An app **Homepwner** keeps an inventory of all possessions.

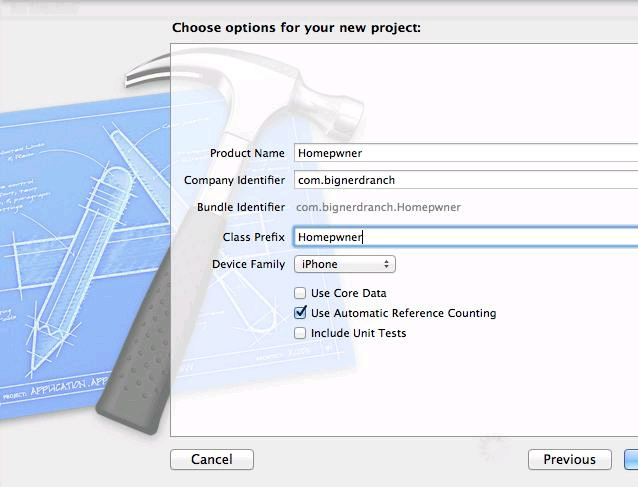
**Chapter 9 [Phase 1]: UITableView and UITableViewController**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

A **UITableView** displays a single column of data with a variable number of rows.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Step 1:** Create a new iOS Empty Application project [Product Name: Homepwner, Class Prefix: Homepwner, Device: iPhone].

UITableViewController

UITableView is a view object, so, according to Model-View- Controller, it knows how to draw itself, but that’s it. It doesn’t handle application logic or data. Thus, when using a UITableView, you must consider what else is necessary to get the table working in your application

A UITableView typically needs a view controller to handle its appearance on the screen.

A UITableView needs a data source. A UITableView asks its data source for the number of rows to display, the data to be shown in those rows, and other tidbits that make a UITableView a useful user interface. Without a data source, a table view is just an empty container. The dataSource for a UITableView can be any type of Objective-C object as long as it conforms to the UITableViewDataSource protocol.

A UITableView typically needs a delegate that can inform other objects of events involving the UITableView. The delegate can be any object as long as (you guessed it!) it conforms to the UITableViewDelegate protocol.

An instance of the class UITableViewController can fill all three roles: view controller, data source, and delegate.

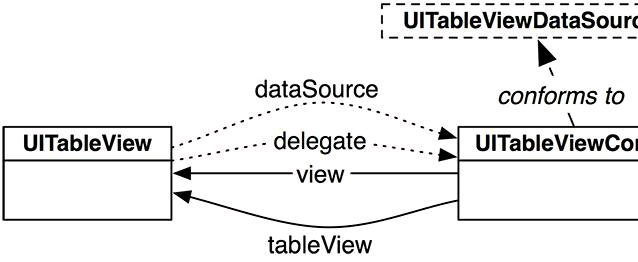
UITableViewController is a subclass of UIViewController, so a UITableViewController has a view. A UITableViewController’s view is always an instance of UITableView, and the UITableViewController handles the preparation and presentation of the UITableView. When a UITableViewController creates its view, the dataSource and delegate instance variables of the UITableView are automatically set to point at the UITableViewController (Figure 9.4).

Figure 9.4 UITableViewController-UITableView

Figure 9.4 UITableViewController-UITableView

relationship



Subclassing UITableViewController

Now you’re going to write a subclass of UITableViewController for Homepwner. For this view controller, we’ll use the NSObject template. From the File menu, select New and then New File.... From the iOS section, select Cocoa Touch, choose Objective-C class, and hit Next. Then, select NSObject from the pop-up menu and enter ItemsViewController as the name of the new class. Click Next and then click Create on the next sheet to save your class.

Open ItemsViewController.h and change its superclass: @interface ItemsViewController : NSObject @interface ItemsViewController : UITableViewController

The designated initializer of UITableViewController is initWithStyle:, which takes a constant that determines the style of the table view. There are two options: UITableViewStylePlain, where each row is a rectangle, and UITableViewStyleGrouped, where the top and bottom rows have rounded corners. In ItemsViewController.m, implement the following initializers.

#import "ItemsViewController.h"

@implementation ItemsViewController

- (id)init {



// Call the superclass's designated initializer

// Call the superclass's designated initializer self = [super initWithStyle:UITableViewStyleGrouped]; if (self) {

}

return self; }

- (id)initWithStyle:(UITableViewStyle)style {

return [self init]; }

This will ensure that all instances of ItemsViewController use the UITableViewStyleGrouped style, no matter what initialization message is sent to it.

Open HomepwnerAppDelegate.m. In application:didFinishLaunchingWithOptions:, create an instance of ItemsViewController and set it as the rootViewController of the window. Make sure to import the header file for ItemsViewController at the top of this file. #import "ItemsViewController.h" @implementation HomepwnerAppDelegate

- (BOOL)application:(UIApplication \*)application didFinishLaunchingWithOptions:(NSDictionary \*)launchOptions

{ self.window = [[UIWindow alloc] initWithFrame:[[UIScreen

mainScreen] bounds]]; // Override point for customization after application launch.

// Create a ItemsViewController

ItemsViewController \*itemsViewController = [[ItemsViewController alloc] init];

// Place ItemsViewController's table view in the window hierarchy

[[self window] setRootViewController:itemsViewController];

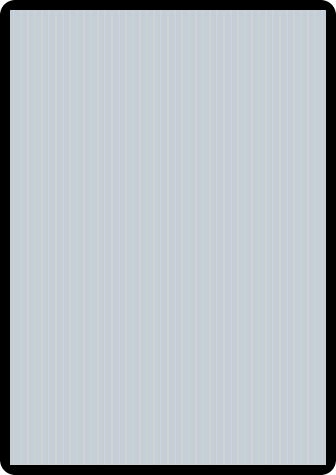
self.window.backgroundColor = [UIColor whiteColor]; [self.window makeKeyAndVisible]; return YES;

}

Build and run your application. You will see the default appearance of a plain UITableView with no content, as shown in Figure 9.5. How did you get a table view? As a subclass of UIViewController, a UITableViewController inherits the view method. This method calls loadView, which creates and loads an empty view object if none exists. A UITableViewController’s view is always an instance of UITableView, so sending view to the UITableViewController gets you a bright, shiny, and empty table view.



Figure 9.5 Empty UITableView



An empty table view is a sad table view. You should give it some rows to display. Remember the BNRItem class you wrote in Chapter 3? Now you’re going to use that class again: each row of the table view will display an instance of BNRItem. Locate the header and implementation files for BNRItem (BNRItem.h and BNRItem.m) in Finder and drag them onto Homepwner’s project navigator. Make sure you use the files from Chapter 3, not the unfinished files from Chapter 2.

When dragging these files onto your project window, select the checkbox labeled Copy items into destination group’s folder when prompted. This will copy the files from their current directory to your project’s directory on the filesystem and add them to your project.

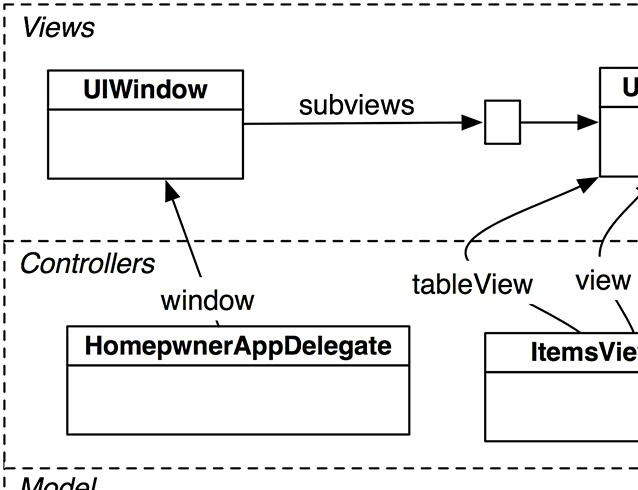
  

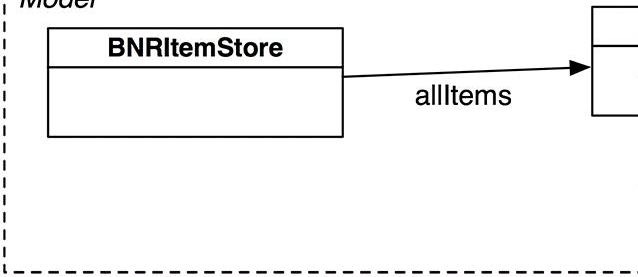
UITableView’s Data Source

The process of providing a UITableView with rows in Cocoa Touch is different from the typical procedural programming task. In a procedural design, you tell the table view what it should display. In Cocoa Touch, the table view asks another object – its dataSource – what it should display. In our case, the ItemsViewController is the data source, so it needs a way to store item data.

In Chapter 2, you used an NSMutableArray to store BNRItem instances. You will do the same thing in this chapter, but with a little twist. The NSMutableArray that holds the BNRItem instances will be abstracted into another object – a BNRItemStore (Figure 9.6).

Figure 9.6 Homepwner object diagram



If an object wants to see all of the items, it will ask the BNRItemStore for the array that contains them. In future chapters, you’ll make the store responsible for performing operations on the array, like reordering, adding, and removing BNRItems. It will also be responsible for saving and loading the BNRItems from disk.

Creating BNRItemStore

From the File menu, select New and then New File.... Create a new NSObject subclass and name it BNRItemStore.

BNRItemStore will be a singleton. This means there will only be one instance of this type in the application; if you try to create another instance, the class will quietly return the existing instance instead. A singleton is useful when you have an object that many objects will talk to. Those objects can ask the singleton class for its one instance, which is better than passing that instance as an argument to every method that will use it.

To get the (single instance of) BNRItemStore, you will send the BNRItemStore class the message sharedStore. Declare this class method in BNRItemStore.h. #import <Foundation/Foundation.h>

@interface BNRItemStore : NSObject {

}

// Notice that this is a class method and prefixed with a + instead of a -

instead of a -

+ (BNRItemStore \*)sharedStore;

@end

When this message is sent to the BNRItemStore class, the class will check to see if the single instance of BNRItemStore has already been created. If it has, the class will return the instance. If not, it will create the instance and return it. In BNRItemStore.m, implement sharedStore.

+ (BNRItemStore \*)sharedStore {

static BNRItemStore \*sharedStore = nil; if (!sharedStore)

sharedStore = [[super allocWithZone:nil] init];

return sharedStore; }

Notice that the variable sharedStore is declared as static. Unlike a local variable, a static variable does not live on the stack and is not destroyed when the method returns. Instead, a static variable is only declared once (when the application is loaded into memory), and it is never destroyed. A static variable is like a local variable in that you can only access this variable in the method in which it is declared. Therefore, no other object or method can use the BNRItemStore pointed to by this variable except via the sharedStore method.

The initial value of sharedStore is nil. The first time this method runs, an instance of BNRItemStore will be created, and sharedStore will be set to point to it. In subsequent calls to this method, sharedStore will still point at that instance of BNRItemStore. This variable has a strong reference to the BNRItemStore and, since this variable will never be destroyed, the object it points to will never be destroyed either.

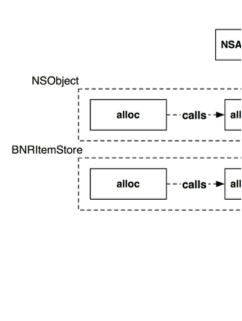
To enforce the singleton status of BNRItemStore, you must ensure that another instance of BNRItemStore cannot be allocated. One approach would be to override alloc in BNRItemStore so that it does not create a new instance but returns the existing instance instead.

However, there is a problem with this approach: alloc is a dummy method. It just calls allocWithZone:, which then calls the C function NSAllocateObject, which does the actual memory allocation (Figure 9.7).



Figure 9.7 Default allocation chain

Figure 9.7 Default allocation chain



Thus, a knowledgeable programmer could still create an instance of BNRItemStore via allocWithZone:, which would bypass our sneaky alloc trap. To prevent this possibility, override allocWithZone: in BNRItemStore.m to return the single BNRItemStore instance.

+ (id)allocWithZone:(NSZone \*)zone {

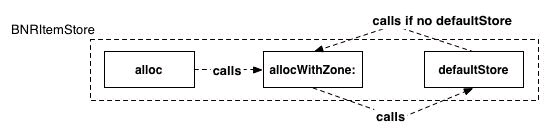
return [self sharedStore]; }

Now if sharedStore were to send alloc or allocWithZone: to BNRItemStore, then the method would call BNRItemStore’s implementation of allocWithZone:. That implementation just calls sharedStore, which would then

call BNRItemStore’s allocWithZone: again, which would

call BNRItemStore’s allocWithZone: again, which would then call sharedStore, which would... well, you get the picture.

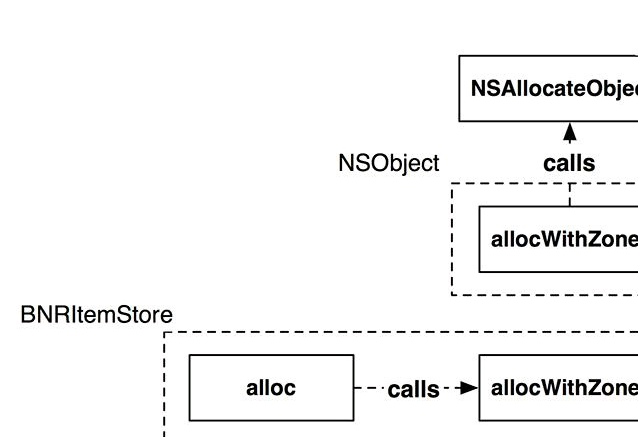
Figure 9.8 Not sending allocWithZone: to NSObject causes loop



This is why we had sharedStore call NSObject’s implementation of allocWithZone:. sharedStore = [[super allocWithZone:nil] init];

By sending allocWithZone: to super, we skip over our trap and get an instance of BNRItemStore when we need it (Figure 9.9).

Figure 9.9 BNRItemStore and NSObject allocation methods



We can only skip over our alloc trap within the implementation of BNRItemStore because the super keyword is only relevant to the class in which the method is implemented.

Now we have ensured that multiple instances of BNRItemStore cannot be created. We have also ensured that once the instance of BNRItemStore is created, it is never destroyed because a static variable (that never gets destroyed) always maintains ownership of it.

In BNRItemStore.h, give BNRItemStore an instance variable to hold an array of BNRItem instances and declare two more methods: #import <Foundation/Foundation.h>

@class BNRItem;

@interface BNRItemStore : NSObject {

NSMutableArray \*allItems;

} + (BNRItemStore \*)sharedStore;

- (NSArray \*)allItems; - (BNRItem \*)createItem;

@end

See the @class directive? That tells the compiler that there is a BNRItem class and that it doesn’t need to know this class’s details in the current file. This allows us to use the BNRItem symbol in the declaration of createItem without importing BNRItem.h. Using the @class directive can speed up compile times considerably because fewer files have to be recompiled when one file changes. (Wonder why? Flip back and read the section called “For the More Curious: Build Phases, Compiler Errors, and Linker Errors”.)

In files that actually send messages to the BNRItem class or

instances of it, you must import the file it was declared in so that

the compiler will have all of its details. At the top of

BNRItemStore.m, import BNRItem.h.

#import "BNRItemStore.h"

#import "BNRItem.h"

In BNRItemStore.m, override init to create an instance of NSMutableArray and assign it to the instance variable. - (id)init

{

{ self = [super init]; if (self) {

allItems = [[NSMutableArray alloc] init]; }

return self; }

Now implement the two methods in BNRItemStore.m.

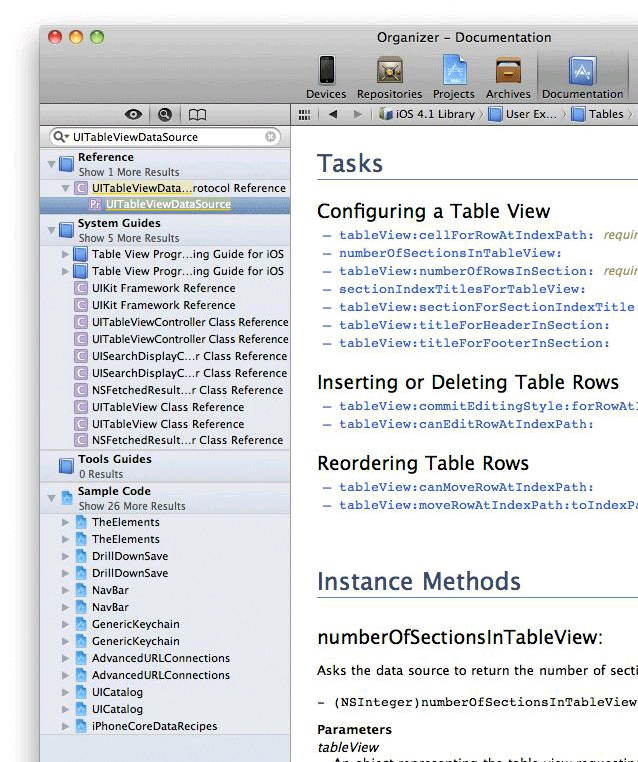
-  (NSArray \*)allItems {  return allItems; }

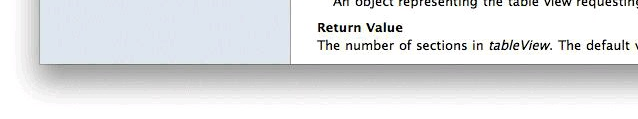
-  (BNRItem \*)createItem {  BNRItem \*p = [BNRItem randomItem];  [allItems addObject:p];  return p; }  Implementing data source methods  In ItemsViewController.m, import BNRItemStore.h and BNRItem.h and update the designated initializer to add 5 random items to the BNRItemStore.  #import "ItemsViewController.h"  #import "BNRItemStore.h" #import "BNRItem.h"  @implementation ItemsViewController  - (id)init {  // Call the superclass's designated initializer self = [super initWithStyle:UITableViewStyleGrouped]; if (self) {  for (int i = 0; i < 5; i++) { [[BNRItemStore sharedStore] createItem];  }  }  return self; }  Now that there are some items in the store, you need to teach ItemsViewController how to turn those items into rows that its UITableView can display. When a UITableView wants to know what to display, it sends messages from the set of messages declared in the UITableViewDataSource protocol.  From the Help menu, choose Documentation and API Reference to open the iOS SDK documentation. Find the

UITableViewDataSource protocol documentation

UITableViewDataSource protocol documentation (Figure 9.10).

Figure 9.10 UITableViewDataSource protocol documentation

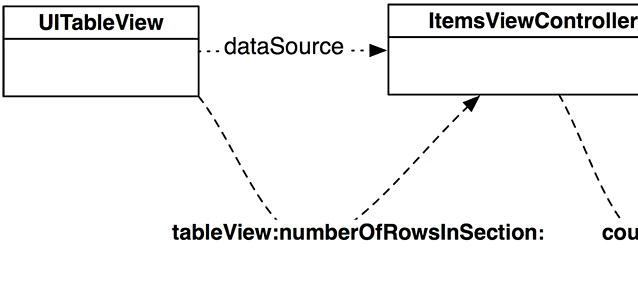
 



There are many methods here, but notice the two marked required method. For ItemsViewController to conform to UITableViewDataSource, it must implement tableView:numberOfRowsInSection: and tableView:cellForRowAtIndexPath:. These methods tell the table view how many rows it should display and what content to display in each row.

Whenever a UITableView needs to display itself, it sends a series of messages (the required methods plus any optional ones that have been implemented) to its dataSource. The required method tableView:numberOfRowsInSection: returns an integer value for the number of rows that the UITableView should display. In the table view for Homepwner, there should be a row for each entry in the store (Figure 9.11).

Figure 9.11 Obtaining the number of rows

In ItemsViewController.m, implement tableView:numberOfRowsInSection:. This method returns the number of rows to display as an NSInteger, which is just another name for int.

- (NSInteger)tableView:(UITableView \*)tableView numberOfRowsInSection:(NSInteger)section

{ return [[[BNRItemStore sharedStore] allItems] count];

}

Wondering about the section that this method refers to? Table views can be broken up into sections, and each section has its own set of rows. For example, in the address book, all names beginning with “D” are grouped together in a section. By default, a table view has one section, and in this chapter, we will work with only one. Once you understand how a table view works, it is not hard to use multiple sections. In fact, using sections is the first challenge at the end of this chapter.

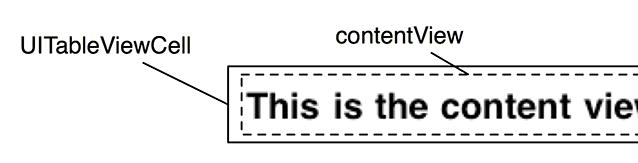
The second required method in the UITableViewDataSource protocol is tableView:cellForRowAtIndexPath:. To implement this method, we’ll need to learn about another class – UITableViewCell.

UITableViewCells

A UITableViewCell is a subclass of UIView, and each row in a UITableView is a UITableViewCell. (Recall that a table in iOS can only have one column, so a row only has one cell.) The UITableViewCells are subviews of the UITableView.

A cell itself has one subview – its contentView (Figure 9.12). The contentView is the superview for the content of the cell. It also can draw an accessory indicator. The accessory indicator shows an action-oriented icon, such as a checkmark, a disclosure icon, or a fancy blue dot with a chevron inside. These icons are accessed through pre-defined constants for the appearance of the accessory indicator. The default is UITableViewCellAccessoryNone, and that’s what we’ll use in this chapter. But you’ll see the accessory indicator again in Chapter 15. (Curious now? See the reference page for UITableViewCell for more details.)

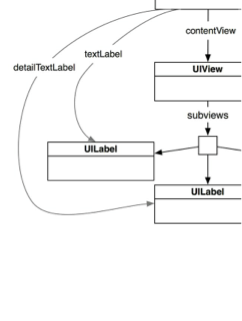
Figure 9.12 UITableViewCell layout

The real meat of a UITableViewCell is the three subviews of the contentView. Two of those subviews are UILabel instances that are properties of UITableViewCell named textLabel and detailTextLabel. The third subview is a UIImageView called imageView (Figure 9.13). In this chapter, we’ll only use textLabel.

Figure 9.13 UITableViewCell hierarchy



Each cell also has a UITableViewCellStyle that determines which subviews are used and their position within the contentView. Examples of these styles and their constants are shown in Figure 9.14.

Figure 9.14 UITableViewCellStyles

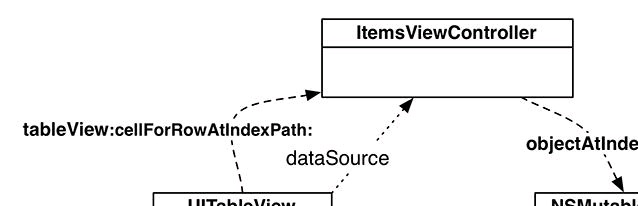
 

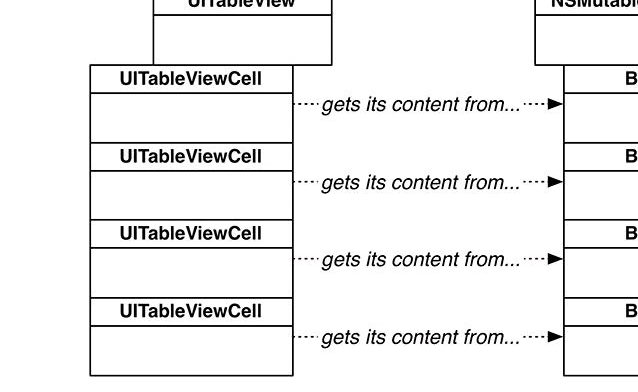


Creating and retrieving UITableViewCells

In this chapter, each cell will display the description of a BNRItem as its textLabel. To make this happen, you need to implement the second required method from the UITableViewDataSource protocol, tableView:cellForRowAtIndexPath:. This method will create a cell, set its textLabel to the description of the corresponding BNRItem, and return it to the UITableView (Figure 9.15).

Figure 9.15 UITableViewCell retrieval



How do you decide which cell a BNRItem corresponds to? One of the parameters sent to tableView:cellForRowAtIndexPath: is an NSIndexPath, which has two properties: section and row. When this message is sent to a data source, the table view is asking, “Can I have a cell to display in section X, row Y?” Because there is only one section in this exercise, the table view only needs to know the row. In ItemsViewController.m, implement tableView:cellForRowAtIndexPath: so that the nth row displays the nth entry in the allItems array.

- (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath

{ // Create an instance of UITableViewCell, with default

appearance UITableViewCell \*cell =

[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault

reuseIdentifier:@"UITableViewCell"];

// Set the text on the cell with the description of the item // that is at the nth index of items, where n = row this cell // will appear in on the tableview BNRItem \*p = [[[BNRItemStore sharedStore] allItems]

objectAtIndex:[indexPath

row]]; [[cell textLabel] setText:[p description]];

return cell; }

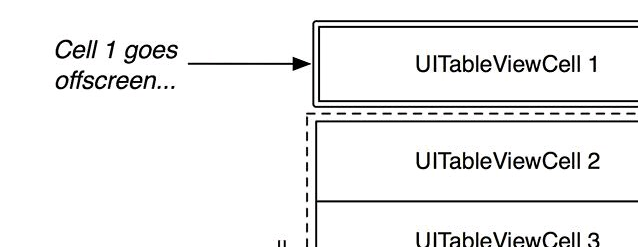
Build and run the application now, and you’ll see a UITableView populated with a list of random BNRItems. Yep, it was that easy. You didn’t have to change anything about BNRItem – you simply changed the controller object and let the controller interface with a different view. This is why Model- View-Controller is such a powerful concept. With a minimal amount of code, you were able to show the same data in an entirely different way.

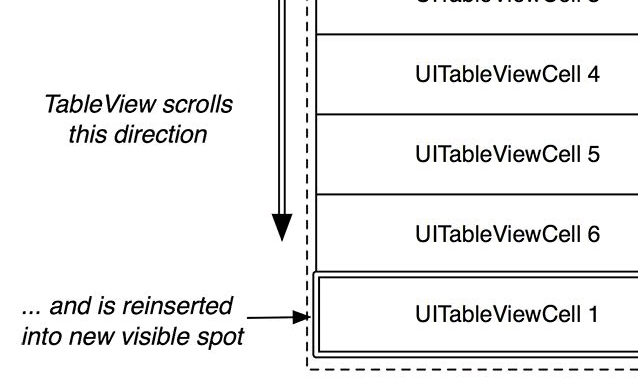
Reusing UITableViewCells

iOS devices have a limited amount of memory. If we were displaying a list with thousands of entries in a UITableView, we would have thousands of instances of UITableViewCell. And your long-suffering iPhone would sputter and die. In its dying breath, it would say “You only needed enough cells to fill the screen... arrrghhh!” It would be right.

To preserve the lives of iOS devices everywhere, you can reuse table view cells. When the user scrolls the table, some cells move offscreen. Offscreen cells are put into a pool of cells available for reuse. Then, instead of creating a brand new cell for every request, the data source first checks the pool. If there is an unused cell, the data source configures it with new data and returns it to the table view.

Figure 9.16 Reusable UITableViewCells





There is one problem: sometimes a UITableView has different types of cells. Occasionally, you have to subclass UITableViewCell to create a special look or behavior. However, different subclasses floating around the pool of reusable cells create the possibility of getting back a cell of the wrong type. You must be sure of the type of the cell returned to you so that you can be sure of what properties and methods it has.

Note that you don’t care about getting any specific cell out of the pool because you’re going to change the cell content anyway. What you need is a cell of a specific type. The good news is every cell has a reuseIdentifier property of type NSString. When a data source asks the table view for a reusable cell, it passes a string and says, “I need a cell with this reuse identifier.” By convention, the reuse identifier is simply the name of the cell class.

In ItemsViewController.m, update

tableView:cellForRowAtIndexPath: to reuse cells:

- (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath

{

UITableViewCell \*cell = [[UITableViewCell alloc]

initWithStyle:UITableViewCellStyleDefault

initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"UITableViewCell"];

// Check for a reusable cell first, use that if it exists UITableViewCell \*cell =

[tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"];

// If there is no reusable cell of this type, create a new one

if (!cell) { cell = [[UITableViewCell alloc]

initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"UITableViewCell"];

}

BNRItem \*p = [[[BNRItemStore sharedStore] allItems] objectAtIndex:[indexPath

row]];

[[cell textLabel] setText:[p description]];

return cell; }

(If you have a table view that uses multiple styles of the same type of cell, you can suffix the reuse identifier with the name of that style, e.g. UITableViewCell-Default.)

Reusing cells means that you only have to create a handful of cells, which puts fewer demands on memory. Your application’s users (and their devices) will thank you. Build and run the application. The behavior of the application should remain the same.

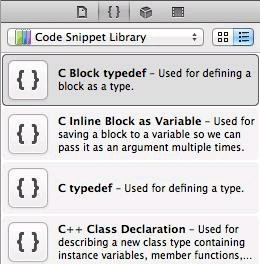


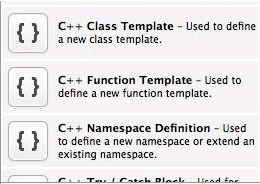
Code Snippet Library

You may have noticed that when you start typing the method definition for init in an implementation file, Xcode will automatically add an init implementation in your source file. If you haven’t noticed this, go ahead and type init in an implementation file and wait for the code-completion to kick in.

The freebie code comes from the code snippet library. You can see the code snippet library by opening the utilities area and selecting the icon in the library selector (Figure 9.17). Alternatively, you can use the shortcut Command-Control- Option-2, which reveals the utilities area and the Code Snippet Library. Substituting another number in the shortcut selects the corresponding library.

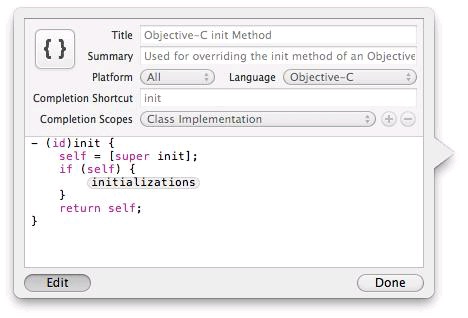
Figure 9.17 Code snippet library



Notice that there are a number of code snippets available (Figure 9.17). Click on one, and in a moment, a window will appear with the details for that snippet. Click the Edit button on the code snippet detail window (Figure 9.18).

Figure 9.18 Snippet editing window

The Completion Shortcut field in the edit window shows you what to type in a source file to have Xcode add the snippet. This window also tells you that this snippet can be used in an Objective-C file as long as you are in the scope of a class implementation.

You can’t edit any of the pre-defined code snippets, but you can create your own. In ItemsViewController.m, locate the implementation of tableView:numberOfRowsInSection:. Highlight the entire method:

- (NSInteger)tableView:(UITableView \*)tableView numberOfRowsInSection:(NSInteger)section

{ return [[[BNRItemStore sharedStore] allItems] count];

}

Drag this highlighted code into the code snippet library. The edit window will appear again, allowing you to fill out the details for this snippet.

One issue with this snippet is that the return statement is really specific to this application – it would be much more useful if the value returned was a code completion placeholder that you could fill in easily. In the edit window, modify the code snippet so it looks like this:

- (NSInteger)tableView:(UITableView \*)tableView numberOfRowsInSection:(NSInteger)section

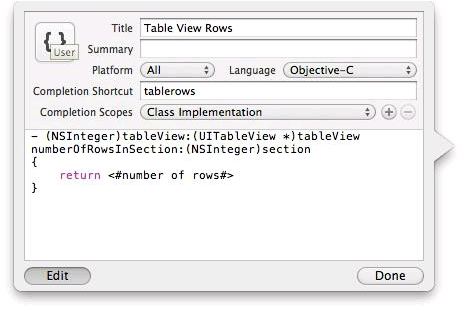
{ return <#number of rows#>;

}

Then fill out the rest of the fields in the edit window as shown in Figure 9.19 and click Done.



Figure9.19 Creatinganewsnippet



Back in ItemsViewController.m, start typing tablerows. Xcode will recommend this code snippet and pressing the return keywillautomaticallycompleteitforyou–andthenumber of rows placeholder will be selected. You’ll have to type in the code to get the number of rows yourself. Snippets aren’t magical – just handy.

Before continuing, make sure to remove the code entered by the snippet because you have already defined tableView:numberOfRowsInSection: in ItemsViewController.m.

Bronze Challenge: Sections

Have the UITableView display two sections – one for items worth more than $50 and one for the rest. Before you start this challenge, copy the folder containing the project and all of its source files in Finder. Then tackle the challenge in the copied project; you’ll need the original to build on in the coming chapters.

Silver Challenge: Constant Rows

Make it so the last row of the UITableView always has the text No more items!. Make sure this row appears regardless of the number of items in the store (including 0 items).

Gold Challenge: Customizing the Table

Make each row’s height 60 points except for the last row from the medium challenge, which should remain 44 points. Then, change the font size of every row except the last to 20 points. Finally, make the background of the UITableView display an image. (This image should be 460x320 pixels or 920x640 pixels depending on whether your device supports retina display. Bonus points for appropriately handling both retina display and non-retina display devices correctly with the same application.)