Errata

Any software project following a schedule ships with bugs. The same is true of books—especially when the book is written using development versions of an OS that is final after the chapters are complete.

The first edition of the book contains three basic types of bugs in either text or code. The first are typos and errors. No matter how many times a book or sample code goes through an editing cycle, some of these always sneak by. The second are small differences between the book and the final version of iOS 7. The last is one major difference in Chapter 10, "Table Views II: Advanced Topics." As you will see, there is a significant difference in how a search display controller works, requiring a rewrite of that section. The rewrite is included at the end of these errata.

There are four critical fixes in the errata. Without making the changes, the code will not compile. One is the first correction for Chapter 9, "Introducing Core Data." The next ones are in Chapter 10, both the second point in the errata for the chapter as well as rewriting search. And the final one is the fix for Chapter 13, "Introducing Blocks."

This document goes through the changes chapter by chapter, section by section. If you are having any strange behaviors as you develop CarValet, this is the first place to turn. The sample code provided on GitHub already incorporates the changes.

Chapter 4: Auto Layout

Putting It All Together

After entering the code in Listing 4-2, the code does not cause an error when you run the app. You can see a similar error by modifying the code from Listing 4-2. For each of the conditions in the `if` statement, reverse the order of removing and adding the constraints. As an example, for the portrait screen condition, move the three lines that call `addConstraints:` above the three that call `removeConstraints:`.

Constraining to the Top or Bottom of Containers

The four steps making changes to `setupLandscapeConstraints` are missing constraints to the top and bottom layout guides. There are three changes to make:

1. Modify the code shown in step 1 so it looks like the following (the changes are in bold):

```
    id topGuide = self.topLayoutGuide;
    id bottomGuide = self.bottomLayoutGuide;
views = NSDictionaryOfVariableBindings(topGuide,
                                       bottomGuide,
                                       addCarView,
                                       separatorView,
                                       viewCarView);
```

2. In step 2, change the constraint string to the following:

   `@"V:[topGuide]-[separatorView]-[bottomGuide]"`

3. In step 3, change the constraint to the following:

   `@"V:[topGuide]-[addCarView]-[bottomGuide]"`

Chapter 5: Localization

Setting Up Localization for the Edit Car Scene

The second bullet point at the top of the section instructs you to add properties to access the labels. Those labels are used in Listing 5-4 but are not shown separately in the book. The labels and elements are shown in Table 1.

Table 1 Property Names for Elements of the Edit Car Scene

String Key	Field
`carMakeFieldLabel`	The label beside the Make field
`carModelFieldLabel`	The label beside the Model field
`carYearFieldLabel`	The label beside the Year field
`carFuelFieldLabel`	The label beside the Fuel field

Formatting and Reading Numbers

The last paragraph in the section incorrectly calls the interim project "`CH06 CarValet Arabic Starter`" instead of "`CH05 CarValet Arabic Starter`."

Adding Arabic Strings

Figure 5-16 and Figure 5-17 should show the Total Cars label right justified, the same justification shown by the Car Info Label. This issue is fixed in the sample code by updating the justification for `totalCarsLabel` at the same time `carInfoLabel` is updated.

Text Alignment

The first paragraph of this section uses Total Cars as an example of labels that automatically work in either right-to-left or left-to-right languages. Unfortunately, Total Cars is not set up to do this. It is constrained to both the leading and trailing edges and the `UILabel` is as wide as the presentation area.

A better example is the Car Number label. It is only constrained to the leading edge. That means it will be only as wide as needed for the content.

Chapter 6: Scrolling

Adding a Scroll View to the View/Edit Scene

1. After opening the project but before starting to add a scroll view, make sure the Arabic languages flag is turned off in the Scheme. You also need to make sure the Simulator is set back to the United States region, or to whatever region you have been using for development.
2. In step 8 of adding the scroll view, there should be two more bullet points to the end:
   - Pin the bottom of the view car group the standard distance from the bottom of its container and update frames.
   - Select all the children of the view car group and update frames.

Resizing for the Keyboard

Resizing the Scroll View

`defaultScrollViewHeightConstant` should be `defaultScrollViewHeightConstraint`. This occurs in two places:

- In the code declaring the variable in the `@implementation` statement. It should be:

```
@implementation CarEditViewController {
    CGFloat defaultScrollViewHeightConstraint;
}
```

- Shortly after in the added code for `viewDidLoad`, which should be:

```
defaultScrollViewHeightConstant =
                    self. defaultScrollViewHeightConstraint.constant;
```

Populating the Scroll View

Listing 6-3, "Initial Code for `CarImageViewController`" should use `CarImageViewController` instead of `CarImagesViewController`. There is an extra ‘s’ in the book.

What Car Is This?

The last paragraph before the code should refer to `viewDidAppear:` not `viewWillAppear:`. The code in the book at the end of the section is correct.

Chapter 7: Navigation Controllers I: Hierarchies and Tabs

Populating the Toolbar

There are two corrections in this section, one for the default navigation controller toolbar state and one for a typo in a class name.

1. Near the bottom of the section is a paragraph on hiding and showing the toolbar and using the `toolbarHidden` property. The book incorrectly states that the default value is `YES`. Adding a toolbar to a navigation controller in IB will set the toolbar to visible.

   You can see this by selecting the navigation controller and looking for the Shows Toolbar checkbox in the Attributes inspector. Adding a toolbar will check the checkbox—that is, it will set `toolbarHidden` to `NO`. In the sample code, the Shows Toolbar checkbox is unchecked.

   When you create toolbars, be aware that they will show up unless you either explicitly set the `toolbarHidden` property to `NO` in your view controller, or uncheck the box in IB. If you do uncheck the box, you will have to explicitly show the toolbar by setting the property to `YES`.

2. The last bullet point in the section should refer to `CarImageViewController`, not `CarImagesViewController`—once again, an extra ‘s’. The error is in the last bullet point above the section entitled “Revisiting Location.”

Revisiting Localization

Remove the ellipses character at the end of the last line of `previousCar:`. The line is easy to find as it has the `// 4` comment at the end.

Moving Car Images to the Tab Bar

1. In the first set of steps, the first sentence of step 2 should be:

"Ctrl-drag a connection from the tab bar controller to the car images view controller."

2. In the second set of steps fixing the Car Number label, there should be one step before the existing ones, step 0:

   "Choose the label, and then choose Update Frames from the constraint fixing popup."

Chapter 8: Table Views I: The Basics

Project TableTry

Figure 8-3 shows content in the prototype and static cells. When a table is first created, the prototype or static cells have no content. Once you change styles, switching between prototype and static might preserve the content.

Phase I: Replacing the Add/View Scene

Step 5 is missing the property qualifiers for `myCar`. The declaration should be:

```
@property (strong, nonatomic) Car *myCar;
```

Adding a Car View Cell

In step 1 there are no synthesized variables. This is an older concept used in iOS 5 and, to a lesser extent, in iOS 6. Step 1 should be:

1. Add the property `dateCreated` of type `NSDate` to the `Car` object. Set it to `[NSDate date]` in the base class initializer.

Adding New Cars

Listing 8-2 shows an older way to create a new Car object. It should be:

```
Car *newCar = [Car new];
```

Adding the Year Edit Protocol

In step 4, the sentence before the code refers to the wrong methods names; it should be:

"Fill out the `cancelTouched:` and `doneTouched:` methods:"

Chapter 9: Introducing Core Data

Updating Basic Table View Data Source Methods

The code for step 3 is syntactically incorrect and will not compile. The sample code is correct. The correct code for this step is:

```
cell.myCar = [fetchedResultsController objectAtIndexPath:indexPath];
```

Chapter 10: Table Views II: Advanced Topics

This chapter contains four minor errors. In addition, as hinted in the callout in "Searching Tables," the sample code did change. The requirements to enable sorting of a table view outside of searching are significantly different than the content in the chapter. A revised section on searching is at the end of these errata.

Adding and Deleting Sections

The second-to-last paragraph has a comma in the middle of the first method signature as well as an extra 's' after "uncomment." The first sentence should be:

"In `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:` uncomment the `NSFetchedResultsChangeMove` case."

Adding Code to Change the Sort and Sections

Delete step 5 setting `self.tableView.tableHeaderView`. The code will crash if you included this line.

An Index for Searching

Although the code in Listings 10-6 and 10-7 does work, it could be implemented more efficiently. The corresponding methods in the sample code provided with the chapter use the more efficient versions of the routines.

Chapter 11: Navigation Controllers II: Split View and the iPad

Adding the Split View Controller

In Listing 11-1, the image name for the about item should be "info" and not "59-info.png." The current code results in a tab bar item with no icon. The provided sample code is correct. The revised code for the book is the following (the change is in bold):

```
UITabBarItem *aboutItem = [[UITabBarItem alloc]
                           initWithTitle:@"About"
                           image:[UIImage imageNamed:@"info"]
                           tag:0];
```

Adding App Section Navigation

In Table 11-1, there are two errors in the second line of the table:

1. The value of the Cell column should be `2`.
2. The Image Name column value should be `photo`.

Tints, Title, and Generic Detail

In step 2 of the second set of steps, the original root view of the view controller is replaced with the `UIImageView`. The sample code does not do that. It adds the image view to the root view, sets the constraints given in step 2, then uses the left-hand menu to move the image view to the top of the list of views (that is, it moves the image view to the back, behind all other views but does not make it the root view).

If you still choose to replace the root view with the image view, remove the second sentence from step 2. You cannot set constraints on the root view of a controller.

Changing the Image Tint

Step 4 should say, "Open `MainMenuViewController.m`..."

Adapting the Car Table to iPad

In the `viewDidLoad` method in `CarTableViewController`, add a step 4a:

4a. Only set the title in the navbar if there is no `delegate`. Add the following `if` condition around the call to set the title in `viewDidLoad` (the new code is in bold):

```
if (!self.delegate) {
    self.title = NSLocalizedStringWithDefaultValue(
                  @"AddViewScreenTitle",
                  nil,
                  [NSBundle mainBundle],
                  @"CarValet",
                  @"Title for the main app screen");
}
```

Controller Layout on the Storyboard

Step 7 in the first set of steps (placing UX elements in the controller) uses an older version of the screen layout and needs different constraints as well as a different image. As with all other changes, the provided sample code has been updated. Replace step 7 with the following:

7. Place an image view the system distance from the container's leading edge and the bottom of the Make Text entry field. Align the trailing edge with the trailing edge of the Model Text entry field, and set the bottom edge the system distance above the Fuel label.

   Set the background to Sky (to make it easy to see during development; you set it back later).

   Import `CH11 Assets Big Placeholder` into `Images.xcassets`. Set the picture for the image to `big-placeholder` and Aspect Fit.

Step 9 in the second set of steps (constraining the elements) is missing one constraint:

- Reduce the Content Compression Resistance priority by 1 for each axis to `749`.

Chapter 12: Touch Basics

Adding the Taxi View with Drag

The last sentence of the last paragraph says that the taxi view stays inside the screen. Although this is correct, it is possible for the taxi view to get stuck under the navigation bar. This happens when the view is dragged up under the bar and the pull-down push notifications panel is dragged down. At this point, the taxi view is under the navigation bar and there is no way to tap it.

It is possible to change the code from Listing 12-11 to exclude the navigation bar from the dragable area. To do this, you would need to change the first argument of the call `CGRectContainsRect`. The new argument would be a rectangle that excluded the navigation bar rectangle from the superview's frame. This is not in the sample code and is left as an extra challenge.

Chapter 13: Introducing Blocks

Pulsing and View Constraints

Listing 13-1 uses a property for the `labelTaxiSpace` constraint. This needs to be created at the same time `taxiWidth` and `taxiHeight` are set up. To create the constraint, drag a connection to the constraint between the bottom of the taxi view and the top of the label below that view. Without this constraint, the code will not work. Again, the sample code is correct.

Chapter 10: Replacement Section for Searching Tables

Much of the introductory material in this section is valid, as are some of the steps for adding searching. However, using both a segmented control for sorting and a search display controller requires changing how the scene is built.

To make things work, the base view controller needs to change from a `UITableViewController` to a `UIViewController` containing the segmented control, search bar and a table view. And that requires large number of changes in the implementation.

Because of that, the rest of this errata document is a replacement for the section and reflects the sample code provided for this book. The replacement starts at the "Searching Tables" section header and continues until the section entitled "An Index for Searching." Because all the figures in the book are correct, the text below contains only references and placeholders.

Searching Tables

The final functionality that could help the valet is to be able to search for a specific car. If the customer knows she has a BMW, filtering the content so that only BMWs show is a quick way to find her car.

There are two basic options for adding search to a table. One is to manage all the UX elements and behaviors manually. You add the search bar in some part of the scene, maintain state between searching or not, handle all the messages from the search bar, and choose how to present those results. This is generally a lot of work, especially compared to the second option shown in this section, using the built-in `UISearchDisplayController`.

Before you look at how to add searching to your table, it is useful to understand how the search display controller works. In Figure 10-10, you can see the two stages of a search. On the left-hand side, the user has tapped in the search field to bring up the keyboard and cover the main table with a translucent view. On the right-hand side, the current search results are displayed in a table below the search field.

Figure 10-10 Search table view

Although the process of implementing a search seems like it might be complicated, most of the hard work is done by the `UISearchDisplayController`. When the user taps in the search field, the search display controller brings up the keyboard and displays the black translucent view. It also creates a new table for any search results, but it defaults to using your view controller's table data source and delegate methods. This last part is very important to understand and is a source of confusion for many people when they implement searching.

The first time a user types something into the search field, the following things happen:

1. Your controller gets a callback to update the search results.
2. The search results controller creates a search results table.
3. The new table view calls your view controller's table data source methods to build the search results table.
4. The search results table is displayed.

As the user updates the search text, callbacks are used to update the set of found items. When the number of items changes, the search results table updates.

Because the search results table uses the same data source and delegate, methods such as `numberOfSectionsInTableView:`, `tableView:numberOfRowsInSection:`, and `tableView:cellForRowAtIndexPath:` can be called for different tables. And even when these methods are called for the search results table, the main table view, `self.tableView`, is still valid. Figure 10-11 shows how this happens. The search results table (highlighted in red) is above the original table (highlighted in green) in the view hierarchy, so it covers the main table, though the main table is still there. It will still respond to messages.

Figure 10-11 Two table views for a search

There is one more important thing to understand about using a search display controller: It expects to be the table header. For many types of data, this is not an issue. For CarValet, you are using the table header to show a sort bar. Although the search controller can provide a scope bar, the same kind of element, it only does so when search is active. As of iOS 7.0, there is no way to force the scope bar to show when not searching.

That leaves three choices for including sorting and a search display controller: remove sorting, allow sorting only when the search bar is active, or provide a UX for sorting "outside" of the table view. In this chapter you implement the last option.

You will do this in three stages:

- Convert the existing `CarTableViewController` from a `UITableViewController` to a `UIViewController` that also contains the table view and segmented controller.
- Add the search display controller and enable searching.
- Enable showing car details for any found cars.

The first step is converting from a `UITableViewController`.

Converting `CarTableViewController`

The view controllers provided by the system help manage complexity in the UX and/or in behavior. `UIViewController` provides the basic class for managing a scene and is the

superclass for any of the other view controllers. You have already seen how useful `UINavigationController` and `UITabBarController` are for managing overall app navigation.

`UITableViewController` does provide some help with managing a `UITableView`, but a quick look at the documentation shows there are only three properties and one specialized initialization method. In other words, there is not much functionality provided by the class. This makes converting from a table view controller to a view controller containing a table view much easier.

In fact, this kind of arrangement is much more common even in mildly complex apps. In addition to enabling the kind of search behaviors shown in this section, it provides flexibility for other behaviors, such as the detail view in Yelp, chat heads in Facebook, and the Tweet detail in Twitter. Note that there are multiple ways to implement any of those app features; a view controller with a table view is just one of them.

Adding Searching

When you add search capability to a table, you are really adding a few things:

- A `UISearchBar` for the user to enter search terms
- A `UISearchDisplayController` for managing the display of search results
- Methods from the `UISearchDisplayDelegate` protocol for updating the found items, as well as other state information
- Any required updates to the table view data source and/or delegate methods
- Any other required methods and/or variables for updating and maintaining state

Although using a `UISearchDisplayController` is not required, it does add a lot of functionality for very little work. It is possible to manage all of the search behavior yourself. That requires managing the search bar, as well as how to present search results in either one table or a second table. The work can become quite complex.

Because CarValet will allow sorting and search, before you can add any search functionality, you need to convert `CarTableViewController` to use a `UIViewController` instead of a `UITableViewController`.

Converting `CarTableViewController` to Use `UIViewController`

Converting to a new type of view controller is done in three stages:

- Create a replacement scene on the storyboard.
- Lay out the new scene using appropriate elements from the old scene.
- Delete the old scene and hook up the new scene.

Once those steps are done, the new scene will have the same functionality as the old one. The first step is creating a replacement scene:

1. Open the main iPhone storyboard and drag a `UIViewController` above the current `CarTableViewController` scene.
2. Set the background color of the new controller's main view to Mercury.
3. Ctrl-drag from the navigation controller to the new scene and set it as the root view controller. This will break the connection with the existing car table view scene.

You now have a new empty `UIViewController` with a navigation bar. The next stage is laying out the new scene using elements from the old one:

1. Drag just the segmented control from the old scene underneath the navigation bar of the new scene. Make sure you do not drag the enclosing `UIView`.
2. Set the constraints for the segmented control and update frames. The new constraints are:
   - `0` points from the leading and trailing superview
   - `4` from the top neighbor (the top layout guide)
   - A height of `20` points
3. Drag in a new `UITableView` from the Object Library in the Utilities area just below the segmented control. Use the Size inspector set the Row Height to `68`.
4. Set the constraints for the table view:
   - The top is `1` point from the segmented control.
   - The sides and bottom are `0` from the leading, trailing, and bottom (bottom layout guide) edges of the superview.
5. Make sure the new table view is using prototypes and drag the `CarCell` prototype from the old table into the new one. Make sure there is only one prototype for the new table.
6. Drag the Add bar button item from the old scene to the right side of the navigation bar in the new scene.
7. Drag the Edit and Done buttons under the Exit Segue in the new controller.

The new scene now looks close to the old one. Even the sort bar is in roughly the same place. You are now ready to remove the old scene and hook up the new scene. This step is very important. Even after you change the class, none of the `IBOutlet`s or `IBAction`s are connected. And that means, at best, the scene will not work.

Finish converting the new scene into the new `CarTableViewController` using these steps:

1. Delete the old scene and move the new one into the same position.
2. Open up `CarTableViewController.h` and change the superclass from `UITableViewController` to `UIViewController`. The new `@interface` statement is:

   ```
   @interface CarTableViewController : UIViewController
   ```

3. Open up the storyboard and change the class of the new view controller to `CarTableViewController`.
4. Reconnect `IBAction`s:
   - Connect the Add button to `newCar:` by Ctrl-dragging from the button to the controller.

     Note that the appearance of the action names in the popup tell you there are no currently connected actions. Any actions connected to a UX element have a dash '-' in front of them.
   - Connect the Edit and Done buttons to `editTableView:`.
   - Connect the segmented control to `carSortChanged:`.
5. Open an Assistant editor in IB and make sure it is showing `CarTableViewController.h`, and then Ctrl-drag from the new table view to the `.h` file and create a new property called `tableView`.
6. Now reconnect the `IBOutlet`s. Each property will have a small white outlined circle next to it. Drag from those circles to the appropriate UX elements.

   Alternately, drag from each element to the related property. This is different from creating a new property. Instead of releasing the mouse in an empty part of the `.h` file, you wait until the desired property highlights, and then release the mouse button.
7. Set the `delegate` and `dataSource` of the table view to the view controller. You can do this by Ctrl-dragging from the table view to the controller.

The new controller now has the same functionality as the old one. Run the app and make sure everything works. Check all the behaviors and make sure each `IBOutlet` is connected to the right place and every `IBAction` is performed.

Adding the Search Display Controller

Now the list of cars is ready for you to add search. This chapter shows you how to integrate the basic features of the search display controller. One of the optional features is displaying a scope bar when search is active. This bar looks similar to the existing segmented control used to sort the contents of the table.

Although this chapter does not cover showing or using the scope bar, the sample code for this chapter includes `CH10 CarValet with ScopeBar`. This implementation of CarValet includes the sort bar above the table, a scope bar for the search display controller, and code to keep the two synchronized. Another extra sample, `CH10 CarValet UISearchBar`, uses a search bar without using a search display controller or a second table view for results.

For this chapter, the next step is to add the search bar with the search display controller:

1. Open `CarTableViewController` in the storyboard.
2. Look for the search bar with a search display controller, shown in Figure 10-12, in the objects shown in the utilities area. Drag it into the table header position for the table

view. The easiest way to do this is to drag it into the left-hand list of view and place it between the Table View element and the Car Table View Cell prototype.

You also can try dragging it into the scene on the storyboard between the top of the table view and the top of the prototype cell.

Figure 10-12 Search bar and display controller

3. Set the search Bar Tint to Mercury.

Adding the Search Predicate

You are already using a fetched results controller for displaying data. You need to limit it to cars matching the search criteria. In addition to sorting cars, the fetch request can also filter the data. It does this using a predicate.

Predicates, or `NSPredicate`, provide a flexible way to filter data on multiple criteria. You need a simple search, based on the current table grouping. For example, if cars are grouped by model, you look for any cars with a `model` attribute that contains the search string.

Whenever the search term changes, the search display delegate is sent a `searchDisplayController:shouldReloadTableForSearchString:` message. The method returns a `BOOL` that controls whether the search results table is updated. Update the predicate of your fetched results controller by following these steps:

1. Open `CarTableViewController.h` and add `UISearchDisplayDelegate` to the list of supported protocols.
2. In `CarTableViewController.m`, add the following just above `carToView`:

```
#pragma mark – UISearchDisplayDelegate

#pragma mark - ViewCarProtocol
```

3. Put the code in Listing 10-5 between the two `#pragma` marks you just added in step 2.

Listing 10-5 **Updating a Car Search Predicate**

```
- (BOOL)searchDisplayController:(UISearchDisplayController *)controller
         shouldReloadTableForSearchString:(NSString *)searchString {
    if (searchString && ([searchString length] > 0)) {                    // 1
        fetchRequest.predicate = [NSPredicate predicateWithFormat:        // 2
                                  @"%K contains[cd] %@",                  // 3
                                  [[fetchRequest.sortDescriptors          // 4
                                    objectAtIndex:0] key],
                                  searchString];                          // 5
    } else {
        fetchRequest.predicate = nil;                                     // 6
```

```
    }

    NSError *error = nil;
    [fetchedResultsController performFetch:&error];                          // 7

    if (error != nil) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return YES;                                                              // 8
}
```

Here's what happens in the numbered lines in Listing 10-5:

1. Check whether there is a search string with at least one character.
2. Set up a predicate based on the search string. Build the predicate using a predicate format string. Note that these are not the same as format strings. See the "Predicate Programming Guide" in the Apple documentation.
3. The string builds a predicate that checks whether the value of key, `%K`, contains the specified string `%@`. A key is the name of an object property. `[cd]` means use a case- and diacritical-insensitive comparison.
4. The name of the property, or key, is the same as the one used for sorting the data. For a table sorted by car model, the property is `model`.
5. Specify the search string to use for `%@` in the predicate format string.
6. There is no search string, so clear any existing predicate.
7. Fetch any car objects that match the filter.
8. Tell the search results table to update.

Run the app, tap to sort the table by make, and then try to enter `b` into the search field. You get a crash report similar to the following (bold added for emphasis):

```
2013-08-30 20:48:01.641 CarValet[22058:a0b] *** Assertion failure in -
[UISearchResultsTableView dequeueReusableCellWithIdentifier:forIndexPath:],
/SourceCache/UIKit_Sim/UIKit-2891.1/UITableView.m:5184
2013-08-30 20:48:01.653 CarValet[22058:a0b] *** Terminating app due to uncaught
exception 'NSInternalInconsistencyException', reason: 'unable to dequeue a cell with
identifier CarCell - must register a nib or a class for the identifier or connect a
prototype cell in a storyboard'
```

The second error tells you where to look for the problem. The table view is unable to dequeue a cell with an identifier of `CarCell` because no such cell has been registered. You might wonder why this crash occurs because it worked fine until now.

Look at the line that allocates the cell in `tableView:cellForRowAtIndexPath:`

```
    CarTableViewCell *cell = [tableView
                              dequeueReusableCellWithIdentifier:CellIdentifier
                                forIndexPath:indexPath];
```

The dequeue message is sent to `tableView`, whatever view is passed in to the method. In this case, `tableView` is the search results view, not the main table. `CarCell` is registered with the main table, not the search table.

Although there is a method to register a cell class with a cell identifier, that does not work for the search table. `CarCell` is based on the storyboard prototype custom cell you set up earlier in this chapter. Xcode associated the cell identifier with the prototype. You have no way to use that same mechanism for the search table. Instead, you can change the following line in `tableView:cellForRowAtIndexPath:` (the change is shown in bold):

```
    CarTableViewCell *cell = [self.tableView dequeueReusableCellWithIdentifier:
                                CellIdentifier];
```

Make sure you remove the index path part of the dequeue call. When you have done that, run the app again, tap to sort by make, and type `b` into the search field. The search results table looks something like the left side of Figure 10-10, depending on your data.

All the correct data is displayed, but the custom cells do not fit. Once again, it is because the search results table is different. You need to set the `rowHeight` to `68` so it is high enough for the custom cell. Add the following method after the code from Listing 10-5:

```
- (void)searchDisplayController:(UISearchDisplayController *)controller
    didLoadSearchResultsTableView:(UITableView *)tableView {
    tableView.rowHeight = self.tableView.rowHeight;
}
```

The method is called the first time the search display controller creates and loads the search results table. The one line sets the row height for the new table to the row height of the main table.

Run the app, change the group and search, and now the search results table looks correct. Cancel the search, and you see another error:

```
2013-06-03 15:06:01.668 CarValet[14554:c07] *** Terminating app due to uncaught
exception 'NSRangeException', reason: '*** -[__NSArrayM objectAtIndex:]: index 2 beyond
bounds [0 .. 0]'
```

Finding the source of this error requires setting a breakpoint for all exceptions, which is covered in Chapter 14, “Instruments and Debugging.” When the breakpoint is set, run the app. You see that the app crashes in `tableView:titleForHeaderInSection:` and, more specifically, it crashes because the fetched results controller has only one section but is being asked for information on a second section.

This error occurs because the fetched results controller is used for both the main and search results tables. Searching can change the data, including the number of sections and the number of cars in each section. When the search is cancelled, no code is run to reset the fetched results controller back to the presearch state. The easiest way to do that is to set the fetch request predicate to `nil`, and that is already done in

`searchDisplayController:shouldReloadTableForSearchString:` when the string is empty.

Add the following to the search display delegate methods:

```
- (void)searchDisplayControllerWillEndSearch:(UISearchDisplayController *)
           controller {
    [self searchDisplayController:controller
        shouldReloadTableForSearchString:@""];
}
```

`searchDisplayControllerWillEndSearch:` is called after the user ends the search but before the main table is updated. Using an empty string results in clearing the fetch request predicate and updating the fetch request controller.

Fixing the Animation

You have probably noticed the messy animation when you tap in the search bar. The search display controller assumes it owns the table header. It also assumes that the only thing above the table view is either a navigation bar or the top of the view. That results in a bad interaction with the existing sort bar.

The right thing to do is move the existing sort bar out of the way. The easiest way to do that is to hide the view and make sure it has a height of zero. That last part is important. If you just hide the view, the space taken by that view is still there.

Changing the height of a view using auto layout requires changing the constant on a constraint or constraints so that the layout engine sets the height to zero. The easiest way to do this is using a height constraint:

1. Open `CarTableViewController` in the storyboard and show the `.h` file in an Assistant editor.
2. Create a new property `carSortViewHeightConstraint` for the vertical height constraint of the sort bar. You can do this by Ctrl-dragging from the constraint in the storyboard or from the list on the left of IB.
3. Insert the following code at the top of the existing search display delegate methods just above `searchDisplayController:shouldReloadTableForString:`

```
- (void)searchDisplayControllerWillBeginSearch:
            (UISearchDisplayController *)controller {
    self.carSortControl.hidden = YES;
    self.carSortViewHeightConstraint.constant = 0.0f;
}

- (void)searchDisplayControllerDidEndSearch:
            (UISearchDisplayController *)controller {
    self.carSortControl.hidden = NO;
    self.carSortViewHeightConstraint.constant = 20.0f;
```

```
}
```

The first method hides the sort bar view and then sets the height to zero. The second one shows the sort bar and sets the height back to the original. For production code, it is a good idea to replace the number arguments with constants. That way the values are only set in one place and are easy to change. In addition, you can use self-documenting names such as `kZeroHeightViewConstant`.

Another way to add flexibility is adding a property or iVar for the full height value of the status bar constraint. You set it in `viewDidLoad` and then use that value in `searchDisplayControllerDidEndSearch:`.

This time when you run the app, tapping in the search bar results in a much better looking animation, though ending the search results in the sort bar suddenly appearing, a jarring experience.

The ideal solution would be to animate the sort bar in as the search bar is animating back to the table view header. Unfortunately there is no way to do this with the current version of iOS 7. Instead, you can let the closing search bar animation complete and then animate in your sort bar. You can do that using the `UIView` class animation methods with a small change to `searchDisplayControllerDidEndSearch:` (the new code is in bold):

```
self.carSortControl.hidden = NO;

[UIView animateWithDuration:0.3f
                 animations:^{
                     self.carSortViewHeightConstraint.constant = 20.0f;
                     [self.view layoutIfNeeded];
                 }];
```

The only change is wrapping the line that set the height constraint constant back to `20.0f` in an animation. If you recall from Chapter 4, "Auto Layout," the call to `layoutIfNeeded` is required to update the screen.

Run the app again, change the sort, search, and then cancel. Everything now works. Search again and tap one of the found cars. There are no car details—or if there are, they are wrong.

Showing Details for a Found Car

The car detail view finds what car to display by calling `carToView`, a method from the `ViewCarProtocol`. The cars table uses the index path of the selected cell to look up the car in the fetched results controller. And that is the problem.

This is the current call:

```
currentViewCarPath = [self.tableView indexPathForSelectedRow];
```

Once again, the message is to the main table view when it needs to be to the search table.

The simplest solution is to track what table view is displayed to the user. You know that it is the main table view when the controller first appears. The only times the current table view

switches are when the search display controller loads and unloads the search table. Both of those events have associated methods. Add the following code to track the currently active table:

1. Open `CarTableViewController.m` in an editor and add the following iVar variable declaration to the ones in curly braces just after the `@implementation` statement:

   ```
   UITableView *currentTableView;
   ```

2. Set the current table to the main table by adding this line to the end of `viewDidLoad`:

   ```
       currentTableView = self.tableView;
   ```

3. Set the current table to the search results table by adding the following method just before `searchDisplayControllerDidEndSearch:`

   ```
   - (void)searchDisplayController:(UISearchDisplayController *)controller
    willShowSearchResultsTableView:(UITableView *)tableView {
       currentTableView = tableView;
   }
   ```

   You may wonder why this method was chosen instead of `searchDisplayController:didLoadSearchResultsTableView:`. The reason is that the search table shown to the user may not be the same one each time. The table can be loaded and unloaded many times during a single search session. The chosen method is always called with the current search table view.

4. Set the current table back to the main table with this line at the end of `searchDisplayControllerWillEndSearch:`

   ```
       currentTableView = self.tableView;
   ```

5. Fix the two `ViewCarProtocol` methods, `carToView` and `carViewDone:`, by changing `self.tableView` to `currentTableView`.

Run the app, sort by make, search for a car, and then look at the details. The details show correctly. Now change the model information and tap Done. You get a crash with errors that look like this:

```
2013-06-03 17:07:12.874 CarValet[16312:c07] *** Assertion failure in -[UITableView
_endCellAnimationsWithContext:], /SourceCache/UIKit_Sim/UIKit-
2380.17/UITableView.m:1054
2013-06-03 17:07:21.106 CarValet[16312:c07] CoreData: error: Serious application error.
An exception was caught from the delegate of NSFetchedResultsController during a call
to -controllerDidChangeContent:.  Invalid update: invalid number of sections.  The
number of sections contained in the table view after the update (1) must be equal to
the number of sections contained in the table view before the update (7), plus or minus
the number of sections inserted or deleted (0 inserted, 0 deleted). with userInfo
(null)
```

You have seen similar problems where the sections are not in sync. In this case, the difference in the number of sections is quite large: `1` after the update and `7` before. There is also the name of the method where the failure occurred, `controllerDidChangeContent:`. A quick look at that method shows that it sends a message to the main table view. This particular update is occurring in the search table.

Searching the file shows that there are still 12 other occurrences of `self.tableView`. Not all of them need to change. Some are used to update the state of `currentTableView`, another occurs when the Edit button is touched, and yet another occurs when the table sort is changed.

Of the others, five are in `NSFetchedResultsControllerDelegate` methods, and all of them need to change. You therefore need to use the find-and-replace mechanism, as follows:

1. Choose Edit > Find > Find and Replace.
2. Type `self.tableView` into the top bar and `currentTableView` into the bottom bar. The search area looks like Figure 10-13.
3. Click the right arrow next to the top bar until `self.tableView` in `controllerWillChangeContent:` is highlighted, as shown in Figure 10-13.
4. Click Replace & Find four times.
5. Click Replace to change the occurrence in `controllerDidChangeContent:`.

Figure 10-13 Finding and replacing in the Xcode editor

You need to make one more change to get the full detail flow working. Run the CarValet app, sort the table by make, and use Find to select a car that was visible in the main table before the Find interface opened. Make a change to the model of that car, go back to Find, and cancel the find. The model is changed in the search results table but not in the main table. Although the data model is correct, nothing updated the main table when searching was done.

Add the following line at the end of `searchDisplayControllerWillEndSearch:`

```
[self.tableView reloadData];
```

Now the whole flow works correctly.