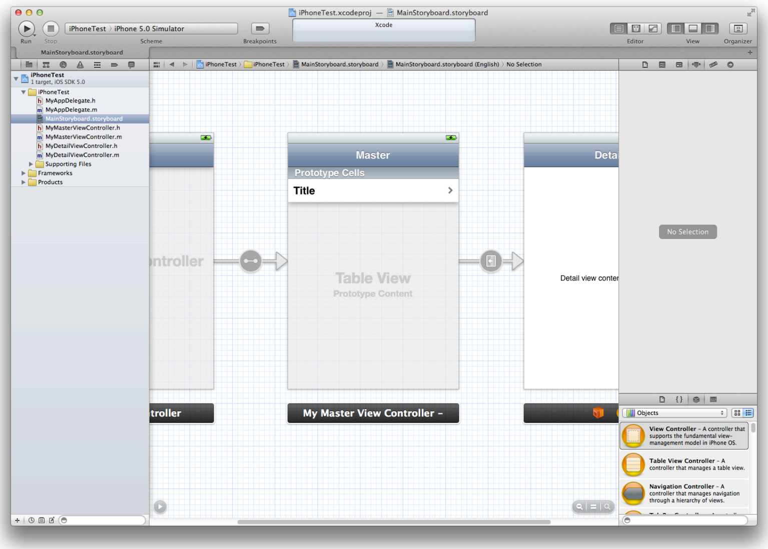
To create an app structured around table views

1. In Xcode, choose File > New > Project.

2. In the iOS section at the left side of the dialog, select Application.
3. In the main area of the dialog, select Master-Detail Application and then click Next.
4. Choose your project options (make sure Use Storyboard is selected), and then click Next.
5. Choose a save location for your project and then click Create.

Depending on which device family you chose in step 4, the project has one or two storyboards. To display the storyboard canvas, double-click a storyboard file in the project navigator. If the device family is iPhone, for example, your storyboard should contain a table view controller that looks similar to the one in Figure 4-2.

Figure 4-2 The master view controller in the Master-Detail Application storyboard



To make sure that the scene on the canvas represents the master view controller class in your code

1. On the canvas, click the scene's title bar to select the table view controller.
2. Click the Identity button at the top of the utility area to open the Identity inspector.
3. Verify that the Class field contains the project's custom subclass of `UITableViewController`.

Choose the Table View's Display Style

As described in "Table View Styles" (page 8), every table view has a display style: plain or grouped.

To choose the display style of a table view in a storyboard

1. Click the center of the scene to select the table view.

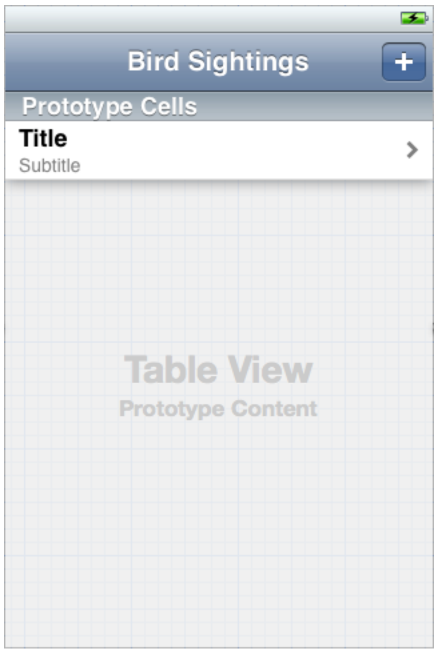
2. In the utility area, display the Attributes inspector.
3. In the Table View section of the Attributes inspector, use the Style pop-up menu to choose Plain or Grouped.

Choose the Table View's Content Type

Storyboards introduce two convenient ways to design a table view's content:

- **Dynamic prototypes**. Design a prototype cell and then use it as the template for other cells in the table. Use a dynamic prototype when multiple cells in a table should use the same layout to display information. Dynamic content is managed by the table view data source (the table view controller) at runtime, with an arbitrary number of cells. Figure 4-3 shows a plain table view with a one prototype cell.

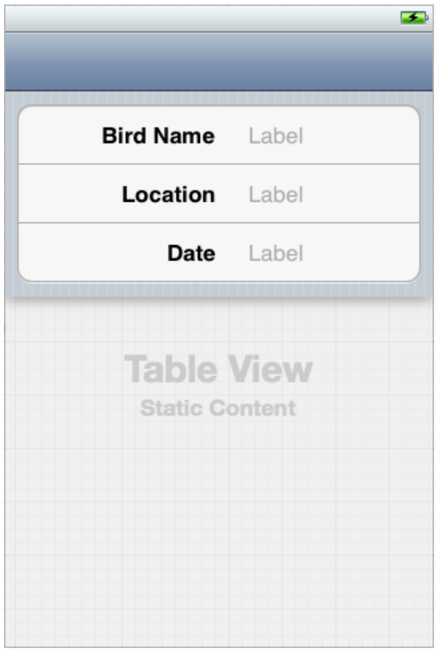
Figure 4-3 A dynamic table view



Note: If a table view in a storyboard is dynamic, the custom subclass of `UITableViewController` that contains the table view needs to implement the data source protocol. For more information, see "Populating a Dynamic Table View with Data" (page 40).

- **Static cells**. Use static content to design the overall layout of the table, including the total number of cells. A table view with static content has a fixed set of cells that you can configure at design time. You can also configure other static data elements such as section headers. Use static cells when a table does not change its layout, regardless of the specific information it displays. Figure 4-4 shows a grouped table view with three static cells.

Figure 4-4 A static table view



Note: If a table view in a storyboard is static, the custom subclass of `UITableViewController` that contains the table view should *not* implement the data source protocol. Instead, the table view controller should use its `viewDidLoad` method to populate the table view's data. For more information, see "Populating a Static Table View With Data" (page 42).

By default, when you add a table view controller to a storyboard, the controller contains a table view that uses prototype-based cells. If you want to use static cells:

1. Select the table view.
2. Display the Attributes inspector.
3. In the the Content pop-up menu, choose Static Cells.

If you're designing a prototype cell, the table view needs a way to identify the prototype when the data source dequeues reusable cells for the table at runtime. You do this by assigning a reuse identifier to the cell. In the Table View Cell section of the Attributes inspector, enter a string in the Identifier text field, which you will also use when asking for a new cell of that type. To make understanding the code easier, a cell's reuse identifier should describe what the cell contains. For example, a cell for displaying bird sightings might have an identifier of `@"BirdSightingCell"`.

Design the Table View's Rows

As described in "Standard Styles for Table View Cells" (page 13), UIKit defines four styles for the cells that a table view uses to draw its rows. You can use one of the four standard styles, design a custom style, or subclass `UITableViewCell` to define additional behavior or properties for the cell. This topic is covered in detail in "A Closer Look at Table View Cells" (page 51).

A table view cell can also have an accessory, as described in "Accessory Views" (page 17). An accessory is a standard user interface element that UIKit draws at the right end of a table cell. For example, the disclosure indicator, which looks similar to a right angle bracket (>), tells users that tapping an item reveals related information in a new screen. In the Attributes inspector, use the Accessory pop-up menu to select a cell's accessory.

Create Additional Table Views

If your app displays and manages more than one table view, add those table views to your storyboard. You add a table view by adding a custom `UITableViewController` object, which contains the table view it manages.

To add custom class files to your project

1. In Xcode, choose File > New > File.
2. In the iOS section at the left side of the dialog, select Cocoa Touch.
3. In the main area of the dialog, select Objective-C class, and then click Next.
4. Enter a name for your new class, choose subclass of `UITableViewController`, and then click Next.
5. Choose a save location for your class files, and then click Create.

To add a table view controller to a storyboard

1. Display the storyboard to which you want to add the table view controller.
2. Drag a table view controller out of the object library and drop it on the storyboard.

3. With the new scene still selected on the canvas, click the Identity button in the utility area to open the Identity inspector.
4. In the Custom Class section, choose the new custom class in the Class pop-up menu.
5. Set the new table view's style and cell content (dynamic or static).
6. Create a segue to the new scene.

The details of step 7 vary depending on the project. To learn more about adding segues, see *Xcode Overview*.

Note: Populating a table view with data and configuring a table view are discussed in "Populating a Dynamic Table View with Data" (page 40) and "Optional Table View Configurations" (page 48).

Learn More by Creating a Sample App

The tutorial *Your Second iOS App: Storyboards* shows how to create a sample app that is structured around table views. After you complete the steps in this tutorial, you'll have a working knowledge of how to create both dynamic and static table views using a storyboard. The tutorial creates a basic navigation-based app called BirdWatching that uses table view controllers connected by both push and modal segues.

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."

Adopt the Data Source and Delegate Protocols

The class creating the table view 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 Listing 4-1.

Listing 4-1 Adopting the data source and delegate protocols

```
@interface RootViewController : UIViewController <UITableViewDelegate,
UITableViewDataSource>


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

Create and Configure a Table View

The next step is for the client to allocate and initialize an instance of the `UITableView` class. Listing 4-2 gives an example of a client that creates a `UITableView` object in the plain style, specifies its autoresizing 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 4-2 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;
}
```

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 a Dynamic Table View with Data

Just after a table view object is created, it 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:` method 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.)

As noted in "Choose the Table View's Content Type" (page 36), if the table view is dynamic then you need to implement the required data source methods. Listing 4-3 shows an example of how the data source and the delegate could configure a dynamic table view.

Listing 4-3 Populating a dynamic 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 = @"MyReuseIdentifier";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:MyIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
  reuseIdentifier:MyIdentifier];
    }
    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 a `dequeueReusableCellWithIdentifier:` message. If no such object exists, the data source creates it, assigning it a reuse identifier. The data source sets the cell's content (in this example, its text) and returns it. "A Closer Look at Table View Cells" (page 51) discusses this data source method and `UITableViewCell` objects in more detail.

If the `dequeueReusableCellWithIdentifier:` method asks for a cell that's defined in a storyboard, the method always returns a valid cell. If there is not a recycled cell waiting to be reused, the method creates a new one using the information in the storyboard itself. This eliminates the need to check the return value for `nil` and create a cell manually.

The implementation of the `tableView:cellForRowAtIndexPath:` method in Listing 4-3 includes an `NSIndexPath` argument that identifies the table view section and row. 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 number. For more information on this category, see *NSIndexPath UIKit Additions* .

Populating a Static Table View With Data

As noted in "Choose the Table View's Content Type" (page 36), if a table view is static then you should not implement any data source methods. The configuration of the table view is known at compile time, so UIKit can get this information from the storyboard at runtime. However, you still need to populate a static table view with data from your data model. "Populating a Static Table View With Data" shows an example of how a table view controller could load user data in a static table view. This example is adapted from *Your Second iOS App: Storyboards* .

Listing 4-4 Populating a static table view with data

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

    BirdSighting *theSighting = self.sighting;
    static NSDateFormatter *formatter = nil;
    if (formatter == nil) {
        formatter = [[NSDateFormatter alloc] init];
        [formatter setDateStyle:NSDateFormatterMediumStyle];
    }
    if (theSighting) {
        self.birdNameLabel.text = theSighting.name;
        self.locationLabel.text = theSighting.location;
       self.dateLabel.text = [formatter stringFromDate:(NSDate*)theSighting.date];
    }
}
```

The table view is populated with data in the `UIViewController` method `viewDidLoad`, which is called after the view is loaded into memory. The data is passed to the table view controller in the `sighting` object, which is set in the previous view controller's `prepareForSegue:sender:` method. The properties `birdNameLabel`, `locationLabel`, and `dateLabel` are outlets connected to labels in the static table view (see Figure 4-4 (page 37)).

Populating an Indexed List

An indexed list (see Figure 1-2 (page 10)) 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:`

  Returns an array of the strings to use as the index entries (in order).
- `tableView:titleForHeaderInSection:`

  Maps these index strings to the titles of the table view's sections (they don't have to be the same).
- `tableView:sectionForSectionIndexTitle:atIndex:`

  Returns the section index related to the entry the user tapped in the index.

The data 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 can build and sort this array of arrays yourself, but fortunately the `UILocalizedIndexedCollation` class greatly simplifies the tasks of building and sorting these data structures and providing data to the table view. The class also collates items in the arrays according to the current localization.

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 `UILocalizedIndexedCollation` 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 also 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 and a `sectionNumber` property.

Listing 4-5 Defining the model-object interface

```
@interface State : NSObject


@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, you 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];
        }
    } else  {
        return;
    }
```

After the data source has this “raw” array of model objects, it can process it with the facilities of the `UILocalizedIndexedCollation` class. In Listing 4-7, the code is annotated 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
```

Here's what the code in Listing 4-7 does:

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];

}
```

Accessibility Note: To change what VoiceOver reads aloud when the indexed list is selected, assign a localized string to the `accessibilityLabel` property of each item in the array that `sectionIndexTitlesForTableView:` returns.

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;
    cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:CellIdentifier];
    }
    State *stateObj = [[self.states objectAtIndex:indexPath.section]
objectAtIndex:indexPath.row];
    cell.textLabel.text = stateObj.name;
    return cell;
}
```

For table views that are indexed lists, when the data source assigns cells for rows in `tableView:cellForRowAtIndexPath:`, it should ensure that the `accessoryType` property of the cell is set to `UITableViewCellAccessoryNone`.

After initially populating the table view following the procedure outlined above, you can reload the contents of the index by calling the `reloadSectionIndexTitles` method.

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.

Add a Custom Title

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
{
    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];
}
```

Provide a Section Title

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

Listing 4-11 Returning a title for a 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];
}
```

Indent a Row

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;
}
```

Vary a Row's Height

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;
}
```

Customize Cells

You can also affect the appearance of rows by returning custom `UITableViewCell` objects with specially formatted subviews for content in `tableView:cellForRowAtIndexPath:`. Cell customization is discussed in "A Closer Look at Table View Cells" (page 51).

A Closer Look at Table View Cells

A table view uses cell objects to draw its visible rows and then caches those objects as long as the rows are visible. Cells 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.

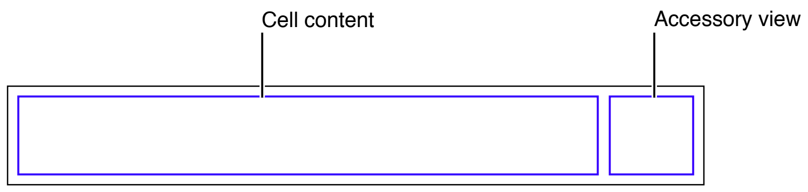
In this chapter, you'll learn about:

- The characteristics of cells
- How to use the default capabilities of `UITableViewCell` for setting cell content
- 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. Figure 5-1 shows the major parts of a cell.

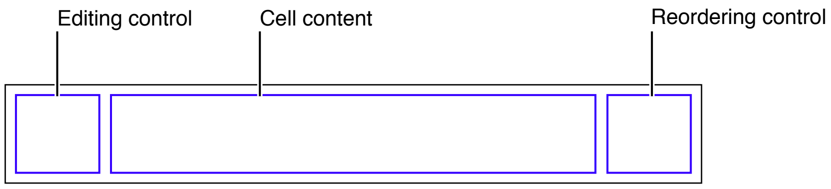
Figure 5-1 Parts of a table view cell



The 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.

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.

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



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.

If a cell object is reusable—the typical case—you assign it a reuse identifier (an arbitrary string) in the storyboard. At runtime, the table view stores cell objects in an internal queue. When the table view asks the data source to configure a cell object for display, 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 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.

With multiple cell objects in a queue, each with its own identifier, you can have table views constructed from cell objects of different types. For example, some rows of a table view can have content based on the image and text properties of a `UITableViewCell` in a predefined style, while other rows can 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 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 Interface Builder), or you can use cell objects created from a custom subclass of `UITableViewCell`. Note that the content view is a container of other views and so displays no content itself.

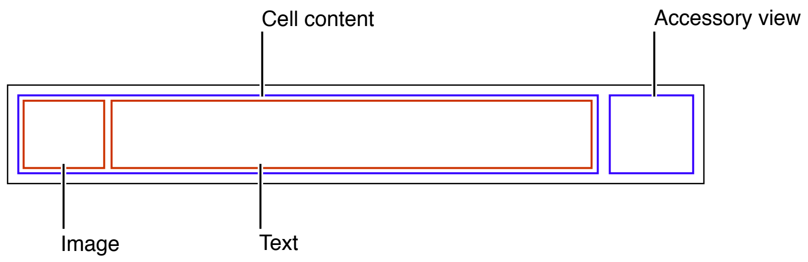
Using Cell Objects in Predefined Styles

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 13) 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;
```

These cell objects have two kinds of content: one or more 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



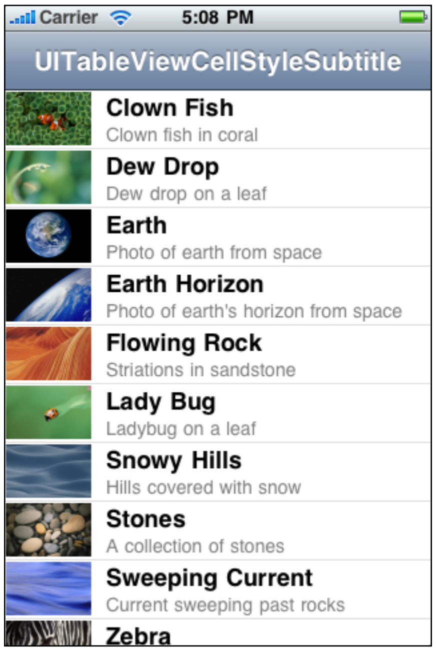
The UITableViewCell class defines three properties for this cell content:

- textLabel—A label for the title (a UILabel object)
- detailTextLabel—A label for the subtitle if there is additional detail (a UILabel object)
- imageView—An image view for 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 alternative 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



Clown Fish
Clown fish in coral
Dew Drop
Dew drop on a leaf
Earth
Photo of earth from space
Earth Horizon
Photo of earth's horizon from space
Flowing Rock
Striations in sandstone
Lady Bug
Ladybug on a leaf
Snowy Hills
Hills covered with snow
Stones
A collection of stones
Sweeping Current
Current sweeping past rocks
Zebra

Listing 5-1 shows the implementation of `tableView:cellForRowAtIndexPath:` that creates the table view rows in Figure 5-4 (page 54). The first thing the data source should do is send `dequeueReusableCellWithIdentifier:` to the table view, passing in a reuse identifier. If a prototype for the cell exists in a storyboard, the table view returns a reusable cell object. 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"];
         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 can also set various other properties, including (but not limited to) the following:

- `selectionStyle`—Controls the appearance of the cell when selected.
- `accessoryType` and `accessoryView`—Allow 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 (nonediting) mode. For a custom view, you may provide any `UIView` object, such as a slider, a switch, or a custom view.
- `editingAccessoryType` and `editingAccessoryView`—Allow 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.
- `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`—Provide a background view (when a cell is unselected and selected) to display behind all other views of the cell.
- `indentationLevel` and `indentationWidth`—Specify 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 a cell's background color, you could set its `backgroundColor` property. Listing 5-2 shows how you might use the delegate method `tableView:willDisplayCell:forRowAtIndexPath:` to alternate the background color of rows (via their backing cells) in a table view.

Listing 5-2 Alternating the background color of cells

```
- (void)tableView:(UITableView *)tableView willDisplayCell:(UITableViewCell *)cell
 forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (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. With this method, the delegate should change only state-based properties that were set earlier by the table view, such as selection and background color, and not content.

Customizing Cells

The four predefined styles of `UITableViewCell` objects 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.

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:

- Add subviews to a cell's content view.
- Create a custom subclass of `UITableViewCell`.

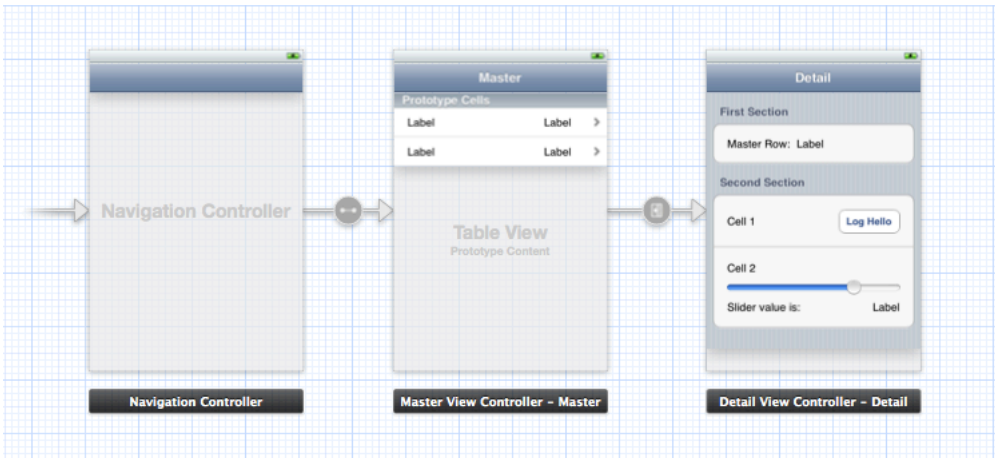
The following sections discuss both approaches.

Loading Table View Cells from a Storyboard

In a storyboard, the cells in a table view are dynamic or static. 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 quantity that's known at compile time. A table view that presents a detail view of an item is a good candidate for static content.

You can design dynamic or static cell content directly inside a table view object. Figure 5-5 shows the master and detail table views in a simple storyboard. In this example, the master table view contains dynamic prototype cells, and the detail table view contains static cells.

Figure 5-5 Table view cells in a storyboard

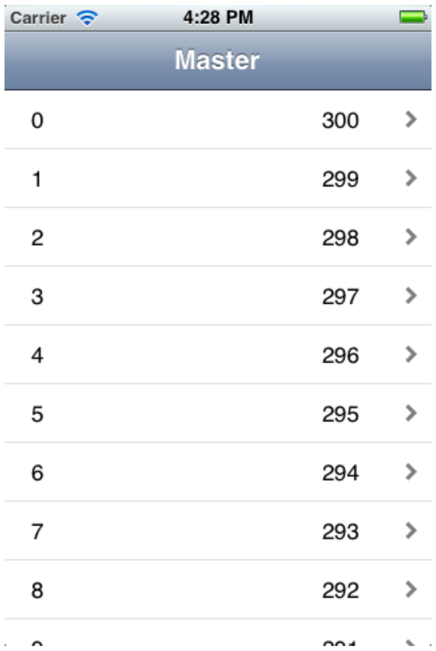


The following sections demonstrate how to load data into table views that contain custom-configured cells.

The Technique for Dynamic Row Content

In this section, you compose a custom prototype cell in a storyboard. At runtime, the data source dequeues cells, prepares them, and gives them to its table view for drawing the rows depicted in Figure 5-6.

Figure 5-6 Table view rows drawn with a custom prototype cell



The data source can use two different ways to access the subviews of the cells. One approach uses the `tag` property defined by `UIView` and the other approach uses outlets. Using tags is convenient, although it makes the code more fragile because it introduces a coupling between the tag numbers in the storyboard and the code. Using outlets requires a little more work because you need to define a custom subclass of `UITableViewCell`. Both approaches are described here.

To create a project that uses a storyboard to load custom table view cells

1. Create a project using the Master-Detail Application template and select the Use Storyboards option.
2. On the storyboard canvas, select the master view controller.
3. In the Identity inspector, verify that Class is set to the custom MasterViewController class.
4. Select the table view inside the master view controller.
5. In the Attributes inspector, verify that the Content pop-up menu is set to Dynamic Prototypes.
6. Select the prototype cell.
7. In the Attributes inspector, choose Custom in the Style pop-up menu.

8. Enter a reuse identifier in the Identifier text field.

   This is the same reuse identifier you send to the table view in the `dequeueReusableCellWithIdentifier:` message. For an example, see Listing 5-3.

9. Choose Disclosure Indicator in the Accessory pop-up menu.
10. Drag objects from the Library onto the cell.

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

11. Select the objects and set their attributes, sizes, and autoresizing 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 inspector and assign each object a unique integer.

Now write the code you would normally write to obtain the table view's data. (For this example, the only data you need is the row number of each cell.) Implement the data source method `tableView:cellForRowAtIndexPath:` to create a new cell from the prototype and populate it with data, in a manner similar to Listing 5-3.

Listing 5-3 Adding data to a cell using tags

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


    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 aspects of this code to note:

- The string identifier you assign to the prototype cell is the same string you pass to the table view in `dequeueReusableCellWithIdentifier:`.
- Because the prototype cell is defined in a storyboard, the `dequeueReusableCellWithIdentifier:` method always returns a valid cell. You don't need to check the return value against nil and create a cell manually.
- 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.

If you prefer not to use tags, you can use an alternative method for setting the content in the cell. Define a custom `UITableViewCell` subclass with outlet properties for the objects you want to set. In the storyboard, associate the new class with the prototype cell and connect the outlets to the corresponding objects in the cell.

To use outlets for the custom cell content

1. Add an Objective-C class named `MyTableViewCell` to your project.
2. Add the following code to the interface in `MyTableViewCell.h`:

   ```
   @interface MyTableViewCell : UITableViewCell


   @property (nonatomic, weak) IBOutlet UILabel *firstLabel;
   @property (nonatomic, weak) IBOutlet UILabel *secondLabel;
   @end
   ```

3. Add the following code to the implementation in `MyTableViewCell.m`:

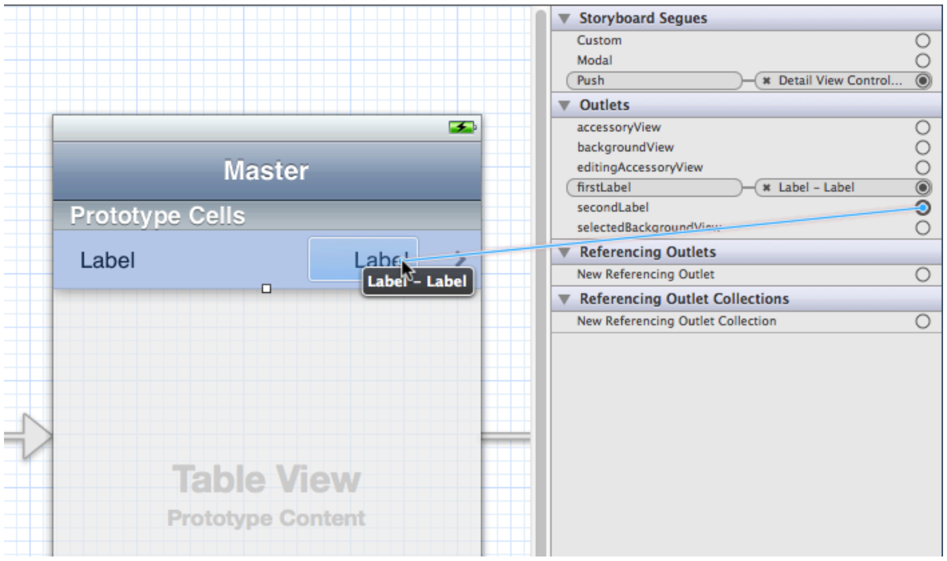
   ```
   @synthesize firstLabel, secondLabel;
   ```

4. Add the following line of code to the source file that implements the data source:

   ```
   #import "MyTableViewCell.h"
   ```

5. Use the Identity inspector to set the Class of the prototype cell to `MyTableViewCell`.

6. Use the Connections inspector to connect the two outlets in the prototype cell to their corresponding labels.



7. Implement the data source method `tableView:cellForRowAtIndexPath:` in a manner similar to Listing 5-4.

Listing 5-4 Adding data to a cell using outlets

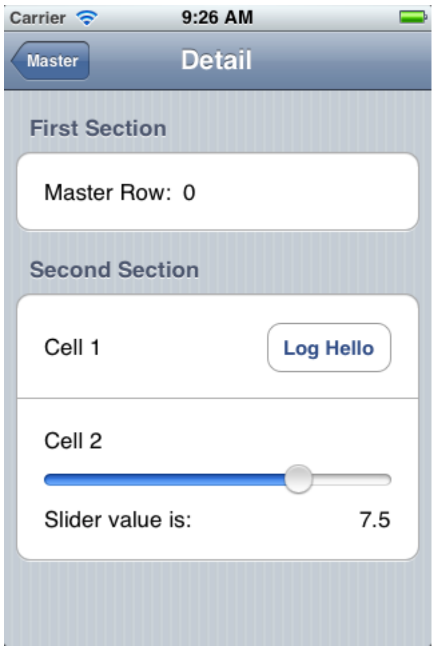
```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    MyTableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"MyIdentifier"];


    cell.firstLabel.text = [NSString stringWithFormat:@"%d", indexPath.row];
    cell.secondLabel.text = [NSString stringWithFormat:@"%d", NUMBER_OF_ROWS -
indexPath.row];


    return cell;
}
```

The code gains access to the labels in the cell using accessor methods (dot notation is used here). The code can then set the textual content of the labels.

The Technique for Static Row Content

In this section, you compose several cells in a table view with static content. At runtime, when the table view is loaded from the storyboard, the table view controller 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



As with the procedure for dynamic content, start by adding a subclass of `UITableViewController` to your project. Define outlet properties for the master row label in the first cell and the slider value label in the last cell, as shown in Listing 5-5.

Listing 5-5 Defining outlet properties for static cell objects

```
@interface DetailViewController : UITableViewController

@property (strong, nonatomic) id detailItem;
@property (weak, nonatomic) IBOutlet UILabel *masterRowLabel;
@property (weak, nonatomic) IBOutlet UILabel *sliderValueLabel;
@property (weak, nonatomic) IBOutlet UISlider *slider;

- (IBAction)logHello;
- (IBAction)sliderValueChanged:(UISlider *)slider;
```

```

@end
```

In the storyboard, drag a Table View Controller object from the Library onto the canvas. Select the table view and set the following attributes in the Attributes inspector:

1. Set the Content pop-up menu to Static Cells.
2. Set the number of sections to 2.
3. Set the Style pop-up menu to Grouped.

For each section in the table view, use the Attributes inspector to enter a string in the Header field. Then for the cells, complete the following steps:

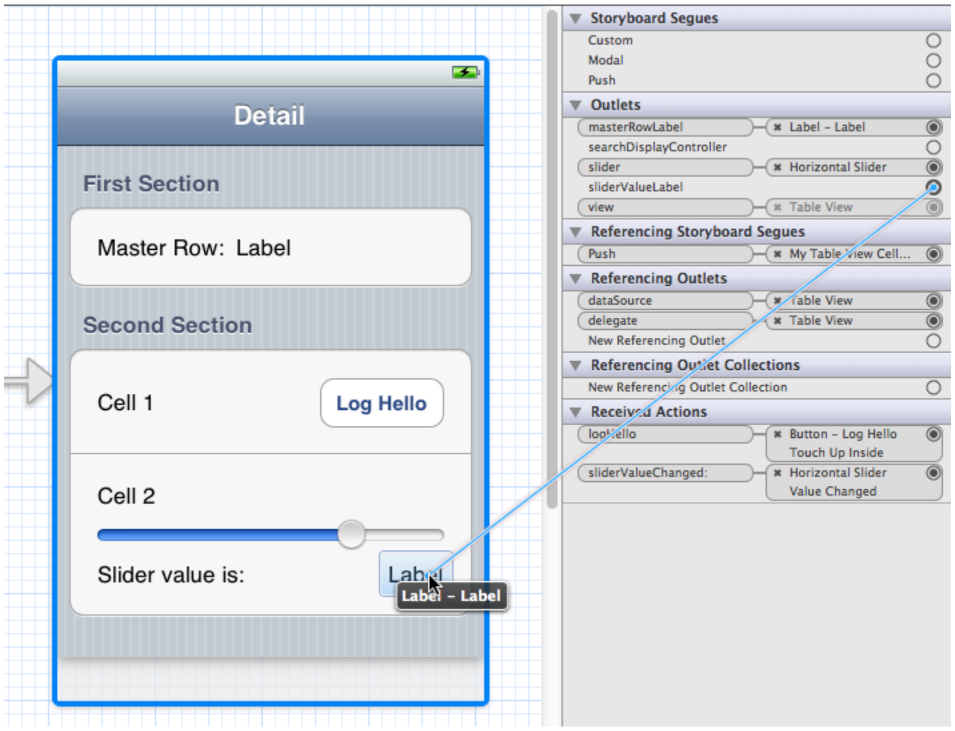
1. Delete two of the three cells in the first table-view section and one cell in the second section.
2. Increase the height of each remaining cell as needed.

   It isn't necessary to assign reuse identifiers of these cells, because you're not going to implement the data source method `tableView:cellForRowAtIndexPath:`.
3. Drag objects from the Library to compose the subviews of each cell as depicted in Figure 5-7 (page 62).
4. Set any desired attributes of these objects.

   The slider in this example has a range of values from 0 to 10 with an initial value of 7.5.

Select the table view controller and display the Connections inspector. Make connections between the three outlets in your table view controller and the corresponding objects, as shown in Figure 5-8. While you're at it, implement the two action methods declared in Listing 5-5 (page 62) and make target-action connections to the button and the slider.

Figure 5-8 Making connections to your static cell content



To populate the data in the static cells, implement a method called `configureView` in the detail view controller. In this example, `detailItem` is an `NSString` object passed in by the master view controller in its `prepareForSegue:sender:` method. The string contains the master row number.

Listing 5-6 Setting the data in the user interface

```
- (void)configureView
{
    if (self.detailItem) {
        self.masterRowLabel.text = [self.detailItem description];
    }
    self.sliderValueLabel.text = [NSString stringWithFormat:@"%1.1f",
self.slider.value];
}
```

The detail view controller calls the `configureView` method in `viewDidLoad` and `setDetailItem:`, as illustrated in the Xcode template Master-Detail Application.

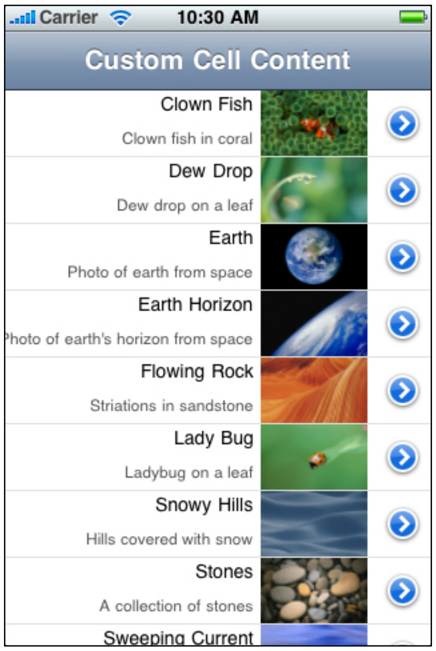
Programmatically Adding Subviews to a Cell's Content View

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, 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 can configure and lay them out programmatically or in Interface Builder. (The approach using Interface Builder is discussed in "Loading Table View Cells from a Storyboard" (page 57).)

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. The content is selected automatically if the subview implements (if appropriate) the accessor methods for the `highlighted` property.

Suppose you want a cell with text and image content in custom locations. 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 5-9 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 5-9 Cells with custom content as subviews



The code example in Listing 5-7 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 5-7 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];
        cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;


        mainLabel = [[UILabel alloc] initWithFrame:CGRectMake(0.0, 0.0, 220.0,
15.0)];
        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)];
        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)];
        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;
}
```

When the data source creates the cells, it assigns each subview an identifier called a tag. With tags, you can locate a view in its view hierarchy by calling the `viewWithTag:` method. If the delegate later 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.

This code creates a `UITableViewCell` object in the predefined default style (`UITableViewCellStyleDefault`). 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: Another approach 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`).

One way to achieve "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. If you want that kind of variation within a label object, create multiple labels and lay them out relative to each other.

Enhancing the Accessibility of Table View Cells

If your app displays a table view in which each cell contains items other than (or in addition to) text, there are a few things you can do to make it more accessible. Similarly, if your table view displays more than one piece of information per row, you can enhance a VoiceOver user's experience by aggregating the information in a single, easy-to-understand label.

Note: If your table cells contain any of the standard table-view elements, such as the disclosure indicator, detail disclosure button, or delete control, you do not have to do anything to make these elements accessible. If, however, your table cells include other types of controls, such as a switch or a slider, you need to make sure that these elements are appropriately accessible.

If the table cells in your app contain a mix of different elements, determine whether users interact with each cell as a unit, or with individual elements inside the cell. If users need to access individual elements inside the cell, you should:

- Make each individual element accessible separately.
- Make sure the table cell itself is *not* accessible.
- Succinctly describe the overall contents of the cell and use this description for the label attribute of the cell. Note that, in this case, the label is considered to be one of the accessible elements within the cell.

You've probably recognized that a table cell that contains multiple items, such as text and controls, fits the criteria of a container view, as defined by the UI Accessibility programming interface. However, you do not have to identify the cell as a container view or implement any of the methods of the `UIAccessibilityContainer` protocol, because the table cell is automatically designated as a container.

If your table contains cells that provide information in discrete chunks, you should consider combining the information from these chunks in the label attribute. When you do this, VoiceOver users can get the meaning of the cell's contents with one gesture, instead of having to access each piece of information separately.

A good example of how this can work is in the built-in Stocks app. Instead of providing the company name, current stock price, and change in price as separate strings, Stocks combines this information in the label, which might sound like this: "Apple Inc., $648.11, up 1.85%." Notice the commas in this label. When you combine discrete pieces of information in this way, you can use commas to tell VoiceOver to pause briefly between phrases, making it easier for users to understand the information.

Listing 5-8 (page 70) shows how to combine the information in the labels of two separate elements into a single label that describes both:

Listing 5-8 Concatenating labels of a table cell

```
@implementation WeatherTableViewController
// This is a view that provides weather information. It contains a city subview
and a temperature subview, each of which provides a separate label.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"Cell"
forIndexPath:indexPath];


    // set up the cell here...


    NSString *cityLabel = [self.weatherCity accessibilityLabel];
    NSString *temperatureLabel = [self.weatherTemp accessibilityLabel];


    // Combine the city and temperature information so that VoiceOver users can
get the weather information with one gesture.
    [cell setAccessibilityLabel:[NSString stringWithFormat:@"%@, %@", cityLabel,
temperatureLabel]];
    return cell;
}
@end
```

Joining the table cell's accessibility labels isn't the only thing you can do to enhance the overall accessibility of your table view. You can also change the way VoiceOver reads the table view's indexed list. To learn more, read "Populating an Indexed List" (page 43).

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. Ensure that 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];
}
```

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"];
```

```
            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:(UIControlEventValueChanged | UIControlEventTouchDragInside)];
        cell.accessoryView = switchObj;

        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 Table View Cells from a Storyboard" (page 57) 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 indexOfObject: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 *)theTableView
         didSelectRowAtIndexPath:(NSIndexPath *)newIndexPath {

    [theTableView deselectRowAtIndexPath:[theTableView indexPathForSelectedRow]
animated:NO];
    UITableViewCell *cell = [theTableView 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
*)newIndexPath {
    NSIndexPath *scrollIndexPath;
    if (newIndexPath.row + 20 < [timeZoneNames count]) {
        scrollIndexPath = [NSIndexPath indexPathForRow:newIndexPath.row+20
inSection:newIndexPath.section];
    } else {
        scrollIndexPath = [NSIndexPath indexPathForRow:newIndexPath.row-20
inSection:newIndexPath.section];
    }
    [theTableView 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 84), 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 88).

Inserting and Deleting Rows in Editing Mode

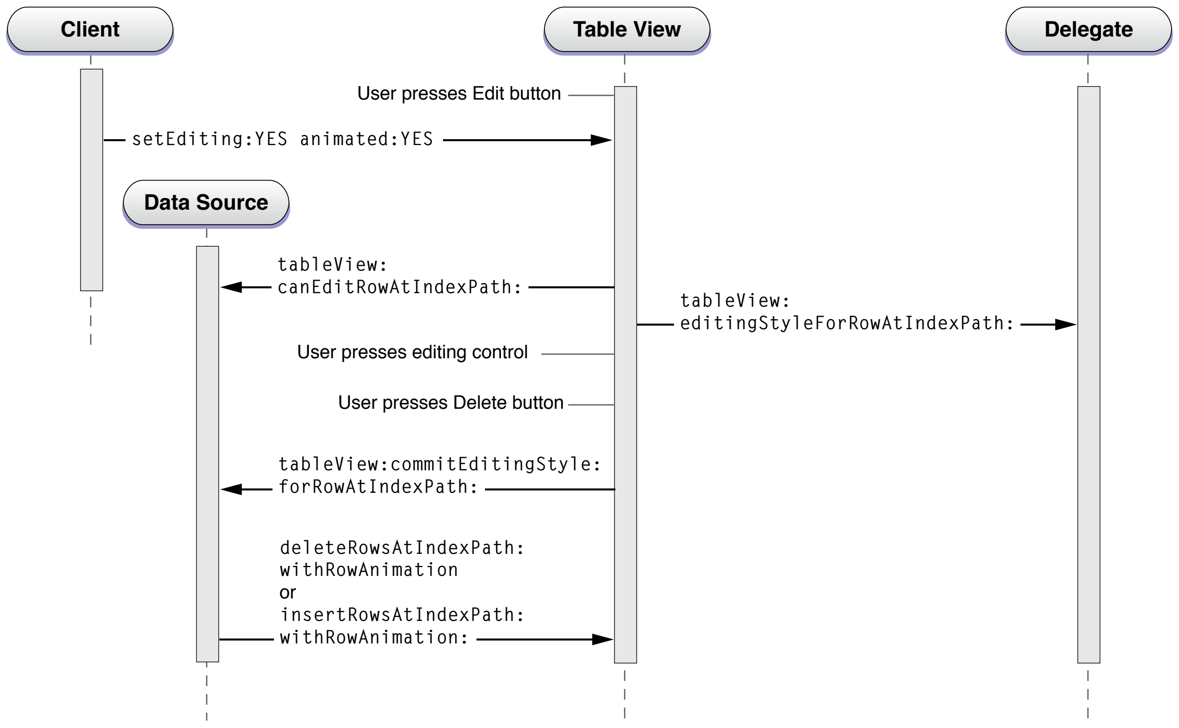
When a Table View is Edited

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 79). 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 82) 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 *)tableView
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];
}
```

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 78)).

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.

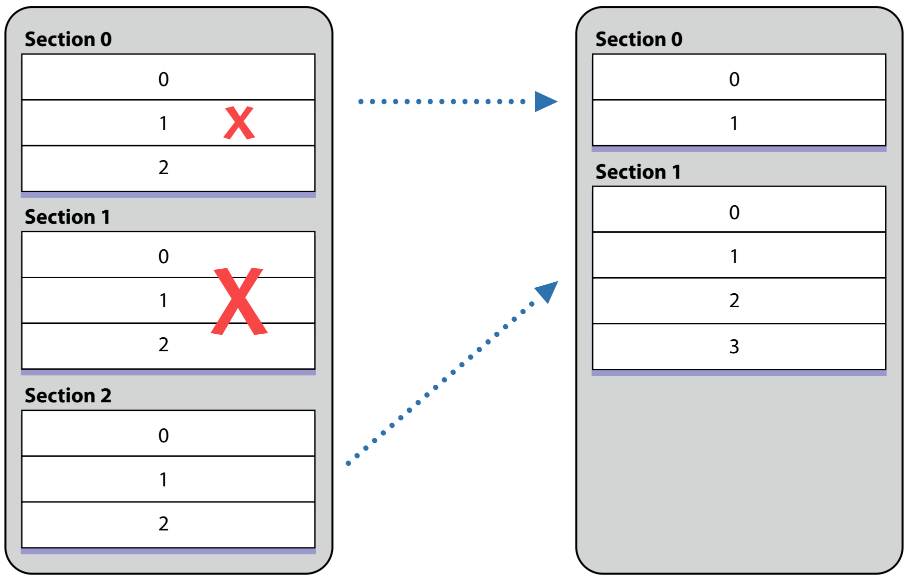
Deletion and reloading operations within an animation block specify which rows and sections in the original table should be removed or reloaded; insertions specify which rows and sections should be added to the resulting table. The index paths used to identify sections and rows follow this model. Inserting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

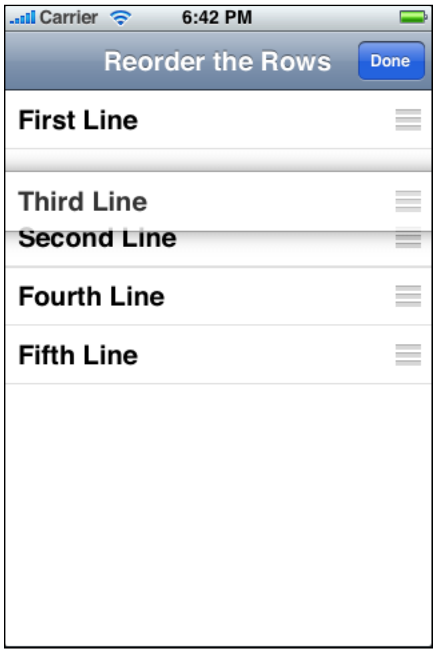
Figure 7-2 Deletion of section and row and insertion of row



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.

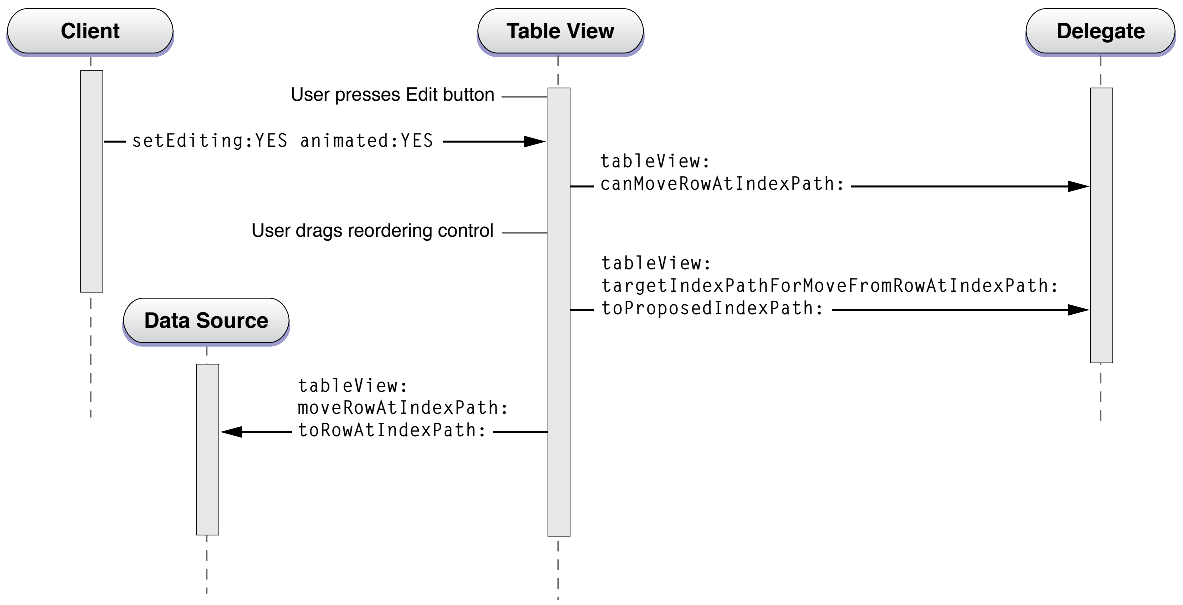
What Happens When a Row is Relocated

A table view goes into editing mode when it receives a `setEditing:animated:` message. This normally happens when the user taps an Edit button in the navigation bar, but you can implement your own controls if you wish. 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`. In order for reorder controls to appear, the data source must support reordering by implementing the `tableView:moveRowAtIndexPath:toIndexPath:` method.

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 88). 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.

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];
    [self.reorderingRows removeObjectAtIndex:sourceIndexPath.row];
    [self.reorderingRows insertObject:stringToMove atIndex:destinationIndexPath.row];
}
```

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;
}
```

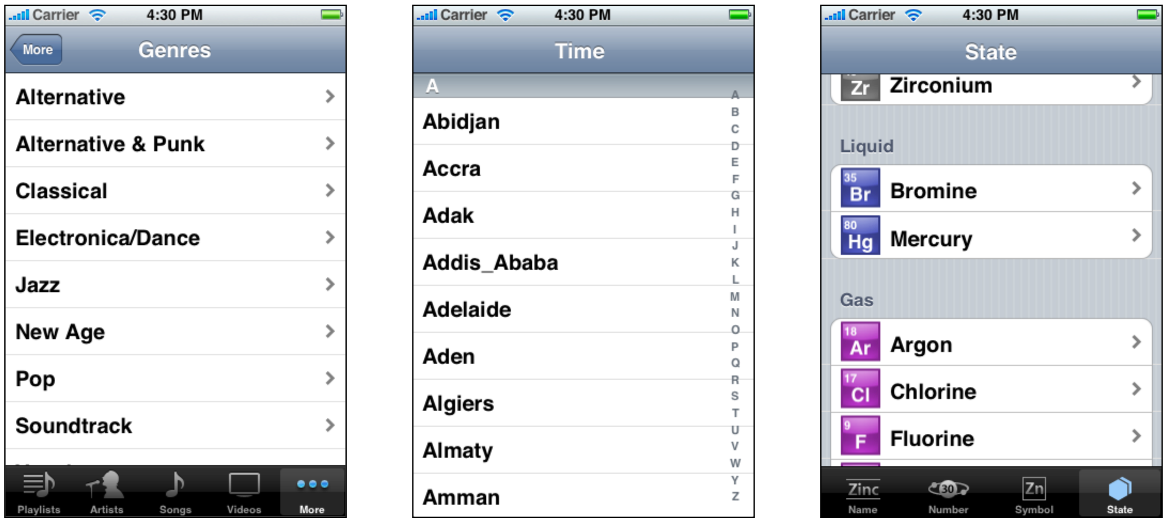
About Table Views in iOS Apps

Table views are versatile user interface objects frequently found in iOS apps. A table view presents data in a scrollable list of multiple rows that may be divided into sections.

Table views 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, the UIKit framework 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.

At a Glance

A table view is an instance of the `UITableView` class in one of two basic styles, plain or grouped. A plain 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. The data source object provides the data for populating the sections and rows of the table view. The delegate object customizes its appearance and behavior.

Related chapters: "Table View Styles and Accessory Views" (page 8)

Table Views Draw Their Rows Using Cells

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 selecting 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 app's table views.

When you configure the attributes of a table view in the storyboard editor, you choose between two types of cell content: static cells or dynamic prototypes.

- **Static cells**. Use static cells to design a table with a fixed number of rows, each with its own layout. Use static cells when you know what the table looks like at design time, regardless of the specific information it displays.
- **Dynamic prototypes**. Use dynamic prototypes to design one cell and then use it as the template for other cells in the table. Use a dynamic prototype when multiple cells in a table should use the same layout to display information. Dynamic prototype content is managed by the data source at runtime, with an arbitrary number of cells.

Related Chapters: "Table View Styles and Accessory Views" (page 8), "A Closer Look at Table View Cells" (page 51)

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 that the row is in. It uses this information to locate the corresponding item in the app's data model. This item might be at an intermediate level in the hierarchy of data or it might be a "leaf node" in the hierarchy. If the item is at an intermediate level, the app displays a new table view. If the item is a leaf node, the app displays details 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 21), "Managing Selections" (page 72)

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

Table views can enter an editing mode in which users 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 users touch a deletion control or, in some table views, swipe across a row, a red Delete button appears, prompting users to delete 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 so that they can manage these operations.

Related Chapters: "Inserting and Deleting Rows and Sections" (page 77), "Managing the Reordering of Rows" (page 88)

To Create a Table View, Use a Storyboard

The easiest and recommended way to create and manage a table view is to use a custom `UITableViewController` object in a storyboard. If your app is based largely on table views, create your Xcode project using the Master-Detail Application template. This template includes an initial custom `UITableViewController` class and a storyboard for the scenes in the user interface, including the custom view controller and its table view. In the storyboard editor, choose one of the two styles for this table view and design its content.

At runtime, `UITableViewController` creates the table view and assigns itself as delegate and 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.

Related Chapters: "Navigating a Data Hierarchy with Table Views" (page 21), "Creating and Configuring a Table View" (page 32)

Prerequisites

Before reading this document, you should read *Start Developing iOS Apps Today* to understand the basic process for developing iOS apps. Then read *View Controller Programming Guide for iOS* for a comprehensive look at view controllers and storyboards. Finally, to gain valuable hands-on experience using table views in a storyboard, read the tutorial *Your Second iOS App: Storyboards*.

The information presented in this introduction and in "Table View Styles and Accessory Views" (page 8) summarizes prescriptive information on table views presented in *iOS Human Interface Guidelines*. You can find a complete description of the styles and characteristics of table views, as well as their recommended uses, in the chapter "Content Views".

See Also

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

- *SimpleDrillDown* project
- *Table View Animations and Gestures* project

For guidance on how to use the standard container view controllers provided by UIKit, see *View Controller Catalog for iOS*. This document describes split view controllers and navigation controllers, which can both contain table view controllers as children.

Document Revision History

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

Date	Notes
2013-09-18	Clarified the requirements for reordering controls to appear. Clarified the requirements for reordering controls to appear in "What Happens When a Row is Relocated" (page 88).
2012-12-13	Added a section about enhancing the accessibility of table view cells.
2012-09-19	Added new information for iOS 5.
2011-01-05	Made some minor corrections.
2010-09-14	Made several minor corrections.
2010-08-03	States now that beginUpdates...endUpdates 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 udpates.
2010-03-24	Made the introduction more informative.
2009-08-19	Explained how to set the background color of cells, emphasized purpose of tableView:willDisplayCell:forRowAtIndexPath:, 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.

Date	Notes
2008-10-15	Warned against calling setEditing:animated: when in tableView:commitEditingStyle:forRowAtIndexPath:. 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.