CHAPTER 8
Table Views and Collection Views

I'm gonna ask you the three big questions. — Go ahead. — Who made you? — You did. — Who owns the biggest piece of you? — You do. — What would happen if I dropped you? — I'd go right down the drain.

—Dialogue by Garson Kanin and Ruth Gordon, *Pat and Mike*

A table view (UITableView) is a vertically scrolling UIScrollView (Chapter 7) containing a single column of rectangular cells (UITableViewCell, a UIView subclass). It is a keystone of Apple's strategy for making the small iPhone screen useful and powerful, and has three main purposes:

Presentation of information
: The cells typically contain text, which the user can read. The cells are usually quite small, in order to maximize the quantity appearing on the screen at once, so this text is often condensed, truncated, or simplified.

Selection
: A table view can provide the user with a column of choices. The user chooses by tapping a cell, which selects the cell; the app responds appropriately to that choice.

Navigation
: The appropriate response to the user's choosing a cell is often navigation to another interface. This might be done, for example, through a presented view controller or a navigation interface (Chapter 6). An extremely common configuration is a *master–detail interface*, where the master view is a table view within a navigation interface: the user taps a table view cell to navigate to the details about that cell. This is one reason why truncation of text in a table view cell is acceptable: the detail view contains the full information.

In addition to its column of cells, a table view can be extended by a number of other features that make it even more useful and flexible:

- A table can display a header view at the top and a footer view at the bottom.
- The cells can be clumped into sections. Each section can have a header and footer, and these remain visible as long as the section itself occupies the screen, giving the user a clue as to where we are within the table. Moreover, a section index can be provided, in the form of an overlay column of abbreviated section titles, which the user can tap or drag to jump to the start of a section, thus making a long table tractable.
- Tables can be editable: the user can be permitted to insert, delete, and reorder cells.
- A table can have a grouped format, with large section headers and footers that travel with their neighbor cells. This is often used for presenting small numbers of related cells; the headers and footers can provide ancillary information.

Table view cells, too, can be extremely flexible. Some basic cell formats are provided, such as a text label along with a small image view, but you are free to design your own cell as you would any other view. There are also some standard interface items that are commonly used in a cell, such as a checkmark to indicate selection or a right-pointing chevron to indicate that tapping the cell navigates to a detail view.

Figure 8-1 shows a familiar table view: Apple's Music app. Each table cell displays a song's name, artist, and album, in truncated form; the user can tap to play the song or to see further options. The table is divided into sections; as the user scrolls, the current section header stays pinned to the top of the table view. The table can also be navigated using the section index at the right.

Figure 8-2 shows a familiar grouped table: Apple's Settings app. It's a master–detail interface. The master view has sections, but they aren't labeled: they merely clump related topics. The detail view often has a single cell per section, using section headers and footers to explain what that cell does.

It would be difficult to overstate the importance of table views. An iOS app without a table view somewhere in its interface would be a rare thing, especially on the small iPhone screen. I've written apps consisting almost entirely of table views. Indeed, it is not uncommon to use a table view even in situations that have nothing particularly table-like about them, simply because it is so convenient.

For example, in one of my apps I want the user to be able to choose between three levels of difficulty and two sets of images. In a desktop application I'd probably use radio buttons; but there are no radio buttons among the standard iOS interface objects. Instead, I use a grouped table view so small that it doesn't even scroll. This gives me section headers, tappable cells, and a checkmark indicating the current choice (Figure 8-3).

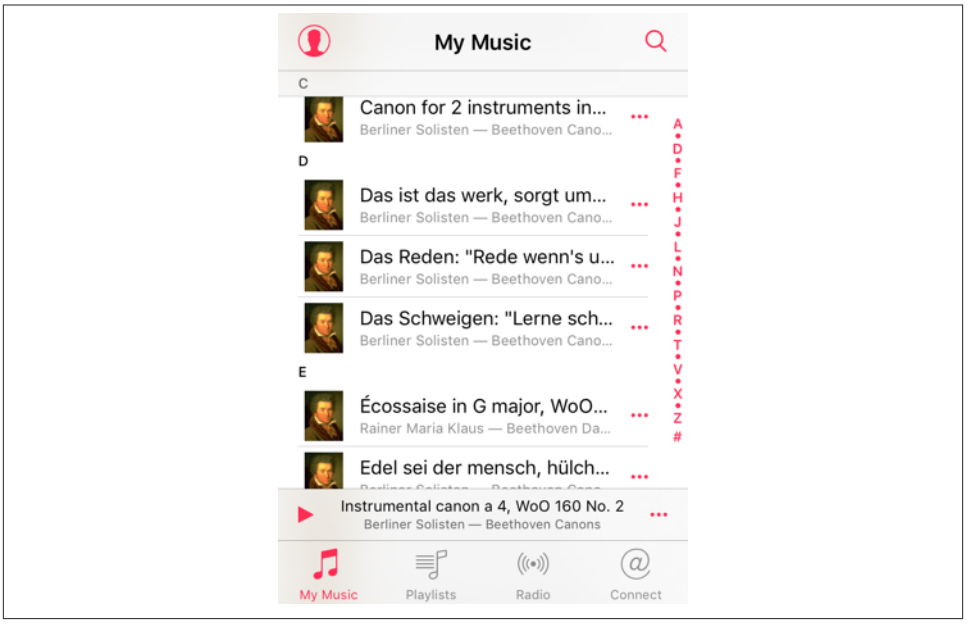

Figure 8-1. A familiar table view

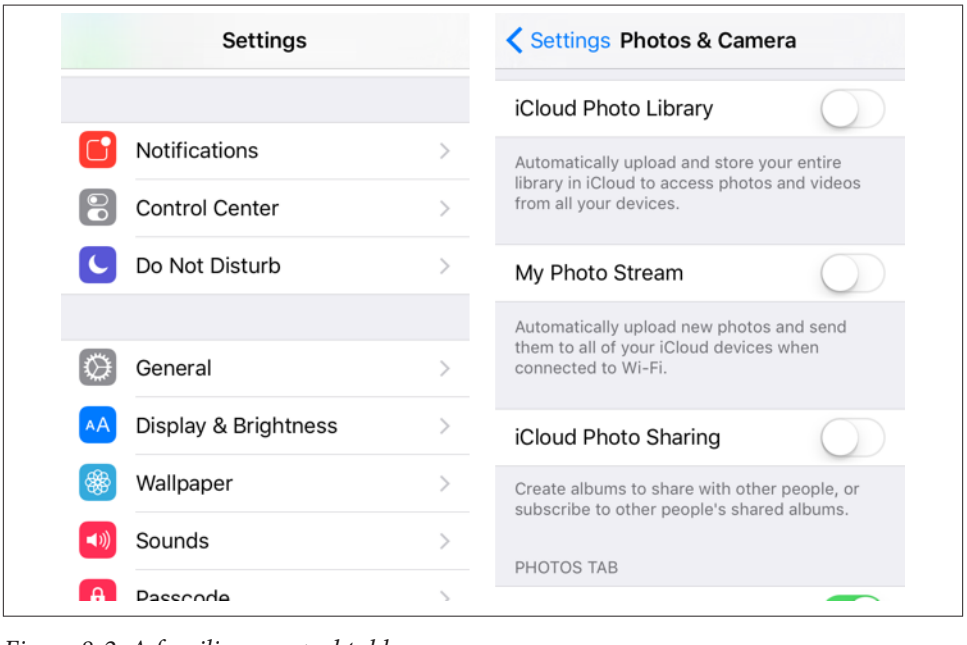

Figure 8-2. A familiar grouped table

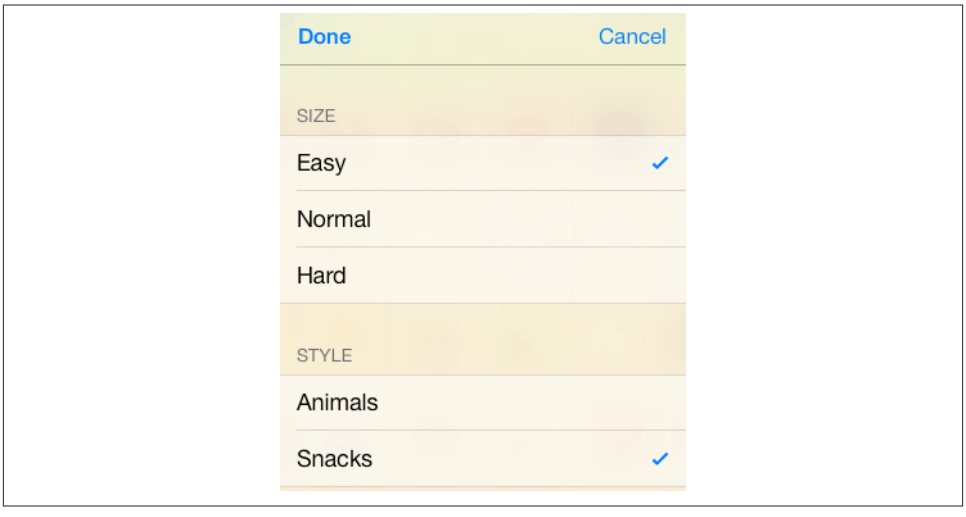

Figure 8-3. A grouped table view as an interface for choosing options

There is a UIViewController subclass, UITableViewController, whose main view is a table view. You never really *need* to use a UITableViewController; it's a convenience, but it doesn't do anything that you couldn't do yourself by other means. Here's some of what using a UITableViewController gives you:

- UITableViewController's `init(style:)` creates the table view with a plain or grouped format.
- The view controller is automatically made the table view's delegate and data source, unless you specify otherwise.
- The table view is made the view controller's `tableView`. It is also, of course, the view controller's `view`, but the `tableView` property is typed as a UITableView, so you can send table view messages to it without casting.

Table View Cells

Beginners may be surprised to learn that a table view's structure and contents are generally not configured in advance. Rather, you supply the table view with a data source and a delegate (which will often be the same object), and the table view turns to these in real time, as the app runs, whenever it needs a piece of information about its own structure and contents.

This architecture may sound odd, but in fact it is part of a brilliant strategy to conserve resources. Imagine a long table consisting of thousands of rows. It must appear, there-

fore, to consist of thousands of cells as the user scrolls. But a cell is a UIView and is memory-intensive; to maintain thousands of cells internally would put a terrible strain on memory. Therefore, the table typically maintains only as many cells as are showing simultaneously at any one moment (about twelve, let's say). As the user scrolls to reveal new cells, those cells are created on the spot; meanwhile, the cells that have been scrolled out of view are permitted to die.

That's ingenious, but wouldn't it be even cleverer if, instead of letting a cell die as it is scrolled *out* of view, it were whisked around to the other side and used again as one of the cells being scrolled *into* view? Yes, and in fact that's exactly what you're supposed to do. You do it by assigning each cell a *reuse identifier*.

As cells with a given reuse identifier are scrolled out of view, the table view maintains a bunch of them in a pile. As cells are scrolled into view, you ask the table view for a cell from that pile, specifying it by means of the reuse identifier. The table view hands an old used cell back to you, and now you can configure it as the cell that is about to be scrolled into view. Cells are thus *reused* to minimize not only the number of actual cells in existence at any one moment, but the number of actual cells *ever created*. A table of 1000 rows might very well never need to create more than about a dozen cells over the entire lifetime of the app.

To facilitate this architecture, your code must be prepared, on demand, to supply the table with pieces of requested data. Of these, the most important is the cell to be slotted into a given position. A position in the table is specified by means of an index path (NSIndexPath), a class used here to combine a section number with a row number, and is often referred to simply as a *row* of the table. Your data source object may at any moment be sent the message `tableView:cellForRowAtIndexPath:`, and must respond by returning the UITableViewCell to be displayed at that row of the table. And you must return it *fast*: the user is scrolling *now*, so the table needs the next cell *now*.

In this section, I'll discuss *what* you're going to be supplying — the table view cell. After that, I'll talk about *how* you supply it.

Built-In Cell Styles

The simplest way to obtain a table view cell is to start with one of the four built-in table view cell styles. To create a cell using a built-in style, call `init(style:reuseIdentifier:)`. The `reuseIdentifier:` is what allows cells previously assigned to rows that are no longer showing to be reused for cells that are; it will usually be the same for all cells in a table. Your choices of cell style (UITableViewCellStyle) are:

`.Default`
: The cell has a UILabel (its `textLabel`), with an optional UIImageView (its `imageView`) at the left. If there is no image, the label occupies the entire width of the cell.

`.Value1`
: The cell has two UILabels (its `textLabel` and its `detailTextLabel`) side by side, with an optional UIImageView (its `imageView`) at the left. The first label is left-aligned; the second label is right-aligned. If the first label's text is too long, the second label won't appear.

`.Value2`
: The cell has two UILabels (its `textLabel` and its `detailTextLabel`) side by side. No UIImageView will appear. The first label is right-aligned; the second label is left-aligned. The label sizes are fixed, and the text of either will be truncated if it's too long.

`.Subtitle`
: The cell has two UILabels (its `textLabel` and its `detailTextLabel`), one above the other, with an optional UIImageView (its `imageView`) at the left.

To experiment with the built-in cell styles, do this:

1. Start with an empty application without a storyboard (I explained how to do that at the start of Chapter 1).
2. Choose File → New → File and specify iOS → Source → Cocoa Touch Class. Click Next.
3. Make this class a UITableViewController subclass called RootViewController. The XIB checkbox should be checked; Xcode will create an eponymous *.xib* file containing a table view, correctly configured with its File's Owner as our RootViewController class. Click Next.
4. Make sure you're saving into the correct folder and group, and that the app target is checked. Click Create.

To get our table view into the interface, insert this line into AppDelegate's `application:didFinishLaunchingWithOptions:` at the appropriate spot:

```
self.window!.rootViewController = RootViewController()
```

Now modify the RootViewController class (which comes with a lot of templated code), as in Example 8-1. Run the app to see the world's simplest table (Figure 8-4).

Example 8-1. The world's simplest table

```
let cellIdentifier = "Cell"
override func numberOfSectionsInTableView(tableView: UITableView)
    -> Int {
        return 1 ❶
}
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
        return 20 ❷
```

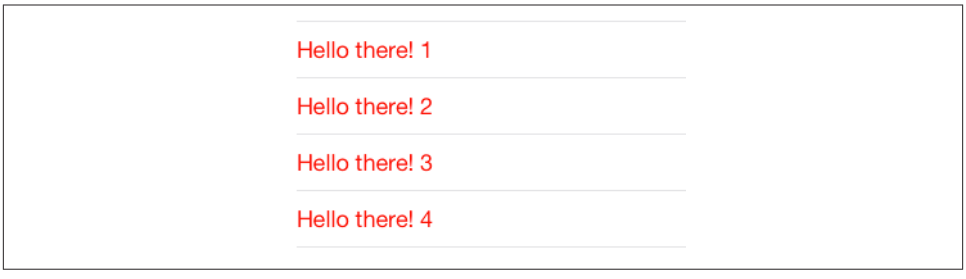

Figure 8-4. The world's simplest table

```
}
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        var cell : UITableViewCell! =
            tableView.dequeueReusableCellWithIdentifier(cellIdentifier) ❸
        if cell == nil {
            cell = UITableViewCell(
                style:.Default, reuseIdentifier:cellIdentifier) ❹
            cell.textLabel!.textColor = UIColor.redColor() ❺
        }
        cell.textLabel!.text = "Hello there! \(indexPath.row)" ❻
        return cell
}
```

The key parts of the code are:

❶ Our table will have one section.

❷ Our table will consist of 20 rows. Having multiple rows will give us a sense of how our cell looks when placed next to other cells.

❸ In `cellForRowAtIndexPath:`, you always start by asking the table view for a reusable cell. Here, we will receive either an already existing reused cell or `nil`; in the latter case, we must create the cell from scratch, ourselves.

❹ If we did receive `nil`, we do create the cell. This is where you specify the built-in table view cell style you want to experiment with.

❺ At this point in the code you can modify characteristics of the cell (`cell`) that are to be the same for *every* cell of the table. For the moment, I've symbolized this by assuming that every cell's text is to be the same color.

❻ We now have the cell to be used for *this* row of the table, so at this point in the code you can modify characteristics of the cell (`cell`) that are unique to this row. I've symbolized this by appending successive numbers to the text of each row. Of course, in real life the different cells would reflect meaningful data. I'll talk about that later in this chapter.

Now you can experiment with your cell's appearance by tweaking the code and running the app. Feel free to try different built-in cell styles in the place where we are now specifying `.Default`.

The flexibility of each built-in style is based mostly on the flexibility of UILabels. Not everything can be customized, because after you return the cell some further configuration takes place, which may override your settings. For example, the size and position of the cell's subviews are not up to you. (I'll explain, a little later, how to get around that.) But you get a remarkable degree of freedom. Here are a few basic UILabel properties for you to play with now (by customizing `cell.textLabel`), and I'll talk much more about UILabels in Chapter 10:

`text`
: The string shown in the label.

`textColor`, `highlightedTextColor`
: The color of the text. The `highlightedTextColor` applies when the cell is highlighted or selected (tap on a cell to select it).

`textAlignment`
: How the text is aligned; some possible choices (NSTextAlignment) are `.Left`, `.Center`, and `.Right`.

`numberOfLines`
: The maximum number of lines of text to appear in the label. Text that is long but permitted to wrap, or that contains explicit linefeed characters, can appear completely in the label if the label is tall enough and the number of permitted lines is sufficient. `0` means there's no maximum; the default is `1`.

`font`
: The label's font. You could reduce the font size as a way of fitting more text into the label. A font name includes its style. For example:

```
cell.textLabel!.font = UIFont(name:"Helvetica-Bold", size:12.0)
```

`shadowColor`, `shadowOffset`
: The text shadow. Adding a little shadow can increase clarity and emphasis for large text.

You can also assign the image view (`cell.imageView`) an image. The frame of the image view can't be changed, but you can inset its apparent size by supplying a smaller image and setting the image view's `contentMode` to `.Center`. It's probably a good idea in any case, for performance reasons, to supply images at their drawn size and resolution rather than making the drawing system scale them for you (see the last section of Chapter 7). For example:

```
let im = UIImage(named:"moi.png")!
UIGraphicsBeginImageContextWithOptions(CGSizeMake(36,36), true, 0.0)
im.drawInRect(CGRectMake(0,0,36,36))
let im2 = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
cell.imageView!.image = im2
cell.imageView!.contentMode = .Center
```

The cell itself also has some properties you can play with:

`accessoryType`
: A built-in type (UITableViewCellAccessoryType) of accessory view, which appears at the cell's right end. For example:

```
cell.accessoryType = .DisclosureIndicator
```

`accessoryView`
: Your own UIView, which appears at the cell's right end (overriding the `accessoryType`). For example:

```
let b = UIButton(type:.System)
b.setTitle("Tap Me", forState:.Normal)
b.sizeToFit()
// ... also assign button a target and action ...
cell.accessoryView = b
```

`indentationLevel`, `indentationWidth`
: These properties give the cell a left margin, useful for suggesting a hierarchy among cells. You can also set a cell's indentation level in real time, with respect to the table row into which it is slotted, by implementing the delegate's `tableView:indentationLevelForRowAtIndexPath:` method.

`separatorInset`
: A UIEdgeInsets. Only the left and right insets matter. The default is a left inset of `15`, but the built-in table view cell styles may shift it; for example, it's `16` for the `.Basic` cell style. This property affects both the drawing of the separator between cells and the indentation of content of the built-in cell styles.

  If you try to set the `separatorInset`, you may run up against the cell's layout margins. For example, setting a cell's `separatorInset` to `UIEdgeInsetsZero` results in a left inset of `8`. A workaround is to set the cell's `layoutMargins` to `UIEdgeInsetsZero` as well.

`selectionStyle`
: How the background looks when the cell is selected (UITableViewCellSelectionStyle). The default is solid gray (`.Default`), or you can choose `.None`.

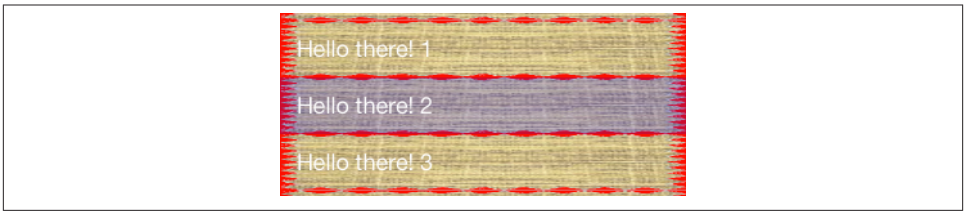

Figure 8-5. A cell with an image background

`backgroundColor`
`backgroundView`
`selectedBackgroundView`
: What's behind everything else drawn in the cell. The `selectedBackgroundView` is drawn in front of the `backgroundView` (if any) when the cell is selected, and will appear instead of whatever the `selectionStyle` dictates. The `backgroundColor` is behind the `backgroundView`. There is no need to set the frame of the `backgroundView` and `selectedBackgroundView`; they will be resized automatically to fit the cell.

`multipleSelectionBackgroundView`
: If defined (not `nil`), and if the table's `allowsMultipleSelection` (or, if editing, `allowsMultipleSelectionDuringEditing`) is `true`, used instead of the `selectedBackgroundView` when the cell is selected.

In this example, we set the cell's `backgroundView` to display an image with some transparency at the outside edges, so that the `backgroundColor` shows behind it, and we set the `selectedBackgroundView` to an almost transparent blue rectangle, to darken that image when the cell is selected (Figure 8-5):

```
cell.textLabel!.textColor = UIColor.whiteColor()
let v = UIImageView() // no need to set frame
v.contentMode = .ScaleToFill
v.image = UIImage(named:"linen.png")
cell.backgroundView = v
let v2 = UIView() // no need to set frame
v2.backgroundColor = UIColor.blueColor().colorWithAlphaComponent(0.2)
cell.selectedBackgroundView = v2;
cell.backgroundColor = UIColor.redColor()
```

If those features are to be true of every cell ever displayed in the table, then that code should go in the spot numbered 4 in Example 8-1; there's no need to waste time doing the same thing all over again when an existing cell is reused.

Finally, here are a few properties of the table view itself worth playing with:

`rowHeight`
: The height of a cell. A taller cell may accommodate more information. You can also change this value in the nib editor; the table view's row height appears in the Size inspector. The cell's subviews have their autoresizing set so as to compensate correctly. You can also set a cell's height in real time by implementing the delegate's `tableView:heightForRowAtIndexPath:` method; thus a table's cells may differ from one another in height (more about that later in this chapter).

`separatorStyle`, `separatorColor`, `separatorInset`
: These can also be set in the nib. The table's `separatorInset` is adopted by individual cells that don't have their own explicit `separatorInset`. Separator styles (UITableViewCellSeparatorStyle) are `.None` and `.SingleLine`.

`backgroundColor`, `backgroundView`
: What's behind all the cells of the table; this may be seen if the cells have transparency, or if the user scrolls the cells beyond their limit. The `backgroundView` is drawn on top of the `backgroundColor`.

`tableHeaderView`, `tableFooterView`
: Views to be shown before the first row and after the last row, respectively (as part of the table's scrolling content). Their background color is, by default, the background color of the table, but you can change that. You dictate their heights; their widths will be dynamically resized to fit the table. The user can, if you like, interact with these views (and their subviews); for example, a view can be (or can contain) a UIButton.

  You can alter a table header or footer view dynamically during the lifetime of the app; if you change its height, you must set the corresponding table view property afresh to notify the table view of what has happened.

Registering a Cell Class

In Example 8-1, I used this method to obtain the reusable cell:

- `dequeueReusableCellWithIdentifier:`

However, there's another way:

- `dequeueReusableCellWithIdentifier:forIndexPath:`

(The second parameter should always be the index path you received to begin with as the last parameter of `tableView:cellForRowAtIndexPath:`.)

Even though these two methods look nearly identical, they behave quite differently. The second method has three advantages:

The result is never `nil`
: Unlike `dequeueReusableCellWithIdentifier:`, the value returned by `dequeueReusableCellWithIdentifier:forIndexPath:` is never `nil` (in Swift, it isn't an Optional). If there is a free reusable cell with the given identifier, it is returned. If there isn't, a new one is created for you. Step 4 of Example 8-1 can thus be eliminated!

The cell size is known earlier
: Unlike `dequeueReusableCellWithIdentifier:`, the cell returned by `dequeueReusableCellWithIdentifier:forIndexPath:` has its final bounds. That's possible because you've passed the index path as an argument, so the runtime knows this cell's ultimate destination within the table, and has already consulted the table's `rowHeight` or the delegate's `tableView:heightForRowAtIndexPath:`. This makes laying out the cell's contents much easier.

The identifier is consistent
: A danger with `dequeueReusableCellWithIdentifier:` is that you may accidentally pass an incorrect reuse identifier, and end up not reusing cells. With `dequeueReusableCellWithIdentifier:forIndexPath:`, that can't happen.

Before you call `dequeueReusableCellWithIdentifier:forIndexPath:` for the first time, you must *register* with the table itself (unless the cell is coming from a storyboard, as I'll explain in a moment). You can do this by calling `registerClass:forCellReuseIdentifier:`. This associates a class (which must be UITableViewCell or a subclass thereof) with a string identifier. That's how `dequeueReusableCellWithIdentifier:forIndexPath:` knows what class to instantiate when it creates a new cell for you: you pass an identifier, and you've already told the table what class it signifies. The only cell types you can obtain are those for which you've registered in this way; if you pass a bad identifier, the app will crash (with a helpful log message).

This is a very elegant mechanism. It also raises some new questions:

When should I register with the table view?
: Do it early, before the table view starts generating cells; `viewDidLoad` is a good place:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.tableView.registerClass(
        UITableViewCell.self, forCellReuseIdentifier: "Cell")
}
```

How do I specify a built-in table view cell style?
: We are no longer calling `init(style:reuseIdentifier:)`, so where do we make our choice of built-in cell style? The default cell style is `.Default`, so if that's what you wanted, the problem is solved. Otherwise, subclass UITableViewCell and override `init(style:reuseIdentifier:)` to substitute the cell style you're after (passing along the reuse identifier you were handed).

For example, suppose we want the `.Subtitle` style. Let's call our UITableViewCell subclass MyCell. So we now specify `MyCell.self` in our call to `registerClass:forCellReuseIdentifier:`. MyCell's initializer looks like this:

```
override init(style: UITableViewCellStyle, reuseIdentifier: String?) {
    super.init(style:.Subtitle, reuseIdentifier: reuseIdentifier)
}
```

How do I know whether the returned cell is new or reused?

Good question! It's important to know this, because our call never returns `nil`, so we need some *other* way to distinguish between configurations that are to apply once and for all to a *new cell* (step 5 of Example 8-1) and configurations that differ for *each row* (step 6). The answer is: It's up to you, when performing one-time configuration on a cell, to give that cell some distinguishing mark that you can look for later to determine whether a cell requires one-time configuration.

For example, if every cell is to have a two-line text label, there is no point configuring the text label of *every* cell returned by `dequeueReusableCellWithIdentifier:forIndexPath:`; the reused cells have already been configured. But how will we know which cells need their text label to be configured? It's easy: they are the ones whose text label *hasn't* been configured:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath:indexPath) as! MyCell
        if cell.textLabel!.numberOfLines != 2 { // never configured
            cell.textLabel!.numberOfLines = 2
            // other one-time configurations here ...
        }
        cell.textLabel!.text = // ...
        // other individual configurations here ...
        return cell
}
```

Custom Cells

The built-in cell styles give the beginner a leg up in getting started with table views, but there is nothing sacred about them, and soon you'll probably want to transcend them, putting yourself in charge of how a table's cells look and what subviews they contain. You are perfectly free to do this. The thing to remember is that the cell has a `contentView` property, which is one of its subviews; things like the `accessoryView` are outside the `contentView`. All *your* customizations must be confined to subviews of the `contentView`; this allows the cell to continue working correctly.

I'll illustrate four possible approaches to customizing the contents of a cell:

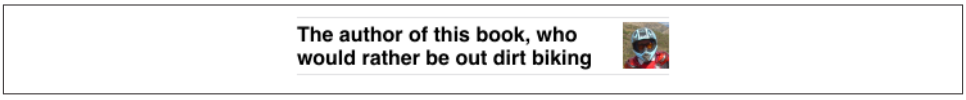

Figure 8-6. A cell with its label and image view swapped

- Start with a built-in cell style, but supply a UITableViewCell subclass and override `layoutSubviews` to alter the frames of the built-in subviews.
- In `tableView:cellForRowAtIndexPath:`, add subviews to each cell's `contentView` as the cell is created. This approach can be combined with the previous one, or you can ignore the built-in subviews and use your own exclusively.
- Design the cell in a nib, and load that nib in `tableView:cellForRowAtIndexPath:` each time a cell needs to be created.
- Design the cell in a storyboard.

What causes the built-in subviews to be present is not the cell style but the fact that you refer to them. As long as you never speak of the cell's `textLabel`, `detailTextLabel`, or `imageView`, they are never created or inserted into the cell. Thus, you don't need to remove them if you don't want to use them.

Overriding a cell's subview layout

You can't directly change the frame of a built-in cell style subview in `tableView:cellForRowAtIndexPath:`, because the cell's `layoutSubviews` comes along later and overrides your changes. The workaround is to override the cell's `layoutSubviews`! This is a straightforward solution if your main objection to a built-in style is the frame of an existing subview.

To illustrate, let's modify a `.Default` cell so that the image is at the right end instead of the left end (Figure 8-6). We'll make a UITableViewCell subclass, MyCell, remembering to register MyCell with the table view, so that `dequeueReusableCellWithIdentifier:forIndexPath:` produces a MyCell instance; here is MyCell's `layoutSubviews`:

```
override func layoutSubviews() {
    super.layoutSubviews()
    let cvb = self.contentView.bounds
    let imf = self.imageView!.frame
    self.imageView!.frame.origin.x = cvb.size.width - imf.size.width - 15
    self.textLabel!.frame.origin.x = 15
}
```

Adding subviews in code

Instead of modifying the existing default subviews, you can add completely new views to each UITableViewCell's content view. The best place to do this in code is `tableView:cellForRowAtIndexPath:`. Here are some things to keep in mind:

- The new views must be added when we instantiate a new cell, but not when we reuse a cell (because a reused cell already has them).
- We must never send `addSubview:` to the cell itself — only to its `contentView` (or some subview thereof).
- We should assign the new views an appropriate `autoresizingMask` or constraints, because the cell's content view might be resized.
- Each new view should be assigned a tag so that it can be identified and referred to elsewhere.

I'll rewrite the previous example (Figure 8-6) to use this technique. We are no longer using a UITableViewCell subclass; the registered cell class is UITableViewCell itself. If this is a new cell, we add the subviews and assign them tags. If this is a reused cell, we don't add the subviews (the cell already has them), and we use the tags to refer to the subviews:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath:indexPath)
        if cell.viewWithTag(1) == nil { // no subviews! add them
            let iv = UIImageView()
            iv.tag = 1
            cell.contentView.addSubview(iv)
            let lab = UILabel()
            lab.tag = 2
            cell.contentView.addSubview(lab)
            // since we are now adding the views ourselves,
            // we can use autolayout to lay them out
            let d = ["iv":iv, "lab":lab]
            iv.translatesAutoresizingMaskIntoConstraints = false
            lab.translatesAutoresizingMaskIntoConstraints = false
            var con = [NSLayoutConstraint]()
            // image view is vertically centered
            con.append(iv.centerYAnchor.constraintEqualToAnchor(
                cell.contentView.centerYAnchor))
            // it's a square
            con.append(iv.widthAnchor.constraintEqualToAnchor(
                iv.heightAnchor))
            // label has height pinned to superview
            con.appendContentsOf(
                NSLayoutConstraint.constraintsWithVisualFormat(
                    "V:|[lab]|",
```

```
                    options:[], metrics:nil, views:d))
            // horizontal margins
            con.appendContentsOf(
                NSLayoutConstraint.constraintsWithVisualFormat(
                    "H:|-15-[lab]-15-[iv]-15-|",
                    options:[], metrics:nil, views:d))
            NSLayoutConstraint.activateConstraints(con)
        }
        // can refer to subviews by their tags
        let lab = cell.viewWithTag(2) as! UILabel
        let iv = cell.viewWithTag(1) as! UIImageView
        // ...
        return cell
}
```

Designing a cell in a nib

We can avoid the verbosity of the previous code by designing the cell in a nib. To do so, we start by creating a *.xib* file that will consist, in effect, solely of this one cell. In Xcode, create the *.xib* file by specifying iOS → User Interface → View. Let's call it *MyCell.xib*. In the nib editor, delete the existing View and replace it with a Table View Cell from the Object library.

The cell's design window shows a standard-sized cell; you can resize it as desired, but the actual size of the cell in the interface will be dictated by the table view's width and its `rowHeight` (or the delegate's response to `tableView:heightForRowAtIndexPath:`). The cell already has a `contentView`, and any subviews you add will be inside that; do not subvert that arrangement.

You can choose a built-in table view cell style in the Style pop-up menu of the Attributes inspector, and this gives you the default subviews, locked in their standard positions; for example, if you choose Basic, the `textLabel` appears, and if you specify an image, the `imageView` appears. If you set the Style pop-up menu to Custom, you start with a blank slate. Let's do that.

We'll implement, from scratch, the same subviews we've already implemented in the preceding two examples: a UILabel on the left side of the cell, and a UIImageView on the right side. Just as when adding subviews in code, we should set each subview's autoresizing behavior or constraints, and *give each subview a tag*, so that later, in `tableView:cellForRowAtIndexPath:`, we'll be able to refer to the label and the image view using `viewWithTag:`, exactly as in the previous example:

```
let lab = cell.viewWithTag(2) as! UILabel
let iv = cell.viewWithTag(1) as! UIImageView
// ...
return cell
```

The only remaining question is how to load the cell from the nib. It's simple! When we register with the table view, which we're currently doing in `viewDidLoad`, instead of

calling `registerClass:forCellReuseIdentifier:`, we call a different method: `registerNib:forCellReuseIdentifier:`. To specify the nib, call UINib's class method `nibWithNibName:bundle:`, like this:

```
self.tableView.registerNib(
    UINib(nibName:"MyCell", bundle:nil), forCellReuseIdentifier: "Cell")
```

That's all there is to it. In `tableView:cellForRowAtIndexPath:`, when we call `dequeueReusableCellWithIdentifier:forIndexPath:`, if the table has no free reusable cell already in existence, the nib will automatically be loaded and the cell will be instantiated from it and returned to us.

You may wonder how that's possible, when we haven't specified a File's Owner class or added an outlet from the File's Owner to the cell in the nib. The answer is that the nib conforms to a specific format. The UINib instance method `instantiateWithOwner:options:` can load a nib with a `nil` owner; regardless, it returns an array of the nib's instantiated top-level objects. A nib registered with the table view is expected to have exactly one top-level object, and that top-level object is expected to be a UITableViewCell; that being so, the cell can easily be extracted from the resulting array, as it is the array's only element. Our nib meets those expectations!

The advantages of this approach should be immediately obvious. The subviews can now be designed in the nib, and code that was creating *and configuring* each subview can be deleted. For example, suppose we previously had this code:

```
if cell.viewWithTag(1) == nil {
    let iv = UIImageView()
    iv.tag = 1
    cell.contentView.addSubview(iv)
    let lab = UILabel()
    lab.tag = 2
    cell.contentView.addSubview(lab)
    // ... position views ...
    lab.font = UIFont(name:"Helvetica-Bold", size:16)
    lab.lineBreakMode = .ByWordWrapping
    lab.numberOfLines = 2
}
```

All of that can now be eliminated, including setting the label's `font`, `lineBreakMode`, and `numberOfLines`; those configurations are to be applied to the label in *every* cell, so they can be performed in the nib instead.

The nib *must* conform to the format I've described: it must have exactly one top-level object, a UITableViewCell. This means that some configurations are difficult or impossible in the nib. For example, a cell's `backgroundView` cannot be configured in the nib, because this would require the presence of a second top-level nib object. The simplest workaround is to add the `backgroundView` in code.

In `tableView:cellForRowAtIndexPath:`, we are still referring to the cell's subviews by way of `viewWithTag:`. There's nothing wrong with that, but perhaps you'd prefer to use names. Now that we're designing the cell in a nib, that's easy. Provide a UITableViewCell subclass with outlet properties, and configure the nib file accordingly:

1. Create a UITableViewCell subclass — let's call it MyCell — and declare two outlet properties:

    ```
    class MyCell : UITableViewCell {
        @IBOutlet var theLabel : UILabel!
        @IBOutlet var theImageView : UIImageView!
    }
    ```

    That is the *entirety* of MyCell's code; it exists solely so that we can create these outlets.
2. Edit the table view cell nib *MyCell.xib*. Change the class of the cell (in the Identity inspector) to MyCell, and connect the outlets from the cell to the respective subviews.

The result is that in our implementation of `tableView:cellForRowAtIndexPath:`, once we've typed the cell as a MyCell, the compiler will let us use the property names to access the subviews:

```
let cell = tableView.dequeueReusableCellWithIdentifier(
    "Cell", forIndexPath:indexPath) as! MyCell
let lab = cell.theLabel
let iv = cell.theImageView
```

Designing a cell in a storyboard

When your table view comes from a storyboard, it is open to you to employ any of the ways of obtaining and designing its cells that I've already described. There is also an additional option, available only in a UITableViewController scene in the storyboard: you can have the table view obtain the cells from the storyboard itself, and you can also *design* the cell directly in the table view in the storyboard.

To experiment with this way of obtaining and designing a cell, start with a project based on the Single View Application template:

1. In the storyboard, delete the View Controller scene. In the project, delete the View Controller class file.
2. In the project, create a file for a UITableViewController subclass called RootViewController, *without* a corresponding *.xib* file.
3. In the storyboard, drag a Table View Controller into the empty canvas, and set its class to RootViewController (and make sure it's the initial view controller).
4. The table view controller in the storyboard comes with a table view. In the storyboard, select that table view, and, in the Attributes inspector, set the Content pop-

up menu to Dynamic Prototypes, and set the number of Prototype Cells to 1 (these are the defaults).

The table view now contains a single table view cell with a content view. You can do in this cell exactly what we were doing before when designing a table view cell in a *.xib* file.

So, let's do that. I like being able to refer to my custom cell subviews with property names. Our procedure is just like what we did in the previous example:

1. In the project, add a UITableViewCell subclass — let's call it MyCell — and declare two outlet properties:

```
class MyCell : UITableViewCell {
    @IBOutlet var theLabel : UILabel!
    @IBOutlet var theImageView : UIImageView!
}
```

2. In the storyboard, select the prototype cell and change its class to MyCell.
3. Drag a label and an image view into the prototype cell, position and configure them as desired, and connect the cell's outlets to them appropriately.

So far, so good; but there is one crucial question I have not yet answered: How will your code tell the table view to get its cells from the storyboard? Clearly, *not* by calling `registerClass:forCellReuseIdentifier:`, and *not* by calling `registerNib:forCellReuseIdentifier:`; each of those would do something perfectly valid, but not the thing we want done in this case. The answer is that you *don't register anything at all* with the table view!_ Instead, when you call `dequeueReusableCellWithIdentifier:forIndexPath:`, you supply an identifier that matches the *prototype cell's identifier* in the storyboard.

So, return once more to the storyboard:

1. Select the prototype cell.
2. In the Attributes inspector, enter `Cell` in the Identifier field (capitalization counts).

Now RootViewController's `tableView:cellForRowAtIndexPath:` works exactly as it did in the previous example:

```
let cell = tableView.dequeueReusableCellWithIdentifier(
    "Cell", forIndexPath:indexPath) as! MyCell
let lab = cell.theLabel
let iv = cell.theImageView
```

If you call `dequeueReusableCellWithIdentifier:forIndexPath:` with an identifier that you have *not* registered with the table view and that *doesn't* match the identifier of a prototype cell in the storyboard, your app will crash (with a helpful message in the console).

Even though I've said it before, I'll say it again: You must *not* call `registerClass:forCellReuseIdentifier:` or `registerNib:forCellReuseIdentifier:` in this situation. If you do, you will be telling the runtime *not* to get the cell from the storyboard. This is a common beginner mistake.

Table View Data

The structure and content of the actual data portrayed in a table view comes from the data source, an object pointed to by the table view's `dataSource` property and adopting the UITableViewDataSource protocol. The data source is thus the heart and soul of the table. What surprises beginners is that the data source operates not by *setting* the table view's structure and content, but by *responding on demand*. The data source, *qua* data source, consists of a set of methods that the table view will call when it needs information; in effect, it will ask your data source some questions. This architecture has important consequences for how you write your code, which can be summarized by these simple guidelines:

Be ready
: Your data source cannot know *when* or *how often* any of these methods will be called, so it must be prepared to answer *any question at any time*.

Be fast
: The table view is asking for data in real time; the user is probably scrolling through the table *right now*. So you mustn't gum up the works; you must be ready to supply responses just as fast as you possibly can. (If you can't supply a piece of data fast enough, you may have to skip it, supply a placeholder, and insert the data into the table later. This may involve you in threading issues that I don't want to get into here. I'll give an example in Chapter 24.)

Be consistent
: There are multiple data source methods, and you cannot know *which* one will be called at a given moment. So you must make sure your responses are mutually consistent at *any* moment. For example, a common beginner error is forgetting to take into account, in your data source methods, the possibility that the data might not yet be ready.

This may sound daunting, but you'll be fine as long as you maintain an unswerving adherence to the principles of model–view–controller. How and when you accumulate the actual data, and how that data is structured, is a *model* concern. Acting as a data source is a *controller* concern. So you can acquire and arrange your data whenever and however you like, just so long as when the table view actually turns to you and asks what to do, you can lay your hands on the relevant data rapidly and consistently. You'll want to design the model in such a way that the controller can access any desired piece of data more or less instantly.

Another source of confusion for beginners is that methods are rather oddly distributed between the data source and the delegate, an object pointed to by the table view's `delegate` property and adopting the UITableViewDelegate protocol; in some cases, one may seem to be doing the job of the other. This is not usually a cause of any real difficulty, because the object serving as data source will probably also be the object serving as delegate. Nevertheless, it is rather inconvenient when you're consulting the documentation; you'll probably want to keep the data source and delegate documentation pages open simultaneously as you work.

If a table view's contents are known beforehand, you can alternatively design the entire table, *including the contents of individual cells*, in a storyboard. I'll give an example later in this chapter.

The Three Big Questions

Like Katherine Hepburn in *Pat and Mike*, the basis of your success (as a data source) is your ability, at any time, to answer the Three Big Questions. The questions the table view will ask you are a little different from the questions Mike asks Pat, but the principle is the same: know the answers, and be able to recite them at any moment. Here they are:

How many sections does this table have?
: The table will call `numberOfSectionsInTableView:`; respond with an integer. In theory you can sometimes omit this method, as the default response is `1`, which is often correct. However, I never omit it; for one thing, returning `0` is a good way to say that the table has no data, and will prevent the table view from asking any other questions.

How many rows does this section have?
: The table will call `tableView:numberOfRowsInSection:`. The table supplies a section number — the first section is numbered `0` — and you respond with an integer. In a table with only one section, of course, there is probably no need to examine the incoming section number.

What cell goes in this row of this section?
: The table will call `tableView:cellForRowAtIndexPath:`. The index path is expressed as an NSIndexPath; this is a sophisticated and powerful class, but you don't actually have to know anything about it, because UITableView provides a category on it that adds two read-only properties — `section` and `row`. Using these, you extract the requested section number and row number, and return a fully configured UITableViewCell, ready for display in the table view. The first row of a section is numbered `0`. I have already explained how to obtain the cell in the first place, by calling `dequeueReusableCellWithIdentifier:forIndexPath:`.

I have nothing particular to say about precisely how you're going to fulfill these obligations. It all depends on your data model and what your table is trying to portray. The important thing is to remember that you're going to be receiving an NSIndexPath specifying a section and a row, and you need to be able to lay your hands on the data corresponding to that slot *now* and configure the cell *now*. So construct your model, and your algorithm for consulting it in the Three Big Questions, and your way of configuring the cell, in accordance with that necessity.

For example, suppose our table is to list the names of the Pep Boys. Our data model might be an array of string names (`self.pep`). Our table has only one section. So our code might look like this:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    if self.pep == nil {
        return 0
    }
    return 1
}
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
        return self.pep.count
}
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath: indexPath)
        cell.textLabel!.text = pep[indexPath.row]
        return cell
}
```

At this point you may be feeling some exasperation. You want to object: "But that's trivial!" Exactly so! Your access to the data model *should* be trivial. That's the sign of a data model that's well designed for access by your table view's data source. Your implementation of `tableView:cellForRowAtIndexPath:` might have some interesting work to do in order to configure the *form* of the cell, but accessing the actual *data* should be simple and boring.

Reusing Cells

Another important goal of `tableView:cellForRowAtIndexPath:` should be to conserve resources by reusing cells. As I've already explained, once a cell's row is no longer visible on the screen, that cell can be slotted into a row that *is* visible — with its portrayed data appropriately modified, of course! — so that only a few more than the number of simultaneously visible cells will ever need to be instantiated.

A table view is ready to implement this strategy for you; all you have to do is call `dequeueReusableCellWithIdentifier:forIndexPath:`. For any given identifier, you'll be

handed either a newly minted cell or a reused cell that previously appeared in the table view but is now no longer needed because it has scrolled out of view.

The table view can maintain more than one cache of reusable cells; this could be useful if your table view contains more than one type of cell (where the meaning of the concept "type of cell" is pretty much up to you). This is why you must *name* each cache, by attaching an identifier string to any cell that can be reused. All the examples in this chapter (and in this book, and in fact in every UITableView I've ever created) use just one cache and just one identifier, but there can be more than one. If you're using a storyboard as a source of cells, there would then need to be more than one prototype cell.

To prove to yourself the efficiency of the cell-caching architecture, do something to differentiate newly instantiated cells from reused cells, and count the newly instantiated cells, like this:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}
override func tableView(
    tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return 1000 // make a lot of rows this time!
}
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath:indexPath) as! MyCell
        let lab = cell.theLabel
        // prove that many rows does not mean many cell objects
        lab.text = "Row \(indexPath.row) of section \(indexPath.section)"
        if lab.tag != 999 {
            lab.tag = 999
            print("New cell \(++self.cells)")
        }
}
```

When we run this code and scroll through the table, every cell is numbered correctly, so there appear to be 1000 cells. But the console messages show that only about a dozen distinct cells are ever actually created.

Be certain that *your* table view code passes that test, and that you are truly reusing cells! Fortunately, one of the benefits of calling `dequeueReusableCellWithIdentifier:forIndexPath:` is that it forces you to use a valid reuse identifier.

A common beginner error is to obtain a cell in some other way, such as instantiating it directly every time `tableView:cellForRowAtIndexPath:` is called. I have even seen beginners call `dequeueReusableCellWithIdentifier:forIndexPath:`, only to instantiate a fresh cell manually in the *next* line. Don't do that.

When your `tableView:cellForRowAtIndexPath:` implementation configures *individual* cells (step 6 in Example 8-1), the cell might be new or reused; at this point in your code, you don't know or care which. Therefore, you should always configure *everything* about the cell that might need configuring. If you fail to do this, and if the cell is reused, you might be surprised when some aspect of the cell is left over from its previous use; similarly, if you fail to do this, and if the cell is new, you might be surprised when some aspect of the cell isn't configured at all.

As usual, I learned that lesson the hard way. In the TidBITS News app, there is a little loudspeaker icon that should appear in a given cell in the master view's table view only if there is a recording associated with this article. So I initially wrote this code:

```
if (item.enclosures != nil) && (item.enclosures.count > 0) {
    cell.speaker.hidden = false
}
```

That turned out to be a mistake, because the cell might be reused. Every reused call *always* had a visible loudspeaker icon if, in a previous usage, that cell had *ever* had a visible loudspeaker icon! The solution was to rewrite the logic to cover all possibilities completely, like this:

```
cell.speaker.hidden =
    !((item.enclosures != nil) && (item.enclosures.count > 0))
```

You do get a sort of second bite of the cherry: there's a delegate method, `tableView:willDisplayCell:forRowAtIndexPath:`, that is called for every cell just before it appears in the table. This is absolutely the last minute to configure a cell. But don't misuse this method. You're functioning as the delegate here, not the data source; you may set the final details of the cell's appearance, but you shouldn't be consulting the data model at this point.

An additional delegate method is `tableView:didEndDisplayingCell:forRowAtIndexPath:`. This tells you that the cell no longer appears in the interface and has become free for reuse. You could take advantage of this to tear down any resource-heavy customization of the cell — I'll give an example in Chapter 24 — or simply to prepare it somehow for subsequent future reuse.

Table View Sections

Your table data may be expressed as divided into sections. You might clump your data into sections for various reasons (and doubtless there are other reasons beyond these):

- You want to supply section headers (or footers, or both).
- You want to make navigation of the table easier by supplying an index down the right side. You can't have an index without sections.

- You want to facilitate programmatic rearrangement of the table. For example, it's very easy to hide or move an entire section at once, possibly with animation.

Section headers and footers

A section header or footer appears between the cells, before the first row of a section or after the last row of a section, respectively. In a nongrouped table, a section header or footer detaches itself while the user scrolls the table, pinning itself to the top or bottom of the table view and floating over the scrolled rows, giving the user a sense, at every moment, of where we are within the table. Also, a section header or footer can contain custom views, so it's a place where you might put additional information, or even functional interface, such as a button the user can tap.

Don't confuse the section headers and footers with the header and footer of the table as a whole. The latter are view properties of the table view itself, its `tableHeaderView` and `tableFooterView`, discussed earlier in this chapter. The table header appears only when the table is scrolled all the way down; the table footer appears only when the table is scrolled all the way up.

The number of sections is determined by your reply to `numberOfSectionsInTableView:`. For each section, the table view will consult your data source and delegate to learn whether this section has a header or a footer, or both, or neither (the default).

The UITableViewHeaderFooterView class is a UIView subclass intended specifically for use as the view of a header or footer; much like a table view cell, it is reusable. It has the following properties:

`textLabel`
: Label (UILabel) for displaying the text of the header or footer.

`detailTextLabel`
: This label, if you set its text, appears only in a grouped style table.

`contentView`
: A subview of the header or footer, to which you can add custom subviews. If you do, you probably should not use the built-in `textLabel`; the `textLabel` is not inside the `contentView` and in a sense doesn't belong to you.

`backgroundView`
: Any view you want to assign. The `contentView` is in front of the `backgroundView`. The `contentView` has a clear background by default, so the `backgroundView` shows through. An opaque `contentView.backgroundColor`, on the other hand, would completely obscure the `backgroundView`.

If the `backgroundView` is `nil` (the default), the header or footer view will supply its own background view whose `backgroundColor` is derived (in some annoyingly unspecified way) from the table's `backgroundColor`.

Don't set a UITableViewHeaderFooterView's `backgroundColor`; instead, set the `backgroundColor` of its `contentView`, or assign a `backgroundView` and configure it as you like. Also, setting its `tintColor` has no effect. (This feels like a bug; the `tintColor` should affect the color of subviews, such as a UIButton's title, but it doesn't.)

There are two ways in which you can supply a header or footer. You can use both, but it is better to pick just one:

Header or footer title string

You implement the data source method `tableView:titleForHeaderInSection:` or `tableView:titleForFooterInSection:` (or both). Return `nil` to indicate that the given section has no header (or footer). The header or footer view itself is a UITableViewHeaderFooterView, and is reused automatically: there will be only as many as needed for simultaneous display on the screen. The string you supply becomes the view's `textLabel.text`.

(In a grouped style table, the string's capitalization may be changed. To avoid that, use the second way of supplying the header or footer.)

Header or footer view

You implement the delegate method `tableView:viewForHeaderInSection:` or `tableView:viewForFooterInSection:` (or both). The view you supply is used as the entire header or footer and is automatically resized to the table's width and the section header or footer height (I'll discuss how the height is determined in a moment).

You are not required to return a UITableViewHeaderFooterView, but you will probably want to, in order to take advantage of reusability. To do so, the procedure is much like making a cell reusable. You register beforehand with the table view by calling `registerClass:forHeaderFooterViewReuseIdentifier:`. To supply the reusable view, send the table view `dequeueReusableHeaderFooterViewWithIdentifier:`; the result will be either a newly instantiated view or a reused view.

You can then configure this view as desired. For example, you can set its `textLabel.text`, or you can give its `contentView` custom subviews. In the latter case, be sure to set proper autoresizing or constraints, so that the subviews will be positioned and sized appropriately when the view itself is resized.

The documentation lists a second way of registering a header or footer view for reuse — `registerNib:forHeaderFooterViewReuseIdentifier:`. But the nib editor's Object library doesn't include a UITableViewHeaderFooterView, so this method is useless.

In addition, two pairs of delegate methods permit you to perform final configurations on your header or footer views:

`tableView:willDisplayHeaderView:forSection:`
`tableView:willDisplayFooterView:forSection:`
: You can perform further configuration here, if desired. A useful possibility is to generate the default UITableViewHeaderFooterView by implementing `titleFor...` and then tweak its form slightly here. These delegate methods are matched by `tableView:didEndDisplayingHeaderView:forSection:` and `tableView:didEndDisplayingFooterView:forSection:`.

`tableView:heightForHeaderInSection:`
`tableView:heightForFooterInSection:`
: The runtime resizes your header or footer before displaying it. Its width will be the table view's width; its height will be the table view's `sectionHeaderHeight` or `sectionFooterHeight` unless you implement one of these methods to say otherwise. Returning `UITableViewAutomaticDimension` means `0` if `titleFor...` returns `nil` or the empty string (or isn't implemented); otherwise, it means the table view's `sectionHeaderHeight` or `sectionFooterHeight`. Be sure to dictate the height *somehow* or you might not see any headers (or footers).

Some lovely effects can be created by making use of the fact that a header or footer view in a nongrouped table will be further forward than the table's cells. For example, a header with transparency shows the cells as they scroll behind it; a header with a shadow casts that shadow on the adjacent cell.

When a header or footer view appears in the middle of the table view (between two table cells), there is a transparent gap behind it. If the header or footer view has some transparency, the table view's background is visible through this gap. You'll want to take this into account when planning your color scheme.

Section data

Clearly, a table that is to have sections may require some advance planning in the formation of its data model. The row data must somehow be clumped into sections, because you're going to be asked for a row *with respect to its section*. And, just as with a cell, a section title must be readily available so that it can be supplied quickly in real time. A structure that I commonly use is a pair of parallel arrays: an array of strings containing the section names, and an array of subarrays containing the data for each section.

For example, suppose we intend to display the names of all 50 U.S. states in alphabetical order as the rows of a table view, and that we wish to divide the table into sections according to the first letter of each state's name. Let's say I have the alphabetized list as a text file, which starts like this:

```
Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
Delaware
...
```

I have properties already initialized as empty arrays, waiting to hold the data model:

```
var sectionNames = [String]()
var sectionData = [[String]]()
```

I'll prepare the data model by loading the text file and walking through it, line by line, creating a new section name and a new subarray when I encounter a new first letter:

```
let s = try! String(contentsOfFile: NSBundle.mainBundle()
    .pathForResource("states", ofType: "txt")!,
    encoding: NSUTF8StringEncoding)
let states = s.componentsSeparatedByString("\n")
var previous = ""
for aState in states {
    // get the first letter
    let c = String(aState.characters.prefix(1))
    // only add a letter to sectionNames when it's a different letter
    if c != previous {
        previous = c
        self.sectionNames.append(c.uppercaseString)
        // and in that case also add new subarray to our array of subarrays
        self.sectionData.append([String]())
    }
    sectionData[sectionData.count-1].append(aState)
}
```

The value of this preparatory dance is evident when we are bombarded with questions from the table view about cells and headers; supplying the answers is trivial, just as it should be:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return self.sectionNames.count
}
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
        return self.sectionData[section].count
}
override func tableView(tableView: UITableView,
```

```
        cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
            let cell = tableView.dequeueReusableCellWithIdentifier(
                "Cell", forIndexPath: indexPath)
            let s = self.sectionData[indexPath.section][indexPath.row]
            cell.textLabel!.text = s
            return cell
    }
    override func tableView(tableView: UITableView,
        titleForHeaderInSection section: Int) -> String? {
            return self.sectionNames[section]
    }
```

Let's modify that example to illustrate customization of a header view. I've already registered my header identifier in `viewDidLoad`:

```
self.tableView.registerClass(UITableViewHeaderFooterView.self,
    forHeaderFooterViewReuseIdentifier: "Header")
```

Now, instead of `tableView:titleForHeaderInSection:`, I'll implement `tableView:viewForHeaderInSection:`. For completely new views, I'll place my own label and an image view inside the `contentView` and give them their basic configuration; then I'll perform individual configuration on all views, new or reused (very much like `tableView:cellForRowAtIndexPath:`). Note my deliberate misuse of the otherwise useless `tintColor` property to mark whether a view needs basic configuration:

```
override func tableView(tableView: UITableView,
    viewForHeaderInSection section: Int) -> UIView? {
        let h = tableView
            .dequeueReusableHeaderFooterViewWithIdentifier("Header")!
        if h.tintColor != UIColor.redColor() {
            h.tintColor = UIColor.redColor() // invisible marker, tee-hee
            h.backgroundView = UIView()
            h.backgroundView?.backgroundColor = UIColor.blackColor()
            let lab = UILabel()
            lab.tag = 1
            lab.font = UIFont(name:"Georgia-Bold", size:22)
            lab.textColor = UIColor.greenColor()
            lab.backgroundColor = UIColor.clearColor()
            h.contentView.addSubview(lab)
            let v = UIImageView()
            v.tag = 2
            v.backgroundColor = UIColor.blackColor()
            v.image = UIImage(named:"us_flag_small.gif")
            h.contentView.addSubview(v)
            lab.translatesAutoresizingMaskIntoConstraints = false
            v.translatesAutoresizingMaskIntoConstraints = false
            NSLayoutConstraint.activateConstraints([
                NSLayoutConstraint.constraintsWithVisualFormat(
                    "H:|-5-[lab(25)]-10-[v(40)]",
                    options:[], metrics:nil, views:["v":v, "lab":lab]),
                NSLayoutConstraint.constraintsWithVisualFormat(
                    "V:|[v]|",
```

```
                    options:[], metrics:nil, views:["v":v]),
                NSLayoutConstraint.constraintsWithVisualFormat(
                    "V:|[lab]|",
                    options:[], metrics:nil, views:["lab":lab])
            ].flatten().map{$0})
        }
        let lab = h.contentView.viewWithTag(1) as! UILabel
        lab.text = self.sectionNames[section]
        return h
    }
```

Section index

If your table view has the plain style, you can add an index down the right side of the table, where the user can tap or drag to jump to the start of a section — helpful for navigating long tables. To generate the index, implement the data source method `sectionIndexTitlesForTableView:`, returning an array of string titles to appear as entries in the index. For our list of state names, that's trivial once again, just as it should be:

```
override func sectionIndexTitlesForTableView(tableView: UITableView)
    -> [String]? {
        return self.sectionNames
}
```

The index can appear even if there are no section headers. It will appear only if the number of rows exceeds the table view's `sectionIndexMinimumDisplayRowCount` property value; the default is `0`, so the index is always displayed by default. You will want the index entries to be short — preferably just one character — because they will be partially obscuring the right edge of the table; plus, each cell's content view will shrink to compensate, so you're sacrificing some cell real estate.

You can modify three properties that affect the index's appearance:

`sectionIndexColor`
: The index text color.

`sectionIndexBackgroundColor`
: The index background color. I advise giving the index some background color, even if it is `clearColor`, because otherwise the index distorts the colors of what's behind it in a distracting way.

`sectionIndexTrackingBackgroundColor`
: The index background color while the user's finger is sliding over it. By default, it's the same as the `sectionIndexBackgroundColor`.

Normally, there will be a one-to-one correspondence between the index entries and the sections; when the user taps an index entry, the table jumps to the start of the corre-

sponding section. However, under certain circumstances you may want to customize this correspondence.

For example, suppose there are 100 sections, but there isn't room to display 100 index entries comfortably on the iPhone. The index will automatically curtail itself, omitting some index entries and inserting bullets to suggest the omission, but you might prefer to take charge of the situation.

To do so, supply a shorter index, and implement the data source method `tableView:sectionForSectionIndexTitle:atIndex:`, returning the number of the section to jump to. You are told both the title and the index number of the section index listing that the user chose, so you can use whichever is convenient.

Refreshing Table View Data

The table view has no direct connection to the underlying data. If you want the table view display to change because the underlying data have changed, you have to cause the table view to refresh itself; basically, you're requesting that the Big Questions be asked all over again. At first blush, this seems inefficient ("regenerate *all* the data??"); but it isn't. Remember, in a table that caches reusable cells, there are no cells of interest other than those actually showing in the table at this moment. Thus, having worked out the layout of the table through the section header and footer heights and row heights, the table has to regenerate only those cells that are actually visible.

You can cause the table data to be refreshed using any of several methods:

`reloadData`
: The table view will ask the Three Big Questions all over again, including heights of rows and section headers and footers, and the index, exactly as it does automatically when the table view first appears.

`reloadRowsAtIndexPaths:withRowAnimation:`
: The table view will ask the Three Big Questions all over again, including heights, but not index entries. Cells are requested only for visible cells among those you specify. The first parameter is an array of index paths; to form an index path, use the initializer `init(forRow:inSection:)`.

`reloadSections:withRowAnimation:`
: The table view will ask the Three Big Questions all over again, including heights of rows and section headers and footers, and the index. Cells, headers, and footers are requested only for visible elements of the sections you specify. The first parameter is an NSIndexSet.

The latter two methods can perform animations that cue the user as to what's changing. For the `rowAnimation:` argument, you'll pass one of the following (UITableViewRowAnimation):

`.Fade`
: The old fades into the new.

`.Right`, `.Left`, `.Top`, `.Bottom`
: The old slides out in the stated direction, and is replaced from the opposite direction.

`.None`
: No animation.

`.Middle`
: Hard to describe; it's a sort of venetian blind effect on each cell individually.

`.Automatic`
: The table view just "does the right thing." This is especially useful for grouped style tables, because if you pick the wrong animation, the display can look very funny as it proceeds.

If all you need to do is to refresh the index, call `reloadSectionIndexTitles`; this calls the data source's `sectionIndexTitlesForTableView:`.

If you want the table view to be laid out freshly without reloading *any* cells, send it `beginUpdates` immediately followed by `endUpdates`. The section and row structure of the table will be asked for, along with calculation of all heights, but no cells and no headers or footers are requested. This is useful as a way of alerting the table that its measurements have changed. It might be considered a misuse of an updates block (the real use of such a block is discussed later in this chapter); but Apple takes advantage of this trick in the Table View Animations and Gestures example, in which a pinch gesture is used to change a table's row height in real time, so it must be legal.

It is also possible to access and alter a table's individual cells directly. This can be a lightweight approach to refreshing the table, plus you can supply your own animation within the cell as it alters its appearance. It is important to bear in mind, however, that the cells are not the data (view is not model). If you change the content of a cell manually, make sure that you have also changed the model corresponding to it, so that the row will appear correctly if its data is reloaded later.

To do this, you need direct access to the cell you want to change. You'll probably want to make sure the cell is visible within the table view's bounds; nonvisible cells don't really exist (except as potential cells waiting in the reuse cache), and there's no point changing them manually, as they'll be changed when they are scrolled into view, through the usual call to `tableView:cellForRowAtIndexPath:`.

Here are some UITableView properties and methods that mediate between cells, rows, and visibility:

`visibleCells`
: An array of the cells actually showing within the table's bounds.

`indexPathsForVisibleRows`
: An array of the rows actually showing within the table's bounds.

`cellForRowAtIndexPath:`
: Returns a UITableViewCell if the table is maintaining a cell for the given row (typically because this is a visible row); otherwise, returns `nil`.

`indexPathForCell:`
: Given a cell obtained from the table view, returns the row into which it is slotted.

By the same token, you can get access to the views constituting headers and footers, by calling `headerViewForSection:` or `footerViewForSection:`. Thus you could modify a view directly. You should assume that if a section is returned by `indexPathsForVisibleRows`, its header or footer might be visible.

If you want to grant the user some interface for requesting that a table view be refreshed, you might like to use a UIRefreshControl. You aren't required to use this; it's just Apple's attempt to provide a standard interface. It is located behind the top of the scrolling part of the table view. To request a refresh, the user scrolls the table view downward to reveal the refresh control and holds long enough to indicate that this scrolling is deliberate. The refresh control then acknowledges visually that it is refreshing, and remains visible until refreshing is complete.

The refresh control is located *behind the table view's `backgroundView`*. If the table view has an opaque background view, the refresh control will be impossible to see.

To give a table view a refresh control, assign a UIRefreshControl to the table view controller's `refreshControl` property; to do this in the nib editor, set the table view controller's Refreshing pop-up menu to Enabled. A refresh control is a control (UIControl, Chapter 12), and you will want to hook its Value Changed event to an action method; you can do that in the nib editor by making an action connection, or you can do it in code. Here's an example of creating and configuring a refresh control entirely in code:

```
self.refreshControl = UIRefreshControl()
self.refreshControl!.addTarget(
    self, action: "doRefresh", forControlEvents: .ValueChanged)
```

Once a refresh control's action message has fired, the control remains visible and indicates by animation (similar to an activity indicator) that it is refreshing, until you send it the `endRefreshing` message:

```
@IBAction func doRefresh(sender: AnyObject) {
    // ... refresh ...
    (sender as! UIRefreshControl).endRefreshing()
}
```

You can initiate a refresh animation in code with `beginRefreshing`, but this does not fire the action message or display the refresh control; to display it, scroll the table view:

```
self.tableView.setContentOffset(
    CGPointMake(0, -self.refreshControl!.bounds.height), animated:true)
self.refreshControl!.beginRefreshing()
self.doRefresh(self.refreshControl!) // fire action message manually
```

A refresh control also has these properties:

`refreshing` *(read-only)*
: Whether the refresh control is refreshing.

`tintColor`
: The refresh control's color. It is *not* inherited from the view hierarchy (I regard this as a bug).

`attributedTitle`
: Styled text displayed below the refresh control's activity indicator. On attributed strings, see Chapter 10.

`backgroundColor` *(inherited from UIView)*
: If you give a table view controller's `refreshControl` a background color, that color completely covers the table view's own background color when the refresh control is revealed. For some reason, I find the drawing of the `attributedTitle` more reliable if the refresh control has a background color.

Variable Row Heights

Most tables have rows that are all the same height, as set by the table view's `rowHeight`. However, the delegate's `tableView:heightForRowAtIndexPath:` can be used to make different rows different heights. You can see an example in the TidBITS News app (Figure 6-1).

Back when I first wrote TidBITS News, variable row heights were possible but virtually unheard-of; I knew of no other app that was using them, and Apple provided no guidance, so I had to invent my own technique by sheer trial-and-error. There were three main challenges:

Measurement
: *What* should the height of a given row be?

Timing
: *When* should the determination of each row's height be made?

Layout
: How should the *subviews* of each cell be configured for its individual height?

Over the years since then, implementing variable row heights has become considerably easier. In iOS 6, with the advent of constraints, both measurement and layout became much simpler. In iOS 7, new table view properties made it possible to improve the timing. And iOS 8 permitted variable row heights to be implemented *automatically*, without your having to worry about any of these problems.

I will briefly describe four different approaches that I have used, in historical order. Perhaps you won't use any of the first three, because the automatic variable row heights feature makes them unnecessary; nevertheless, a basic understanding of them will give you an appreciation of what the fourth approach is doing for you. Besides, in my experience, the automatic variable row heights feature can be slow; for efficiency and speed, you might want to revert to one of the earlier techniques.

Manual row height measurement

The TidBITS News app, in its earliest implementation, works as follows. Each cell contains two labels. The *measurement* question is, then, given the content that each label will have in a particular cell in a particular row of the table, how tall should the cell be in order to accomodate both labels and their contents?

The cells don't use autolayout, so we have to measure them manually. The procedure is simple but somewhat laborious. The NSAttributedString method `boundingRectWithSize:options:context:` (Chapter 10) answers the question, "How tall would this text be if laid out at a fixed width?" Thus, for each cell, we must answer that question for each label, allow for any vertical spacing above the first label, below the second label, and between the labels, and sum the results.

Then, however, the question of *timing* intrudes. The problem is that the moment when `tableView:heightForRowAtIndexPath:` is called is very different from the moment when `tableView:cellForRowAtIndexPath:` is called. The runtime needs to know the heights of *everything* in the table immediately, before it starts asking for *any* cells. Thus, before we are asked `tableView:cellForRowAtIndexPath:` for even *one* row, we are asked `tableView:heightForRowAtIndexPath:` for *every* row.

In effect, this means we have to gather *all* the data and lay out *all* the cells before we can start showing the data in any *single* row. You can see why this can be problematic. We are being asked up front to measure the entire table, row by row. If that measurement takes a long time, the table view will remain blank during the calculation.

In addition, there is now a danger of duplicating our own work later on, during *layout* (in `tableView:cellForRowAtIndexPath:`, or perhaps in `tableView:willDisplayCell:forRowAtIndexPath:`); it appears we will ultimately be laying out every cell

twice, once when we're asked for all the heights initially, and again later when we're asked for an actual cell.

My solution is to start with an empty array of CGFloat stored in a property, `self.rowHeights`. (A single array is all that's needed, because the table has just one section; the row number can thus serve directly as an index into the array.) Once that array is constructed, it can be used to supply a requested height *instantly*.

Calculating a cell height requires me to lay out that cell in at least a theoretical way. Thus, I have a utility method that lays out a cell for a given row, using the actual data for that row; let's say its name is `setUpCell:forIndexPath:`. It takes a cell and an index path, lays out the cell, and returns the cell's required height as a CGFloat.

When the delegate's `tableView:heightForRowAtIndexPath:` is called, either this is the very first time it's been called or it isn't. Thus, either we've already constructed `self.rowHeights` or we haven't. If we haven't, we construct it, by immediately calling the `setUpCell:forIndexPath:` utility method for *every* row and storing each resulting height in `self.rowHeights`. I have no real cells at this point in the story, but I'm using a UITableViewCell subclass designed in a nib, so I simply load the nib directly and pull out the cell to use a model.

Now I'm ready to answer `tableView:heightForRowAtIndexPath:` for *any* row, *immediately* — all I have to do is return the appropriate value from the `self.rowHeights` array.

Finally, we come to `tableView:cellForRowAtIndexPath:`. Every time it is called, I call my `setUpCell:forIndexPath:` utility method *again* — but this time, I'm laying out the *real* cell (and ignoring the returned height value).

Measurement and layout with constraints

Constraints assist the process in two ways. Early in the process, in `tableView:heightForRowAtIndexPath:`, they perform the *measurement* for us. How do they do that? Well, if the cell is designed with constraints that ultimately pin every subview to the `contentView` in such a way as to size the `contentView` height unambiguously *from the inside out* — because every subview either is given explicit size constraints or else is the kind of view that has an implicit size based on its contents, like a label or an image view — then we can simply call `systemLayoutSizeFittingSize:` to tell us the resulting height of the cell.

Later in the process, when we come to `tableView:cellForRowAtIndexPath:`, constraints obviously help with *layout* of each cell, because that's what constraints do. Thanks to `dequeueReusableCellWithIdentifier:forIndexPath:`, the cell has the correct height, so the constraints are now determining the size of the subviews *from the outside in*.

The one danger to watch out for here is that a `.SingleLine` separator eats into the cell height. This can cause the height of the cell in real life to differ very slightly from its height as calculated by `systemLayoutSizeFittingSize:`. If you've overdetermined the subview constraints, this can result in a conflict among constraints. Careful use of lowered constraint priorities can solve this problem nicely if it arises (though it is simpler, in practice, to set the cell separator to `.None`).

I'll show the actual code from another app of mine that uses this technique. My `setUpCell:forIndexPath:` no longer needs to return a value; I hand it a reference to a cell, it sets up the cell, and now I can do whatever I like with that cell. If this is the model cell being used for measurement in `tableView:heightForRowAtIndexPath:`, I call `systemLayoutSizeFittingSize:` to get the height; if it's the real cell generated by dequeuing in `tableView:cellForRowAtIndexPath:`, I return it. Thus, `setUpCell:forIndexPath:` is extremely simple: it just configures the cell with actual data from the model:

```
func setUpCell(cell:UITableViewCell, forIndexPath indexPath:NSIndexPath) {
    let row = indexPath.row
    (cell.viewWithTag(1) as! UILabel).text = self.titles[row]
    (cell.viewWithTag(2) as! UILabel).text = self.artists[row]
    (cell.viewWithTag(3) as! UILabel).text = self.composers[row]
}
```

My `self.rowHeights` property is typed as `[CGFloat?]`, and has been initialized to an array the same size as my data model (`self.titles` and so on) with every element set to `nil`. My implemention of `tableView:heightForRowAtIndexPath:`: is called repeatedly (`self.titles.count` times, in fact) before the table is displayed; the *first* time it is called, I calculate *all* the row height values *once* and store them all in `self.rowHeights`:

```
override func tableView(tableView: UITableView,
    heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
        let ix = indexPath.row
        if self.rowHeights[ix] == nil {
            let objects = UINib(nibName: "TrackCell2", bundle: nil)
                .instantiateWithOwner(nil, options: nil)
            let cell = objects.first as! UITableViewCell
            for ix in 0..<self.rowHeights.count {
                let indexPath = NSIndexPath(forRow: ix, inSection: 0)
                self.setUpCell(cell, forIndexPath: indexPath)
                let v = cell.contentView
                let sz = v.systemLayoutSizeFittingSize(
                    UILayoutFittingCompressedSize)
                self.rowHeights[ix] = sz.height
            }
        }
        return self.rowHeights[ix]!
}
```

My `tableView:cellForRowAtIndexPath:` implementation is trivial, because `setUpCell:forIndexPath:` does all the real work:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "TrackCell", forIndexPath: indexPath)
        self.setUpCell(cell, forIndexPath:indexPath)
        return cell
}
```

Estimated height

In iOS 7, three new table view properties were introduced:

- `estimatedRowHeight`
- `estimatedSectionHeaderHeight`
- `estimatedSectionFooterHeight`

To accompany those, there are also three table view delegate methods:

- `tableView:estimatedHeightForRowAtIndexPath:`
- `tableView:estimatedHeightForHeaderInSection:`
- `tableView:estimatedHeightForFooterInSection:`

The purpose of these properties and methods is to reduce the amount of time spent calculating row heights at the outset. If you supply an estimated row height, for example, then when `tableView:heightForRowAtIndexPath:` is called repeatedly before the table is displayed, it is called *only for the visible cells* of the table; for the remaining cells, the *estimated* height is used. The runtime thus has enough information to lay out the entire table very quickly: in a table with 300 rows, you don't have to provide the real heights for all 300 rows up front — you only have to provide real heights for, say, the half dozen visible rows. The downside is that this layout is incorrect, and will have to be corrected later: as new rows are scrolled into view, `tableView:heightForRowAtIndexPath:` will be called *again* for those new rows, and the layout of the whole table will be revised accordingly.

Thus, using an estimated height changes the *timing* of when `tableView:heightForRowAtIndexPath:` is called. To illustrate, I'll revise the previous example to use estimated heights. The estimated height is set in `viewDidLoad`:

```
self.tableView.estimatedRowHeight = 75
```

Now in my `tableView:heightForRowAtIndexPath:` implementation, when I find that a requested height value in `self.rowHeights` is `nil`, I don't fill in *all* the values of

`self.rowHeights` — I fill in *just that one height*. It's simply a matter of removing the for loop:

```
override func tableView(tableView: UITableView,
    heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
        let ix = indexPath.row
        if self.rowHeights[ix] == nil {
            let objects = UINib(nibName: "TrackCell2", bundle: nil)
                .instantiateWithOwner(nil, options: nil)
            let cell = objects.first as! UITableViewCell
            let indexPath = NSIndexPath(forRow: ix, inSection: 0)
            self.setUpCell(cell, forIndexPath: indexPath)
            let v = cell.contentView
            let sz = v.systemLayoutSizeFittingSize(
                UILayoutFittingCompressedSize)
            self.rowHeights[ix] = sz.height
        }
        return self.rowHeights[ix]!
}
```

Automatic row height

In iOS 8, a completely automatic calculation of variable row heights was introduced. This, in effect, simply does automatically what I'm already doing in `tableView:heightForRowAtIndexPath:` in the preceding code: it relies upon autolayout for the calculation of each row's height, and it calculates and caches a row's height the first time it is needed, as it is about to appear on the screen.

To use this mechanism, first configure your cell using autolayout to determine the size of the `contentView` from the inside out. Now all you have to do is to set the table view's `estimatedRowHeight` and *don't* implement `tableView:heightForRowAtIndexPath:` at all!

Thus, to adopt this approach in my app, all I have to do at this point is delete my `tableView:heightForRowAtIndexPath:` implementation entirely.

Obviously, that's very easy: but easy does not necessarily mean best. There is also a question of performance. The four techniques I've outlined here run not only from oldest to newest but also from fastest to slowest. Manual layout is faster than calling `systemLayoutSizeFittingSize:`, and calculating the heights of all rows up front, though it may cause a longer pause initially, makes *scrolling* faster for the user because no row heights have to be calculated while scrolling. You will have to measure and decide which approach is most suitable.

And there's one more thing to watch out for. I said earlier that the cell returned to you from `dequeueReusableCellWithIdentifier:forIndexPath:` in your implementation of `tableView:cellForRowAtIndexPath:` already has its final size. But if you use automatic variable row heights, that's not true, because automatic calculation of a cell's height

can't take place until after the cell exists! Any code that relies on the cell having its final size in `tableView:cellForRowAtIndexPath:` will break when you switch to automatic variable row heights, and may need to be moved to `tableView:willDisplayCell:forRowAtIndexPath:`, where the final cell size has definitely been achieved.

Table View Cell Selection

A table view cell has a normal state, a highlighted state (according to its `highlighted` property), and a selected state (according to its `selected` property). It is possible to change these states directly, possibly with animation, by calling `setHighlighted:animated:` or `setSelected:animated:` on the cell. But you don't want to act behind the table's back, so you are more likely to manage selection through the table view, letting the table view manage and track the state of its cells.

Selection implies highlighting. When a cell is selected, it propagates the highlighted state down through its subviews by setting each subview's `highlighted` property if it has one. That is why a UILabel's `highlightedTextColor` applies when the cell is selected. Similarly, a UIImageView (such as the cell's `imageView`) can have a `highlightedImage` that is shown when the cell is selected, and a UIControl (such as a UIButton) takes on its `highlighted` state when the cell is selected.

One of the chief purposes of your table view is likely to be to let the user select a cell. This will be possible, provided you have not set the value of the table view's `allowsSelection` property to `false`. The user taps a cell, and the cell switches to its selected state. Table views can also permit the user to select multiple cells simultaneously. Set the table view's `allowsMultipleSelection` property to `true`. If the user taps an already selected cell, by default it stays selected if the table doesn't allow multiple selection, but is deselected if the table does allow multiple selection.

By default, being selected will mean that the cell is redrawn with a gray background view, but you can change this at the individual cell level, as I've already explained: you can set a cell's `selectedBackgroundView` (or `multipleSelectionBackgroundView`), or change its `selectionStyle`.

Managing Cell Selection

Your code can learn and manage the selection through these UITableView properties and instance methods:

`indexPathForSelectedRow`
`indexPathsForSelectedRows`
: These read-only properties report the currently selected row(s), or `nil` if there is no selection. Don't accidentally call the wrong one. For example, asking for `indexPathForSelectedRow` when the table view allows multiple selection gives a result

that will have you scratching your head in confusion. (As usual, I speak from experience.)

`selectRowAtIndexPath:animated:scrollPosition:`
: The animation involves fading in the selection, but the user may not see this unless the selected row is already visible. The last parameter dictates whether and how the table view should scroll to reveal the newly selected row; your choices (UITableViewScrollPosition) are `.Top`, `.Middle`, `.Bottom`, and `.None`. For the first three options, the table view scrolls (with animation, if the second parameter is `true`) so that the selected row is at the specified position among the visible cells. For `.None`, the table view does not scroll; if the selected row is not already visible, it does not become visible.

`deselectRowAtIndexPath:animated:`
: Deselects the given row (if it is selected); the optional animation involves fading out the selection. No automatic scrolling takes place.

To deselect *all* currently selected rows, call `selectRowAtIndexPath:animated:scrollPosition:` with a `nil` index path. Reloading a cell's data also deselects that cell, and calling `reloadData` deselects all selected rows.

Responding to Cell Selection

Response to user selection is through the table view's delegate:

- `tableView:shouldHighlightRowAtIndexPath:`
- `tableView:didHighlightRowAtIndexPath:`
- `tableView:didUnhighlightRowAtIndexPath:`
- `tableView:willSelectRowAtIndexPath:`
- `tableView:didSelectRowAtIndexPath:`
- `tableView:willDeselectRowAtIndexPath:`
- `tableView:didDeselectRowAtIndexPath:`

Despite their names, the two `will` methods are actually `should` methods and expect a return value:

- Return `nil` to prevent the selection (or deselection) from taking place.
- Return the index path handed in as argument to permit the selection (or deselection), or a different index path to cause a different cell to be selected (or deselected).

The `Highlight` methods are more sensibly named, and they arrive first, so you can return `false` from `tableView:shouldHighlightRowAtIndexPath:` to prevent a cell from being selected.

Let's focus in more detail on the relationship between a cell's highlighted state and its selected state. They are, in fact, two different states. When the user touches a cell, the cell passes through a complete highlight cycle. Then, if the touch turns out to be the beginning of a scroll motion, the cell is unhighlighted immediately, and the cell is not selected. Otherwise, the cell is unhighlighted and selected.

But the user doesn't know the difference between these two states: whether the cell is highlighted or selected, the cell's subviews are highlighted, and the `selectedBackgroundView` appears. Thus, if the user touches and scrolls, what the user sees is the flash of the `selectedBackgroundView` and the highlighted subviews, until the table begins to scroll and the cell returns to normal. If the user touches and lifts the finger, the `selectedBackgroundView` and highlighted subviews appear and remain; there is actually a moment in the sequence where the cell has been highlighted and then unhighlighted and not yet selected, but the user doesn't see any momentary unhighlighting of the cell, because no redraw moment occurs (see Chapter 4).

Here's a summary of the sequence:

1. The user's finger goes down. If `shouldHighlight` permits, the cell highlights, which propagates to its subviews. Then `didHighlight` arrives.
2. There is a redraw moment. Thus, the user will *see* the cell as highlighted (including the appearance of the `selectedBackgroundView`), regardless of what happens next.
3. The user either starts scrolling or lifts the finger. The cell unhighlights, which also propagates to its subviews, and `didUnhighlight` arrives.
    a. If the user starts scrolling, there is a redraw moment, so the user now sees the cell unhighlighted. The sequence ends.
    b. If the user merely lifts the finger, there is no redraw moment, so the cell keeps its highlighted appearance. The sequence continues.
4. If `willSelect` permits, the cell is selected, and `didSelect` arrives. The cell is *not* highlighted, but highlighting is propagated to its subviews.
5. There's another redraw moment. The user still sees the cell as highlighted (including the appearance of the `selectedBackgroundView`).

When `tableView:willSelectRowAtIndexPath:` is called because the user taps a cell, and if this table view permits only single cell selection, `tableView:willDeselectRowAtIndexPath:` will be called subsequently for any previously selected cells.

Here's an example of implementing `tableView:willSelectRowAtIndexPath:`. The default behavior for `allowsSelection` (not multiple selection) is that the user can select by tapping, and the cell remains selected; if the user taps a selected row, the selection does not change. We can alter this so that tapping a selected row deselects it:

```
override func tableView(tableView: UITableView,
    willSelectRowAtIndexPath indexPath: NSIndexPath) -> NSIndexPath? {
        if tableView.indexPathForSelectedRow == indexPath {
            tableView.deselectRowAtIndexPath(indexPath, animated:false)
            return nil
        }
        return indexPath
}
```

Navigation From a Table View

An extremely common response to user selection is navigation. A master–detail architecture is typical: the table view lists things the user can see in more detail, and a tap displays the detailed view of the selected thing. On the iPhone, very often the table view will be in a navigation interface, and you will respond to user selection by creating the detail view and pushing it onto the navigation controller's stack.

For example, here's the code from my Albumen app that navigates from the list of albums to the list of songs in the album that the user has tapped:

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
        let t = TracksViewController(
            mediaItemCollection: self.albums[indexPath.row])
        self.navigationController!.pushViewController(t, animated: true)
}
```

In a storyboard, when you draw a segue from a UITableViewCell, you are given a choice of two segue triggers: Selection Segue and Accessory Action. If you create a Selection Segue, the segue will be triggered when the user selects a cell. Thus you can readily push or present another view controller in response to cell selection.

If you're using a UITableViewController, then by default, whenever the table view appears, the selection is cleared automatically in `viewWillAppear:`, and the scroll indicators are flashed in `viewDidAppear:`. You can prevent this automatic clearing of the selection by setting the table view controller's `clearsSelectionOnViewWillAppear` to `false`. I sometimes do that, preferring to implement deselection in `viewDidAppear:`; the effect is that when the user returns to the table, the row is still momentarily selected before it deselects itself.

By convention, if selecting a table view cell causes navigation, the cell should be given an `accessoryType` (UITableViewCellAccessory) of `.DisclosureIndicator`. This is a plain gray right-pointing chevron at the right end of the cell. The chevron itself doesn't

respond to user interaction; it is not a button, but just a visual cue that the user can tap the cell to learn more.

Two additional `accessoryType` settings *are* buttons:

`.DetailButton`
: Drawn as a letter "i" in a circle.

`.DetailDisclosureButton`
: Drawn like `.DetailButton`, along with a disclosure indicator chevron to its right.

To respond to the tapping of an accessory button, implement the table view delegate's `tableView:accessoryButtonTappedForRowWithIndexPath:`. Or, in a storyboard, you can Control-drag a connection from a cell and choose an Accessory Action segue.

A common convention is that selecting the cell as a whole does one thing and tapping the detail button does something else. For example, in Apple's Phone app, tapping a contact's listing in the Recents table places a call to that contact, but tapping the detail button navigates to that contact's detail view.

Cell Choice and Static Tables

Another use of cell selection is to implement a choice among cells, where a section of a table effectively functions as an iOS alternative to OS X radio buttons. The table view usually has the grouped format. An `accessoryType` of `.Checkmark` is typically used to indicate the current choice. Implementing radio button behavior is up to you.

As an example, I'll implement the interface shown in Figure 8-3. The table view has the grouped style, with two sections. The first section, with a "Size" header, has three mutually exclusive choices: "Easy," "Normal," or "Hard." The second section, with a "Style" header, has two choices: "Animals" or "Snacks."

This is a *static table*; its contents are known beforehand and won't change. In a case like this, if we're using a UITableViewController subclass instantiated from a storyboard, the nib editor lets us design the entire table, including the headers and the cells *and their content*, directly in the storyboard. Select the table and set its Content pop-up menu in the Attributes inspector to Static Cells to make the table editable in this way (Figure 8-7).

Even though each cell is designed initially in the storyboard, I can still implement `tableView:cellForRowAtIndexPath:` to call `super` and then add further functionality. That's how I'll add the checkmarks. The user defaults are storing the current choice in each of the two categories; there's a `"Size"` preference and a `"Style"` preference, each consisting of a string denoting the title of the chosen cell:

```
override func tableView(tv: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = super.tableView(
            tv, cellForRowAtIndexPath:indexPath)
```

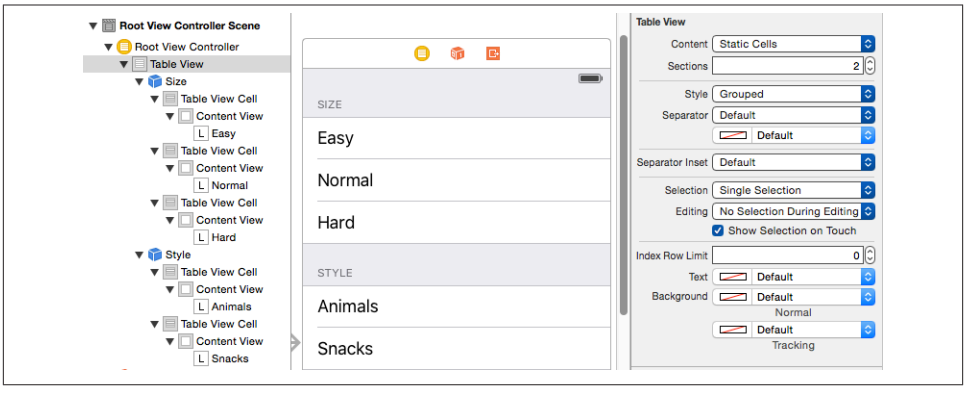

Figure 8-7. Designing a static table in the storyboard editor

```
        let ud = NSUserDefaults.standardUserDefaults()
        cell.accessoryType = .None
        if ud.valueForKey("Style") as? String == cell.textLabel!.text! ||
            ud.valueForKey("Size") as? String == cell.textLabel!.text! {
                cell.accessoryType = .Checkmark
        }
        return cell
}
```

When the user taps a cell, the cell is selected. I want the user to see that selection momentarily, as feedback, but then I want to deselect, adjusting the checkmarks so that that cell is the only one checked in its section. In `tableView:didSelectRowAtIndexPath:`, I set the user defaults, and then I reload the table view's data. This removes the selection and causes `tableView:cellForRowAtIndexPath:` to be called to adjust the checkmarks:

```
override func tableView(tv: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
        let ud = NSUserDefaults.standardUserDefaults()
        let setting = tv.cellForRowAtIndexPath(indexPath)!.textLabel!.text
        let header = self.tableView(
            tv, titleForHeaderInSection:indexPath.section)!
        ud.setValue(setting, forKey:header)
        tv.reloadData()
}
```

Table View Scrolling and Layout

A UITableView is a UIScrollView, so everything you already know about scroll views is applicable (Chapter 7). In addition, a table view supplies two convenience scrolling methods:

- `scrollToRowAtIndexPath:atScrollPosition:animated:`
- `scrollToNearestSelectedRowAtScrollPosition:animated:`

The `scrollPosition` parameter is as for `selectRowAtIndexPath:...`, discussed earlier in this chapter.

The following UITableView methods mediate between the table's bounds coordinates on the one hand and table structure on the other:

- `indexPathForRowAtPoint:`
- `indexPathsForRowsInRect:`
- `rectForSection:`
- `rectForRowAtIndexPath:`
- `rectForFooterInSection:`
- `rectForHeaderInSection:`

The table's own header and footer are direct subviews of the table view, so their positions within the table's bounds are given by their frames.

Table View State Restoration

If a UITableView participates in state saving and restoration (Chapter 6), the restoration mechanism would like to restore the selection and the scroll position. This behavior is automatic; the restoration mechanism knows both what cells should be visible and what cells should be selected, in terms of their index paths. If that's satisfactory, you've no further work to do.

In some apps, however, there is a possibility that when the app is relaunched, the underlying data may have been rearranged somehow. Perhaps what's meaningful in dictating what the user should see in such a case is not the previous *rows* but the previous *data*. The state saving and restoration mechanism doesn't know anything about the relationship between the cells and the underlying data. If you'd like to tell it, adopt the UIDataSourceModelAssociation protocol and implement two methods:

`modelIdentifierForElementAtIndexPath:inView:`
: Based on the index path, you return some string that you will *later* be able to use to identify uniquely this bit of model data.

`indexPathForElementWithModelIdentifier:inView:`
: Based on the unique identifier you provided earlier, you return the index path at which this bit of model data is displayed in the table *now*.

Devising a system of unique identification and incorporating it into your data model is up to you.

Table View Searching

A common need is to make a table view searchable, typically through a search field (a UISearchBar; see Chapter 12). A commonly used interface for presenting the results of such a search is a table view! Thus, in effect, entering characters in the search field appears to filter the original table.

This interface is managed through a UIViewController subclass, UISearchController. It is extremely important to understand, before I tell you about UISearchController, that it has nothing to do, *per se*, with table views! A table view is not the only thing you might want to search, and a table view is not the only way you might want to present the results of a search. UISearchController itself is completely agnostic about the form of those results. However, I'm introducing this class in a chapter about table views, so what I'm going to describe is the particular (and common) case of how to use a table view to present the results of searching a table view.

Configuring a Search Controller

Here are the steps for configuring a UISearchController:

1. Create *and retain* a UISearchController instance. To do so, you'll call the designated initializer, `init(searchResultsController:)`. The parameter is a view controller — a UIViewController subclass instance that you will have created for this purpose. The search controller will retain this view controller as a child view controller. When the time comes to display search results, the search controller will *present* itself as a presented view controller, with this view controller's view inside its own view; that is where the search results are to be displayed.
2. Assign to the search controller's `searchResultsUpdater` an object to be notified when the search results change. This must be an object adopting the UISearchResultsUpdating protocol, which means that it implements one method: `updateSearchResultsForSearchController:`. Very typically, this will be the same view controller that you passed as the `searchResultsController:` parameter when you initialized the search controller, but no law says that it has to be the same object or even that it has to be a view controller.
3. Acquire the search controller's `searchBar` and put it into the interface.

Thinking about these steps, you can see what the search controller is proposing to do for you — and what it *isn't* going to do for you. It isn't going to display the search results. It isn't going to manage the search results. It isn't even going to do any searching! It owns a search bar, which you have placed into the interface; and it's going to keep an eye on

that search bar. When the user taps in that search bar to begin searching, the search controller will respond by presenting itself and managing the view controller you specified. Then, as the user enters characters in the search bar, the search controller will keep calling the search results updater's `updateSearchResultsForSearchController:`. Finally, when the user taps the search bar's Cancel button, the search controller will dismiss itself.

A UISearchController has just a few other properties you might want to configure:

`dimsBackgroundDuringPresentation`
: Whether a "dimming view" should appear behind the search controller's own view. Defaults to `true`, but I'll give an example later where it needs to be set to `false`.

`hidesNavigationBarDuringPresentation`
: Whether a navigation bar, if present, should be hidden. The default is `true`.

A UISearchController can also be assigned a delegate (UISearchControllerDelegate), which is notified before and after presentation and dismissal (and has one more important ability that I'll mention a bit later).

The minimalistic nature of the search controller's behavior is exactly the source of its power and flexibility, because it leaves you the freedom to take care of the details: what searching means, and what displaying search results means, is up to you.

Using a Search Controller

I'll demonstrate several variations on the theme of using a search controller to make a table view searchable. In this example, searching will mean finding the search bar text within the text displayed in the table view's cells.

Minimal search results table

Let's start with the simplest possible case. We will have two table view controllers — one managing the original table view, the other managing the search results table view. The original table view can be elaborate; we'll use the table of U.S. states, with sections and an index, developed earlier in this chapter. The search results table view will be as minimal as possible: I propose to use a rock-bottom table view with `.Default` style cells, where each search result will be the `text` of a cell's `textLabel` (Figure 8-8).

Here's the configuration of the original table's UITableViewController. I have a property, `self.searcher`, waiting to retain the search controller. I also have a second UITableViewController subclass, which I have rather boringly called SearchResultsController, whose job will be to obtain and present the search results. In `viewDidLoad`, I instantiate SearchResultsController, create the UISearchController, and put its search bar into the interface as the table view's header view (and scroll to hide that search bar initially, a common convention):

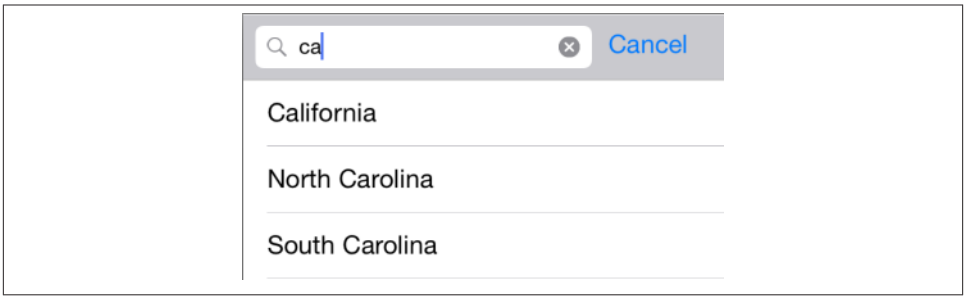

Figure 8-8. Searching a table

```
let src = SearchResultsController(data: self.sectionData)
let searcher = UISearchController(searchResultsController: src)
self.searcher = searcher
searcher.searchResultsUpdater = src
let b = searcher.searchBar
b.sizeToFit() // crucial, trust me on this one
b.autocapitalizationType = .None
self.tableView.tableHeaderView = b
self.tableView.reloadData()
self.tableView.scrollToRowAtIndexPath(
    NSIndexPath(forRow: 0, inSection: 0),
    atScrollPosition:.Top, animated:false)
```

Adding the search bar as the table view's header view has an odd side effect: it causes the table view's background color to be covered by an ugly gray color, visible above the search bar when the user scrolls down. The official workaround is to assign the table view a `backgroundView` with the desired color.

Now we turn to SearchResultsController. It is a completely vanilla table view controller, *qua* table view controller. But I've given it two special features:

- It is capable of *receiving the searchable data*. You can see this happening, in fact, in the first line of the preceding code.
- It is capable of *filtering* that data and displaying the filtered data in its table view.

I'm not using sections in the SearchResultsController's table, so it will simplify things if, as I receive the searchable data in the SearchResultsController, I flatten it from an array of arrays to a simple array:

```
init(data:[[String]]) {
    self.originalData = data.flatten().map{$0}
    super.init(nibName: nil, bundle: nil)
}
```

I have stored the flattened data in the `self.originalData` array, but what I display in the table view is a *different* array, `self.filteredData`. This is initially empty, because there are no search results until the user starts typing in the search field:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
        return self.filteredData.count
}
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath: indexPath)
        cell.textLabel!.text = self.filteredData[indexPath.row]
        return cell
}
```

All of that is sheer boilerplate and is perfectly obvious; but how does our search results table go from being empty to displaying any search results? That's the second special feature of SearchResultsController. It adopts UISearchResultsUpdating, so it implements `updateSearchResultsForSearchController:`. In that implementation, it uses the current text of the search controller's `searchBar` to filter `self.originalData` into `self.filteredData` and reloads the table view:

```
func updateSearchResultsForSearchController(
    searchController: UISearchController) {
        let sb = searchController.searchBar
        let target = sb.text!
        self.filteredData = self.originalData.filter {
            s in
            let options = NSStringCompareOptions.CaseInsensitiveSearch
            let found = s.rangeOfString(target, options: options)
            return (found != nil)
        }
        self.tableView.reloadData()
}
```

That's all! Of course, it's an artificially simple example; I'm describing the interface and the use of a UISearchController, not a real app. In real life you would presumably want to allow the user to *do* something with the search results, perhaps by tapping on a cell in the search results table.

Search bar scope buttons

If we wanted our search bar to have scope buttons, we would set its `scopeButtonTitles` immediately after calling `sizeToFit` in the preceding code:

```
let b = searcher.searchBar
b.sizeToFit() // crucial, trust me on this one
b.scopeButtonTitles = ["Starts", "Contains"]
```

The scope buttons don't appear in the table header view, but they do appear when the search controller presents itself. However, the search controller does not automatically call us back in `updateSearchResultsForSearchController:` when the user taps on a scope button. I regard that as a bug, but it's easy to work around it: we must simply make ourselves the search bar's `delegate`, so as to be notified through the delegate method `searchBar:selectedScopeButtonIndexDidChange:` — which can then turn right around and call `updateSearchResultsForSearchController:` (provided it has a reference to the search controller, which is easy to arrange beforehand). Here, I'll make our SearchResultsController respond to the distinction that a state name either starts with or contains the search text:

```
func updateSearchResultsForSearchController(
    searchController: UISearchController) {
        self.searchController = searchController // weak reference
        let sb = searchController.searchBar
        let target = sb.text!
        self.filteredData = self.originalData.filter {
            s in
            var options = NSStringCompareOptions.CaseInsensitiveSearch
            // we now have scope buttons; 0 means "starts with"
            if searchController.searchBar.selectedScopeButtonIndex == 0 {
                options.insert(.AnchoredSearch)
            }
            let found = s.rangeOfString(target, options: options)
            return (found != nil)
        }
        self.tableView.reloadData()
}
func searchBar(searchBar: UISearchBar,
    selectedScopeButtonIndexDidChange selectedScope: Int) {
        self.updateSearchResultsForSearchController(self.searchController!)
}
```

Customizing the presentation

The search controller is a presented view controller. This means that the presentation is customizable, as I explained in "Custom Presented View Controller Transition" on page 326. In particular:

- You can customize the presentation *animation* by setting the search controller's `transitioningDelegate`. As part of customizing the animation, you can (in fact you must) take charge of *placing the search bar* into the search controller's view; of course you can animate this however you like.

- The search controller's presentation controller is a normal presentation controller, and thus can be adaptive.
- If the search controller's delegate implements `presentSearchController:`, the very act of *presenting the search controller* is left up to you. If you don't call `presentViewController:animated:completion:` here, it will be called for you, but this is your chance to perform preparatory customizations, to add a completion handler, to present without animation, and so on.

In this excerpt from the animation controller's `animateTransition:`, I cause the search bar to appear during the presentation by sliding it down from above:

```
if let v2 = v2 { // presenting, vc2 is the search controller
    con.addSubview(v2) // con is the container view
    v2.frame = r2end
    let sc = vc2 as! UISearchController
    let sb = sc.searchBar
    sb.removeFromSuperview() // take it out of the original table view
    sb.showsScopeBar = true
    sb.sizeToFit()
    v2.addSubview(sb)
    sb.frame.origin.y = -sb.frame.height
    UIView.animateWithDuration(0.3, animations: {
        sb.frame.origin.y = 0
        }, completion: {
            _ in
            sb.setShowsCancelButton(true, animated: true)
            transitionContext.completeTransition(true)
        })
}
```

Search bar in navigation bar

No law says that you have to put the UISearchController's search bar into a table view's header view. Another common interface is for the search bar to appear in a navigation bar at the top of the screen. For example, assuming we are already in a navigation interface, you might make the search bar your view controller's `navigationItem.titleView`. You won't want the navigation bar to vanish when the user searches, so you'll set the search controller's `hidesNavigationBarDuringPresentation` to `false`. To prevent the presented search controller's view from covering the navigation bar, set your view controller's `definesPresentationContext` to `true`; the presented search controller's view will cover your view controller's view, but not the navigation bar, which belongs to the navigation controller's view:

```
let src = SearchResultsController(data: self.sectionData)
let searcher = UISearchController(searchResultsController: src)
self.searcher = searcher
searcher.searchResultsUpdater = src
let b = searcher.searchBar
```

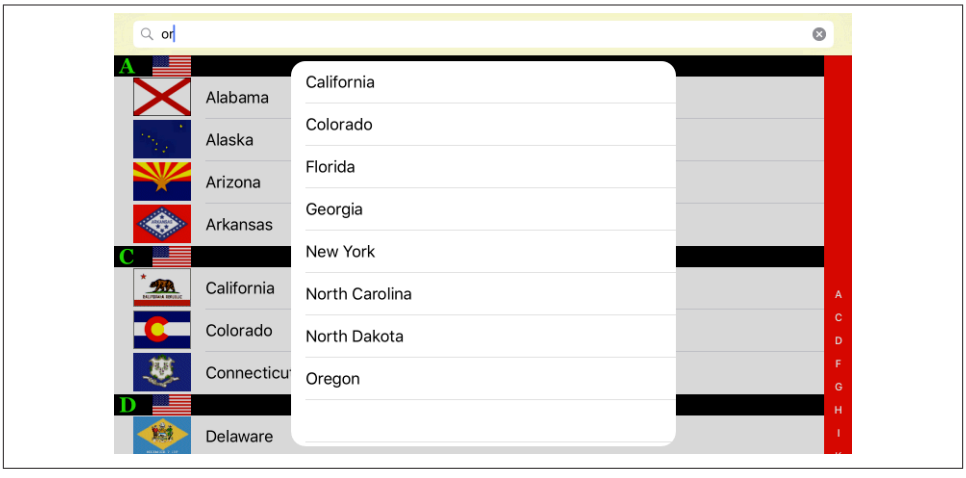

Figure 8-9. Searching from a navigation bar

```
b.sizeToFit()
b.autocapitalizationType = .None
self.navigationItem.titleView = b // *
searcher.hidesNavigationBarDuringPresentation = false // *
self.definesPresentationContext = true // *
```

An interesting question is how the search results should be displayed on an iPad. The usual thing is that they should pop down from the search bar as the user types. In iOS 7 and before, this was typically done by means of a popover (Chapter 9), but starting in iOS 8, Apple seems to prefer a small view that floats in front of everything. Apple's Safari app on the iPad is a familiar case in point. I'll imitate that interface (Figure 8-9).

It's a surprisingly tricky interface to achieve. The UISearchController is a presented view controller, so I would have expected to be able to customize its view position and chrome by setting its `modalPresentationStyle` to `.Custom` and injecting my own custom UIPresentationController; but I can't find a place to do that, because for some reason the transitioning delegate's `presentationControllerForPresentedViewController:...` is never called.

So I've resorted to a different strategy: the UISearchController's `searchResultsController` is a custom parent view controller with a transparent view, and its child view controller is the UITableViewController whose table view will display the search results:

```
class SearchResultsController : UIViewController {
    let child : ChildViewController // a UITableViewController
    init(data:[[String]]) {
        self.child = ChildViewController(data:data)
        super.init(nibName:nil, bundle:nil)
```

```
        }
        required init?(coder aDecoder: NSCoder) {
            fatalError("init(coder:) has not been implemented")
        }
        override func viewDidLoad() {
            self.automaticallyAdjustsScrollViewInsets = false
            self.view.backgroundColor = UIColor.clearColor()
            self.addChildViewController(self.child)
            let v = self.child.view
            self.view.addSubview(self.child.view)
            v.layer.cornerRadius = 15
            v.translatesAutoresizingMaskIntoConstraints = false
            NSLayoutConstraint.activateConstraints([
                v.heightAnchor.constraintEqualToConstant(400),
                v.widthAnchor.constraintEqualToConstant(400),
                v.topAnchor.constraintEqualToAnchor(
                    self.view.topAnchor, constant: 50),
                v.centerXAnchor.constraintEqualToAnchor(self.view.centerXAnchor)
            ])
            self.child.didMoveToParentViewController(self)
            let t = UITapGestureRecognizer(target: self, action: "tap:")
            t.delegate = self
            self.view.addGestureRecognizer(t)
        }
    }
```

The purpose of the tap gesture recognizer is so that a tap on the background can dismiss the search controller's view. We respond to the tap by walking up the responder chain until we come to our parent, the UISearchController; setting its `active` to `false` dismisses it. We don't want this to happen if the tap is on the results table itself, so we act as the tap gesture recognizer's delegate to prevent it:

```
    extension SearchResultsController : UIGestureRecognizerDelegate {
        func tap(g:UITapGestureRecognizer) {
            var r : UIResponder = g.view!
            while !(r is UISearchController) {r = r.nextResponder()!}
            (r as! UISearchController).active = false
        }
        func gestureRecognizerShouldBegin(g: UIGestureRecognizer) -> Bool {
            let pt = g.locationOfTouch(0, inView: self.child.view)
            if self.child.tableView.pointInside(pt, withEvent: nil) {
                return false
            }
            return true
        }
    }
```

Finally, the child table view controller adopts the UISearchResultsUpdating protocol, and we pass any `updateSearchResultsForSearchController:` calls along to it:

```
extension SearchResultsController : UISearchResultsUpdating {
    func updateSearchResultsForSearchController(sc: UISearchController) {
        self.child.updateSearchResultsForSearchController(sc)
    }
}
```

No secondary search results view controller

As a final variation, I'll demonstrate how to use a search controller *without* a distinct search results view controller. There will be no SearchResultsController; instead, we'll present the search results *in the original table view*.

To configure our search controller, we pass `nil` as its `searchResultsController` and set ourselves as the `searchResultsUpdater`. We also set the search controller's `dimsBackgroundDuringPresentation` to `false`; this allows the original table view to remain *visible and touchable* behind the search controller's view:

```
let searcher = UISearchController(searchResultsController:nil)
self.searcher = searcher
searcher.dimsBackgroundDuringPresentation = false
searcher.searchResultsUpdater = self
searcher.delegate = self
```

The implementation is a simple problem in table data source management. We keep an immutable copy of our data model arrays, `self.sectionData` and `self.sectionNames` — let's call the copies `self.originalSectionData` and `self.originalSectionNames`. These copies are unused if we're not searching. If we *are* searching, we hear about it through the search controller's delegate methods, and we raise a Bool flag in a property:

```
func willPresentSearchController(searchController: UISearchController) {
    self.searching = true
}
func willDismissSearchController(searchController: UISearchController) {
    self.searching = false
}
```

Any of our table view delegate or data source methods can consult this flag. For example, it might be nice to remove the index while searching is going on:

```
override func sectionIndexTitlesForTableView(tableView: UITableView)
    -> [String]? {
        return self.searching ? nil : self.sectionNames
}
```

All that remains is to implement `updateSearchResultsForSearchController:` to filter `self.originalSectionData` and `self.originalSectionNames` into the data model arrays `self.sectionData` and `self.sectionNames` — or to copy them unfiltered if the search bar's text is empty, which is also the signal that the search controller presentation is over:

```
func updateSearchResultsForSearchController(
    searchController: UISearchController) {
        let sb = searchController.searchBar
        let target = sb.text
        if target == "" {
            self.sectionNames = self.originalSectionNames
            self.sectionData = self.originalSectionData
            self.tableView.reloadData()
            return
        }
        // we have a target string
        self.sectionData = self.originalSectionData.map {
            $0.filter {
                let options = NSStringCompareOptions.CaseInsensitiveSearch
                let found = $0.rangeOfString(target, options: options)
                return (found != nil)
            }
        }.filter {$0.count > 0}
        self.sectionNames =
            self.sectionData.map {String($0[0].characters.prefix(1))}
        self.tableView.reloadData()
}
```

Table View Editing

A table view cell has a normal state and an editing state, according to its `editing` property. The editing state (or *edit mode*) is typically indicated visually by one or more of the following:

Editing controls
: At least one editing control will usually appear, such as a Minus button (for deletion) at the left side.

Shrinkage
: The content of the cell will usually shrink to allow room for an editing control. If there is no editing control, you can prevent a cell shifting its left end rightward in edit mode with the table delegate's `tableView:shouldIndentWhileEditingRowAtIndexPath:`. (There is also a cell property `shouldIndentWhileEditing`, but I find it unreliable.)

Changing accessory view
: The cell's accessory view will change automatically in accordance with its `editingAccessoryType` or `editingAccessoryView`. If you assign neither, so that they are `nil`, the cell's existing accessory view will vanish when in edit mode.

As with selection, you could set a cell's `editing` property directly (or use `setEditing:animated:` to get animation), but you are more likely to let the table view manage editability. Table view editability is controlled through the table view's `editing`

property, usually by sending the table the `setEditing:animated:` message. The table is then responsible for putting its cells into edit mode.

A cell in edit mode can also be selected by the user if the table view's `allowsSelectionDuringEditing` or `allowsMultipleSelectionDuringEditing` is `true`. But this would be unusual.

Putting the table into edit mode is usually left up to the user. A typical interface would be an Edit button that the user can tap. In a navigation interface, we might have our view controller supply the button as a bar button item in the navigation bar:

```
let b = UIBarButtonItem(
    barButtonSystemItem: .Edit, target: self, action: "doEdit:")
self.navigationItem.rightBarButtonItem = b
```

Our action handler will be responsible for putting the table into edit mode, so in its simplest form it might look like this:

```
func doEdit(sender:AnyObject?) {
    self.tableView.setEditing(true, animated:true)
}
```

But that does not solve the problem of getting *out* of edit mode. The standard solution is to have the Edit button replace itself by a Done button:

```
func doEdit(sender:AnyObject?) {
    var which : UIBarButtonSystemItem
    if !self.tableView.editing {
        self.tableView.setEditing(true, animated:true)
        which = .Done
    } else {
        self.tableView.setEditing(false, animated:true)
        which = .Edit
    }
    let b = UIBarButtonItem(
        barButtonSystemItem: which, target: self, action: "doEdit:")
    self.navigationItem.rightBarButtonItem = b
}
```

However, it turns out that all of that is completely unnecessary! If we want standard behavior, it's already implemented for us. A UIViewController's `editButtonItem` method vends a bar button item that calls the UIViewController's `setEditing:animated:` when tapped, tracks whether we're in edit mode with the UIViewController's `editing` property, and changes its own title accordingly (Edit or Done). Moreover, a UITableViewController's implementation of `setEditing:animated:` is to call `setEditing:animated:` on its table view. Thus, if we're using a UITableViewController, we get all of that behavior for free, just by calling `editButtonItem` and inserting the resulting button into our interface:

```
self.navigationItem.rightBarButtonItem = self.editButtonItem()
```

When the table view enters edit mode, it consults its data source and delegate about the editability of individual rows:

`tableView:canEditRowAtIndexPath:` *to the data source*
: The default is `true`. The data source can return `false` to prevent the given row from entering edit mode.

`tableView:editingStyleForRowAtIndexPath:` *to the delegate*
: Each standard editing style corresponds to a control that will appear in the cell. The choices (UITableViewCellEditingStyle) are:

  `.Delete`
  : The cell shows a Minus button at its left end. The user can tap this to summon a Delete button, which the user can then tap to confirm the deletion. This is the default.

  `.Insert`
  : The cell shows a Plus button at its left end; this is usually taken to be an insert button.

  `.None`
  : No editing control appears.

If the user taps an insert button (the Plus button) or a delete button (the Delete button that appears after the user taps the Minus button), the data source is sent the `tableView:commitEditingStyle:forRowAtIndexPath:` message and is responsible for obeying it. In your response, you will probably want to alter the structure of the table, and UITableView methods for doing this are provided:

- `insertRowsAtIndexPaths:withRowAnimation:`
- `deleteRowsAtIndexPaths:withRowAnimation:`
- `insertSections:withRowAnimation:`
- `deleteSections:withRowAnimation:`
- `moveSection:toSection:`
- `moveRowAtIndexPath:toIndexPath:`

The row animations here are effectively the same ones discussed earlier in connection with refreshing table data; `.Left` for an insertion means to slide in from the left, and for a deletion it means to slide out to the left, and so on. The two "move" methods provide animation with no provision for customizing it.

If you're issuing more than one of these commands, you can combine them by surrounding them with `beginUpdates` and `endUpdates`, forming an *updates block*. An updates block combines not just the animations but the requested changes themselves.

This relieves you from having to worry about how a command is affected by earlier commands in the same updates block; indeed, the order of commands within an updates block doesn't really matter.

For example, if you delete row 1 of a certain section and then (in a separate command in the same updates block) delete row 2 of the same section, you delete two successive rows, just as you would expect; the notion "2" does not change its meaning because you deleted an earlier row first, because you *didn't* delete an earlier row first — the updates block combines the commands for you, interpreting both index paths with respect to the state of the table before any changes are made. If you perform insertions and deletions together in one updates block, the deletions are performed first, regardless of the order of your commands, and the insertion row and section numbers refer to the state of the table after the deletions.

An updates block can also include `reloadRows...` and `reloadSections...` commands (but not `reloadData`).

I need hardly emphasize once again (but I will anyway) that view is not model. It is one thing to rearrange the appearance of the table, another to alter the underlying data. It is up to you to make certain you do both together. Do not, even for a moment, permit the data and the view to get out of synch with each other! If you delete a row, you must first remove from the model the datum that it represents. The runtime will try to help you with error messages if you forget to do this, but in the end the responsibility is yours. I'll give examples as we proceed.

Deleting Cells

Deletion of cells is the default, so there's not much for us to do in order to implement it. If our view controller is a UITableViewController and we've displayed the Edit button in a navigation bar, everything happens automatically: when the user taps the Edit button, the view controller's `setEditing:animated:` is called, the table view's `setEditing:animated:` is called, and the cells all show the Minus button at the left end. The user can then tap a Minus button; a Delete button is shown at the cell's right end. You can customize the Delete button's title with the table view delegate method `tableView:titleForDeleteConfirmationButtonForRowAtIndexPath:`.

What is *not* automatic is the actual response to the Delete button. For that, we need to implement `tableView:commitEditingStyle:forRowAtIndexPath:`. Typically, you'll remove the corresponding entry from the underlying model data, and you'll call `deleteRowsAtIndexPaths:withRowAnimation:` or `deleteSections:withRowAnimation:` to update the appearance of the table.

To illustrate, let's suppose once again that the underlying model is a pair of parallel arrays of strings (`self.sectionNames`) and arrays (`self.sectionData`). Our approach will be in two stages:

1. Deal with the model data. We'll delete the datum for the requested row; if this empties the section array, we'll also delete that section array and the corresponding section name.
2. Deal with the table's appearance. If we deleted the section array, we'll call `deleteSections:withRowAnimation:` (and reload the section index if there is one); otherwise, we'll call `deleteRowsAtIndexPaths:withRowAnimation:`.

That's the strategy; here's the implementation:

```
override func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath ip: NSIndexPath) {
        switch editingStyle {
        case .Delete:
            if self.sectionData[ip.section].count == 0 {
                self.sectionData.removeAtIndex(ip.section)
                self.sectionNames.removeAtIndex(ip.section)
                tableView.deleteSections(NSIndexSet(index: ip.section),
                    withRowAnimation:.Automatic)
                tableView.reloadSectionIndexTitles()
            } else {
                tableView.deleteRowsAtIndexPaths([ip],
                    withRowAnimation:.Automatic)
            }
        default: break
        }
}
```

The user can also delete a row by sliding it to the left to show its Delete button *without* having explicitly entered edit mode; no other row is editable, and no other editing controls are shown. This feature is implemented "for free" by virtue of our having supplied an implementation of `tableView:commitEditingStyle:forRowAtIndexPath:`.

If you're like me, your first response will be: "Thanks for the free functionality, Apple, and now how do I turn this off?" Because the Edit button is already using the UIViewController's `editing` property to track edit mode, we can take advantage of this and refuse to let any cells be edited unless the view controller *is* in edit mode:

```
override func tableView(tableView: UITableView,
    editingStyleForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCellEditingStyle {
        return self.editing ? .Delete : .None
}
```

Custom Action Buttons

The table cell is itself inside a little horizontal scroll view; the user who slides a cell to the left is actually scrolling the cell to the left, revealing the Delete button behind it. You

can customize what buttons will appear when the user slides a cell leftward to enter edit mode, or enters edit mode and taps the Minus button.

To configure the buttons for a row of the table, implement the table view delegate method `tableView:editActionsForRowAtIndexPath:` and return an array of UITableViewRowAction objects in right-to-left order (or `nil` to get the default Delete button). Create a row action button with its initializer, `init(style:title:handler:)`. The parameters are:

`style:`
: A UITableViewRowActionStyle, either `.Default` or `.Normal`. By default, `.Default` is a red button signalling a destructive action, like the Delete button, while `.Normal` is a gray button. You can subsequently change the color by setting the button's `backgroundColor`.

`title:`
: The text of the button.

`handler:`
: A function to be called when the button is tapped; it takes two parameters, a reference to the row action and the index path for this cell.

If you want the user to be able to slide the cell to reveal the buttons, you must implement `tableView:commitEditingStyle:forRowAtIndexPath:`, even if your implementation is empty. But even if you *don't* implement this method, the buttons can be revealed by putting the table view into edit mode and tapping the Minus button. Your `handler:` can call `tableView:commitEditingStyle:forRowAtIndexPath:` if appropriate; a custom Delete button, for example, might do so.

In this example, we give our cells a blue Mark button in addition to the default Delete button:

```
override func tableView(tableView: UITableView,
    editActionsForRowAtIndexPath indexPath: NSIndexPath)
    -> [UITableViewRowAction]? {
        let act = UITableViewRowAction(style: .Normal, title: "Mark") {
            action, ip in
            print("Mark") // in real life, do something here
        }
        act.backgroundColor = UIColor.blueColor()
        let act2 = UITableViewRowAction(style: .Default, title: "Delete") {
            action, ip in
            self.tableView(self.tableView, commitEditingStyle:.Delete,
                forRowAtIndexPath:ip)
        }
        return [act2, act]
}
```

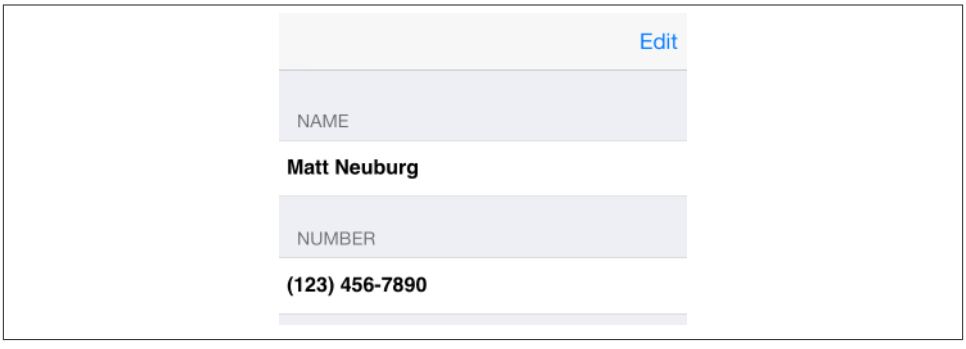

Figure 8-10. A simple phone directory app

Configuration of these buttons is disappointingly inflexible — for example, you can't achieve anything like the Mail app's interface — and many developers will prefer to continue rolling their own sliding table cells, as in the past.

Editable Content in Cells

A cell might have content that the user can edit directly, such as a UITextField (Chapter 10). Because the user is working in the view, you need a way to reflect the user's changes into the model. This will probably involve putting yourself in contact with the interface objects where the user does the editing.

To illustrate, I'll implement a table view cell with a text field that is editable when the cell is in edit mode. Imagine an app that maintains a list of names and phone numbers. A name and phone number are displayed as a grouped style table, and they become editable when the user taps the Edit button (Figure 8-10).

We don't need a button at the left end of the cell when it's being edited:

```
override func tableView(tableView: UITableView,
    editingStyleForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCellEditingStyle {
        return .None
}
```

A UITextField is editable if its `enabled` is `true`. To tie this to the cell's `editing` state, it is probably simplest to implement a custom UITableViewCell class. I'll call it MyCell, and I'll design it in the nib, giving it a single UITextField that's pointed to through an outlet property called `textField`. In the code for MyCell, we override `didTransitionToState:`, as follows:

```
class MyCell : UITableViewCell {
    @IBOutlet weak var textField : UITextField!
    override func didTransitionToState(state: UITableViewCellStateMask) {
        self.textField.enabled = state.contains(.ShowingEditControlMask)
        super.didTransitionToState(state)
    }
}
```

In the table view's data source, we make ourselves the text field's delegate when we create and configure the cell:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier(
            "Cell", forIndexPath:indexPath) as! MyCell
            switch indexPath.section {
        case 0:
            cell.textField.text = self.name
        case 1:
            cell.textField.text = self.numbers[indexPath.row]
            cell.textField.keyboardType = .NumbersAndPunctuation
        default: break
        }
        cell.textField.delegate = self
        return cell
}
```

We are the UITextField's delegate, so we are responsible for implementing the Return button in the keyboard to dismiss the keyboard (I'll talk more about this in Chapter 10):

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    textField.endEditing(true)
    return false
}
```

When a text field stops editing, we are its delegate, so we can hear about it in `textFieldDidEndEditing:`. We work out which cell this text field belongs to — I like to do this by simply walking up the view hierarchy until I come to a table view cell — and update the model accordingly:

```
func textFieldDidEndEditing(textField: UITextField) {
    // some cell's text field has finished editing; which cell?
    var v : UIView = textField
    repeat { v = v.superview! } while !(v is UITableViewCell)
    let cell = v as! MyCell
    // update data model to match
    let ip = self.tableView.indexPathForCell(cell)!
    if ip.section == 1 {
        self.numbers[ip.row] = cell.textField.text!
    } else if ip.section == 0 {
        self.name = cell.textField.text!
    }
}
```

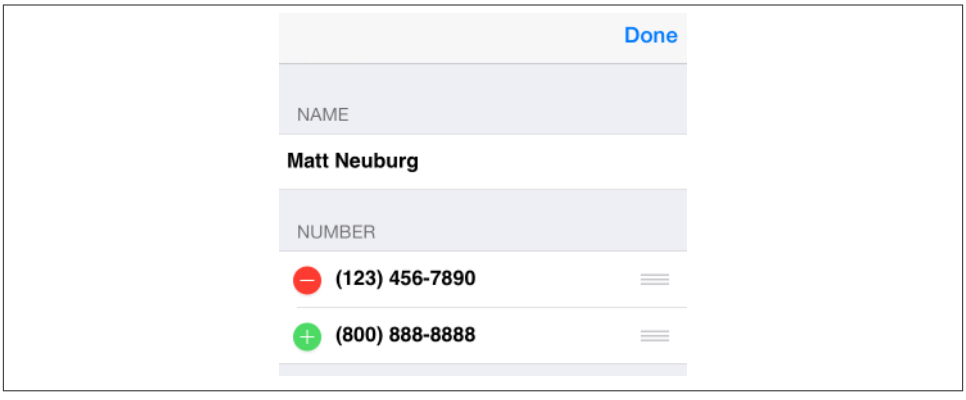

Figure 8-11. Phone directory app in edit mode

Inserting Cells

You are unlikely to attach a Plus (insert) button to every row. A more likely interface is that when a table is edited, every row has a Minus button except the last row, which has a Plus button; this shows the user that a new row can be appended at the end of the list.

Let's implement this for phone numbers in our name-and-phone-number app, allowing the user to give a person any quantity of phone numbers (Figure 8-11):

```
override func tableView(tableView: UITableView,
    editingStyleForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCellEditingStyle {
        if indexPath.section == 1 {
            let ct = self.tableView(
                tableView, numberOfRowsInSection:indexPath.section)
            if ct-1 == indexPath.row {
                return .Insert
            }
            return .Delete;
        }
        return .None
}
```

The person's name has no editing control (a person must have exactly one name), so we prevent it from indenting in edit mode:

```
override func tableView(tableView: UITableView,
    shouldIndentWhileEditingRowAtIndexPath indexPath: NSIndexPath)
    -> Bool {
        if indexPath.section == 1 {
            return true
        }
        return false
}
```

When the user taps an editing control, we must respond. We immediately force our text fields to cease editing: the user have may tapped the editing control while editing, and we want our model to contain the very latest changes, so this is effectively a way of causing our `textFieldDidEndEditing:` to be called. The model for our phone numbers is an array of strings (`self.numbers`). We already know what to do when the tapped control is a delete button; things are similar when it's an insert button, but we've a little more work to do. The new row will be empty, and it will be at the end of the table; so we append an empty string to the `self.numbers` model array, and then we insert a corresponding row at the end of the table view. But now two successive rows have a Plus button; the way to fix that is to reload the first of those rows. Finally, we also show the keyboard for the new, empty phone number, so that the user can start editing it immediately; we do that outside the updates block:

```
override func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath) {
        tableView.endEditing(true)
        if editingStyle == .Insert {
            self.numbers += [""]
            let ct = self.numbers.count
            tableView.beginUpdates()
            tableView.insertRowsAtIndexPaths(
                [NSIndexPath(forRow:ct-1, inSection:1)],
                withRowAnimation:.Automatic)
            tableView.reloadRowsAtIndexPaths(
                [NSIndexPath(forRow:ct-2, inSection:1)],
                withRowAnimation:.Automatic)
            tableView.endUpdates()
            // crucial that this next bit be *outside* the update block
            let cell = self.tableView.cellForRowAtIndexPath(
                NSIndexPath(forRow:ct-1, inSection:1))
            (cell as! MyCell).textField.becomeFirstResponder()
        }
        if editingStyle == .Delete {
            self.numbers.removeAtIndex(indexPath.row)
            tableView.beginUpdates()
            tableView.deleteRowsAtIndexPaths(
                [indexPath], withRowAnimation:.Automatic)
            tableView.reloadSections(
                NSIndexSet(index:1), withRowAnimation:.Automatic)
            tableView.endUpdates()
        }
}
```

Rearranging Cells

If the data source implements `tableView:moveRowAtIndexPath:toIndexPath:`, the table displays a reordering control at the right end of each row in edit mode (Figure 8-11), and the user can drag it to rearrange cells. The reordering control can be

suppressed for individual cells by implementing `tableView:canMoveRowAtIndexPath:`. The user is free to move rows that display a reordering control, but the delegate can limit where a row can be moved to by implementing `tableView:targetIndexPathForMoveFromRowAtIndexPath:toProposedIndexPath:`.

To illustrate, we'll add to our name-and-phone-number app the ability to rearrange phone numbers. There must be multiple phone numbers to rearrange:

```
override func tableView(tableView: UITableView,
    canMoveRowAtIndexPath indexPath: NSIndexPath) -> Bool {
        if indexPath.section == 1 && self.numbers.count > 1 {
            return true
        }
        return false
}
```

A phone number must not be moved out of its section, so we implement the delegate method to prevent this. We also take this opportunity to dismiss the keyboard if it is showing:

```
override func tableView(tableView: UITableView,
    targetIndexPathForMoveFromRowAtIndexPath sourceIndexPath: NSIndexPath,
    toProposedIndexPath proposedDestinationIndexPath: NSIndexPath)
    -> NSIndexPath {
        tableView.endEditing(true)
        if proposedDestinationIndexPath.section == 0 {
            return NSIndexPath(forRow:0, inSection:1)
        }
        return proposedDestinationIndexPath
}
```

After the user moves an item, `tableView:moveRowAtIndexPath:toIndexPath:` is called, and we trivially update the model to match. We also reload the table, to fix the editing controls:

```
override func tableView(tableView: UITableView,
    moveRowAtIndexPath fromIndexPath: NSIndexPath,
    toIndexPath: NSIndexPath) {
        let s = self.numbers[fromIndexPath.row]
        self.numbers.removeAtIndex(fromIndexPath.row)
        self.numbers.insert(s, atIndex: toIndexPath.row)
        tableView.reloadData()
}
```

Dynamic Cells

A table may be rearranged not just in response to the user working in edit mode, but for some other reason entirely. In this way, many interesting and original interfaces are possible.

In this example, we permit the user to double tap on a section header as a way of collapsing or expanding the section — that is, we'll suppress or permit the display of the rows of the section, with a nice animation as the change takes place. (This idea is shamelessly stolen from a WWDC 2010 video.)

One more time, our data model consists of the two arrays, `self.sectionNames` and `self.sectionData`. I've also got a Set (of Int), `self.hiddenSections`, in which I'll list the sections that aren't displaying their rows. That list is all I'll need, since either a section is showing all its rows or it's showing none of them:

```
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
        if self.hiddenSections.contains(section) {
            return 0
        }
        return self.sectionData[section].count
}
```

We need a correspondence between a section header and the number of its section. It's odd that UITableView doesn't give us such a correspondence; it provides `indexPathForCell:`, but there is no `sectionForHeaderFooterView:`. My solution is to subclass UITableViewHeaderFooterView and give my subclass a public property `section`:

```
class MyHeaderView : UITableViewHeaderFooterView {
    var section = 0
}
```

Whenever `tableView:viewForHeaderInSection:` is called, I set the header view's `section` property:

```
override func tableView(tableView: UITableView,
    viewForHeaderInSection section: Int) -> UIView? {
        let h = tableView.dequeueReusableHeaderFooterViewWithIdentifier(
            "Header") as! MyHeaderView
        // ...
        h.section = section // *
        return h
}
```

The section headers are a UITableViewHeaderFooterView subclass with `userInteractionEnabled` set to `true` and a UITapGestureRecognizer attached, so we can detect a double tap. When the user double taps a section header, we learn from the header what section this is, we find out from the model how many rows this section has, and we derive the index paths of the rows we're about to insert or remove. Now we look for the section number in our `hiddenSections` set. If it's there, we're about to display the rows, so we *remove* that section number from `hiddenSections`, and we *insert* the rows. If it's *not* there, we're about to hide the rows, so we *insert* that section number into `hiddenSections`, and we *delete* the rows:

```
    func tap (g : UIGestureRecognizer) {
        let v = g.view as! MyHeaderView
        let sec = v.section
        let ct = self.sectionData[sec].count
        let arr = Array(0..<ct).map {NSIndexPath(forRow:$0, inSection:sec)}
        if self.hiddenSections.contains(sec) {
            self.hiddenSections.remove(sec)
            self.tableView.beginUpdates()
            self.tableView.insertRowsAtIndexPaths(arr,
                withRowAnimation:.Automatic)
            self.tableView.endUpdates()
            self.tableView.scrollToRowAtIndexPath(arr[ct-1],
                atScrollPosition:.None,
                animated:true)
        } else {
            self.hiddenSections.insert(sec)
            self.tableView.beginUpdates()
            self.tableView.deleteRowsAtIndexPaths(arr,
                withRowAnimation:.Automatic)
            self.tableView.endUpdates()
        }
    }
```

Table View Menus

A menu, in iOS, is a sort of balloon containing tappable words such as Copy, Cut, and Paste. You can permit the user to display a menu from a table view cell by performing a long press on the cell. The long press followed by display of the menu gives the cell a selected appearance, which goes away when the menu is dismissed.

To allow the user to display a menu from a table view's cells, you implement three delegate methods:

`tableView:shouldShowMenuForRowAtIndexPath:`
: Return `true` if the user is to be permitted to summon a menu by performing a long press on this cell.

`tableView:canPerformAction:forRowAtIndexPath:withSender:`
: You'll be called repeatedly with selectors for various actions that the system knows about. Returning `true`, regardless, causes the Copy, Cut, and Paste menu items to appear in the menu, corresponding to the `copy:`, `cut:`, and `paste:` actions; return `false` to prevent the menu item for an action from appearing. The menu itself will then appear unless you return `false` to all three actions. The sender is the shared UIMenuController.

`tableView:performAction:forRowAtIndexPath:withSender:`
: The user has tapped one of the menu items; your job is to respond to it somehow.

Here's an example where the user can summon a Copy menu from any cell (Figure 8-12):

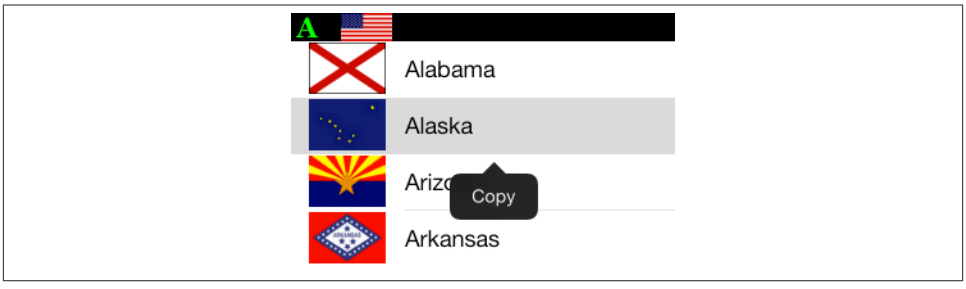

Figure 8-12. A table view cell with a menu

```
override func tableView(tableView: UITableView,
    shouldShowMenuForRowAtIndexPath indexPath: NSIndexPath) -> Bool {
        return true
}
override func tableView(tableView: UITableView,
    canPerformAction action: Selector,
    forRowAtIndexPath indexPath: NSIndexPath,
    withSender sender: AnyObject?) -> Bool {
        return action == "copy:"
}
override func tableView(tableView: UITableView,
    performAction action: Selector,
    forRowAtIndexPath indexPath: NSIndexPath,
    withSender sender: AnyObject?) {
        if action == "copy:" {
            // ... do whatever copying consists of ...
        }
}
```

To add a custom menu item to the menu (other than `copy:`, `cut:`, and `paste:`) is a little more work. First, you must tell the shared UIMenuController to append the menu item to the global menu; the `tableView:shouldShowMenuForRowAtIndexPath:` delegate method is a good place to do this:

```
override func tableView(tableView: UITableView,
    shouldShowMenuForRowAtIndexPath indexPath: NSIndexPath) -> Bool {
        let mi = UIMenuItem(title: "Abbrev", action: "abbrev:")
        UIMenuController.sharedMenuController().menuItems = [mi]
        return true
}
```

We have now given the menu an additional menu item whose title is Abbrev, and whose action when the menu item is tapped is `abbrev:`. (I am imagining here a table of the names of U.S. states, where one can copy a state's two-letter abbreviation to the clipboard.) If we want this menu item to appear in the menu, and if we want to respond to it when the user taps it, we must add that selector to the two `performAction:` delegate methods:

```
override func tableView(tableView: UITableView,
    canPerformAction action: Selector,
    forRowAtIndexPath indexPath: NSIndexPath,
    withSender sender: AnyObject) -> Bool {
        return action == "copy:" || action == "abbrev:"
}
override func tableView(tableView: UITableView,
    performAction action: Selector,
    forRowAtIndexPath indexPath: NSIndexPath,
    withSender sender: AnyObject) {
        if action == "copy:" {
            // ... do whatever copying consists of ...
        }
        if action == "abbrev:" {
            // ... do whatever abbreviating consists of ...
        }
    }
```

Now comes the tricky part: we must implement our custom selector, `abbrev:`, *in the cell*. We will therefore need our table to use a custom UITableViewCell subclass. Let's call it MyCell:

```
class MyCell : UITableViewCell {
    func abbrev(sender:AnyObject!) {
        // ...
    }
}
```

The Abbrev menu item now appears when the user long-presses a cell of our table, and the cell's `abbrev:` method is called when the user taps that menu item. We could respond directly to the tap in the cell, but it seems more consistent that our table view delegate should respond. So we work out what table view this cell belongs to and send its delegate the very message it is already expecting:

```
func abbrev(sender:AnyObject!) {
    // find my table view
    var v : UIView = self
    repeat {v = v.superview!} while !(v is UITableView)
    let tv = v as! UITableView
    // ask it what index path we are
    let ip = tv.indexPathForCell(self)!
    // talk to its delegate
    let action = Selector(__FUNCTION__ + ":") // *
    tv.delegate?.tableView?(
        tv, performAction:action, forRowAtIndexPath:ip, withSender:sender)
}
```

The starred line calls attention to the fact that Swift's `__FUNCTION__` literal does not evaluate to a valid Objective-C selector string; I regard this as a bug.