Chapter 10
Interacting with Table Views and Using UIAlertController

> There is no learning without trying lots of ideas and failing lots of times.
>
> - Jonathan Ive

Up till now, we only focus on displaying data in a table view. I guess you are thinking how we can interact with the table view and detect row selections. This is what we're going to discuss in this chapter.

We'll continue to polish the FoodPin app, which we have built in the previous chapter (https://www.dropbox.com/s/xrvpamowqykwfrb/FoodPinCustomTable.zip?dl=0), and add a couple of enhancements:

- Bring up a menu when a user taps a cell. The menu offers two options: *Call* and *I've been here*.
- Display a heart icon when a user selects "I've been here".

Through implementing these new features, you will also learn how to use UIAlertController, which is commonly used to display alerts in iOS apps.

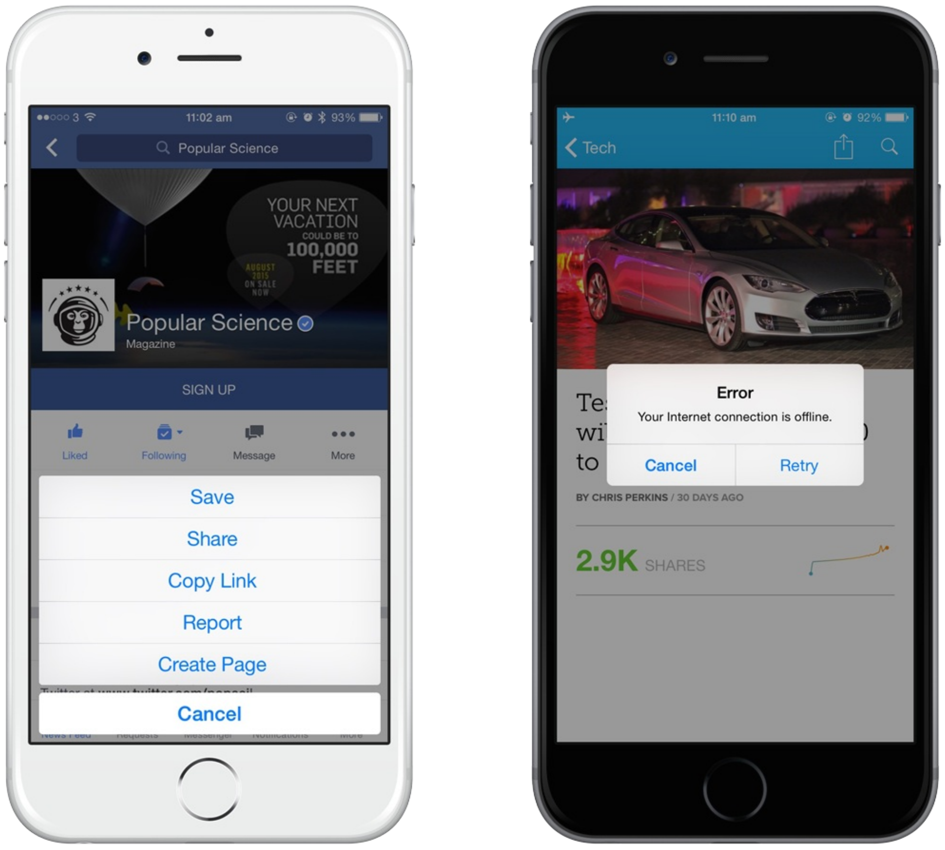


Figure 10-1. Sample alerts in Facebook and Mashable apps

```
Quick note: This class replaces the UIActionSheet and UIAlertView classes for
displaying alerts in iOS 8 (or up).
```

Understanding the UITableViewDelegate Protocol

When we first built the SimpleTable app in Chapter 8, we adopted two delegates, `UITableViewDelegate` and `UITableViewDataSource`, in the `RestaurantTableViewController` class. I have discussed with you the `UITableViewDataSource` protocol but barely mentioned about the `UITableViewDelegate` protocol.

As said before, the delegate pattern is very common in iOS programming. Each delegate is responsible for a specific role or task to keep the system simple and clean. Whenever an object needs to perform a certain task, it depends on another object to handle it. This is usually known as "separation of concerns" in software design.

The `UITableView` class applies this design concept. The two protocols are designed for different purposes. The `UITableViewDataSource` protocol defines methods, which are used for managing table data. It relies on the delegate to provide the table data. On the other hand, the `UITableViewDelegate` protocol is responsible for setting the section headings and footers of the table view, as well as, handling cell selections and cell reordering.

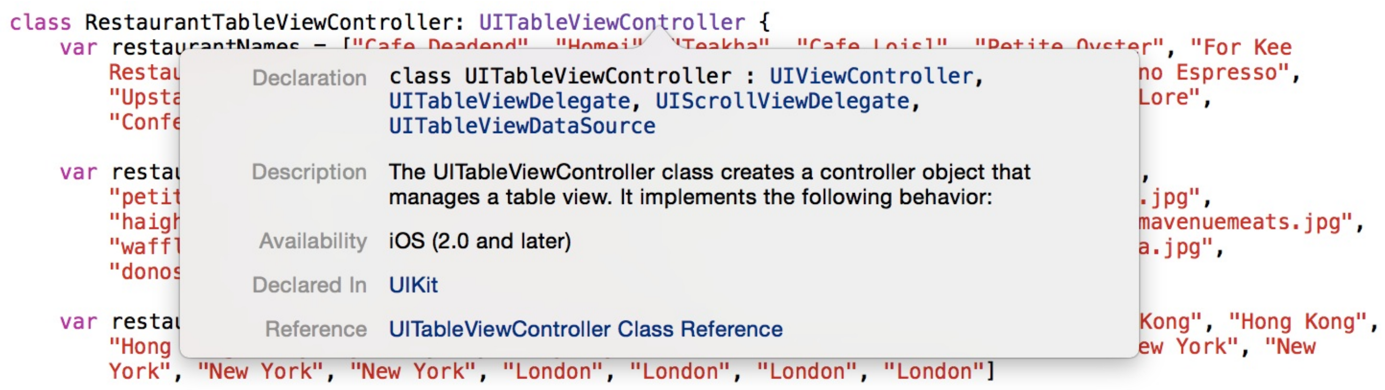
To manage the row selection, we will implement some of the methods in the `UITableViewDelegate` protocol.

Reading the Documentation

Before implementing the methods, you may wonder:

How do we know which methods in UITableViewDelegate to implement?

The answer is "Read the documentation". You're granted free access to the Apple's official iOS developer reference (https://developer.apple.com/library/ios/). As an iOS developer, you need to get used to reading the API documentation. There is no single book on earth to cover everything about the iOS SDK. Most of the time when we want to learn more about a class or a protocol, we have to look up to the API document. Apple provides a simple way to access the documentation in Xcode. All you need to do is place the cursor over a class or a protocol (e.g. UITableViewController) and press 'control-command-?'. This brings up a popover showing the details of the class like the protocols it has adopted.

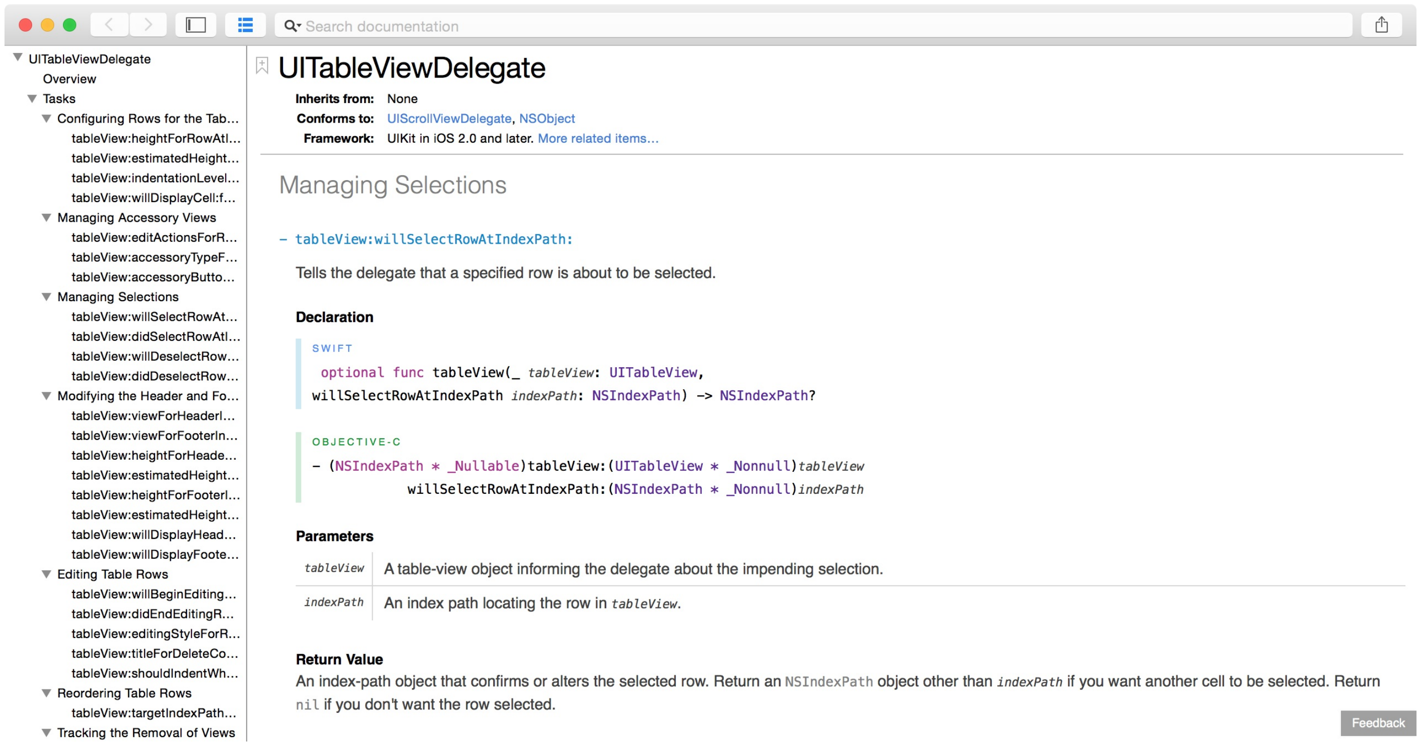


```
class RestaurantTableViewController: UITableViewController {
    var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite Oyster", "For Kee
        Restau                                                                   no Espresso",
        "Upsta                                                                   Lore",
        "Confe

    var restau                                                                   ,
        "petit                                                                   .jpg",
        "haigh                                                                   mavenuemeats.jpg",
        "waffl                                                                   a.jpg",
        "donos

    var restau                                                                   Kong", "Hong Kong",
        "Hong                                                                    ew York", "New
        York", "New York", "New York", "London", "London", "London", "London"]
```

Declaration	`class UITableViewController : UIViewController, UITableViewDelegate, UIScrollViewDelegate, UITableViewDataSource`
Description	The UITableViewController class creates a controller object that manages a table view. It implements the following behavior:
Availability	iOS (2.0 and later)
Declared In	UIKit
Reference	UITableViewController Class Reference

Figure 10-2. Accessing the API documentation by using a shortcut key

Clicking UITableViewDelegate would further bring up a documentation browser. From there, you'll find all the methods defined in the protocol.



- UITableViewDelegate
  - Overview
  - Tasks
    - Configuring Rows for the Tab…
      - tableView:heightForRowAtI…
      - tableView:estimatedHeight…
      - tableView:indentationLevel…
      - tableView:willDisplayCell:f…
    - Managing Accessory Views
      - tableView:editActionsForR…
      - tableView:accessoryTypeF…
      - tableView:accessoryButto…
    - Managing Selections
      - tableView:willSelectRowAt…
      - tableView:didSelectRowAtI…
      - tableView:willDeselectRow…
      - tableView:didDeselectRow…
    - Modifying the Header and Fo…
      - tableView:viewForHeaderI…
      - tableView:viewForFooterIn…
      - tableView:heightForHeade…
      - tableView:estimatedHeight…
      - tableView:heightForFooterI…
      - tableView:estimatedHeight…
      - tableView:willDisplayHead…
      - tableView:willDisplayFoote…
    - Editing Table Rows
      - tableView:willBeginEditing…
      - tableView:didEndEditingR…
      - tableView:editingStyleForR…
      - tableView:titleForDeleteCo…
      - tableView:shouldIndentWh…
    - Reordering Table Rows
      - tableView:targetIndexPath…
    - Tracking the Removal of Views

UITableViewDelegate

Inherits from: None
Conforms to: UIScrollViewDelegate, NSObject
Framework: UIKit in iOS 2.0 and later. More related items…

Managing Selections

`- tableView:willSelectRowAtIndexPath:`

Tells the delegate that a specified row is about to be selected.

Declaration

SWIFT

```
 optional func tableView(_ tableView: UITableView,
willSelectRowAtIndexPath indexPath: NSIndexPath) -> NSIndexPath?
```

OBJECTIVE-C

```
- (NSIndexPath * _Nullable)tableView:(UITableView * _Nonnull)tableView
           willSelectRowAtIndexPath:(NSIndexPath * _Nonnull)indexPath
```

Parameters

tableView	A table-view object informing the delegate about the impending selection.
indexPath	An index path locating the row in *tableView*.

Return Value

An index-path object that confirms or alters the selected row. Return an `NSIndexPath` object other than *indexPath* if you want another cell to be selected. Return `nil` if you don't want the row selected.

Figure 10-3. UITableViewDelegate Documentation

By glancing through the document, you will find the methods below for managing row selections:

- tableView(_:willSelectRowAtIndexPath:)
- tableView(_:didSelectRowAtIndexPath:)

Both methods are designed for row selections. The only difference is that `tableView(_:willSelectRowAtIndexPath:)` is called when a specified row is about to be selected. You may use this method to prevent the selection of a particular cell from taking place. Typically, you use the `tableView(_:didSelectRowAtIndexPath:)` method. It is called after the user selects a row, to follow up with the row selection. We will implement this method to perform addtional tasks (e.g. bringing up a menu) after a row is selected.

Managing Row Selections by Implementing the Protocol

Okay, that's enough for the explanation. Let's move onto the interesting part and write some code. In the FoodPin project, open the `RestaurantTableViewController.swift` file and implement the `tableView(_:didSelectRowAtIndexPath:)` method in the `RestaurantTableViewController` class:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath
indexPath: NSIndexPath) {

    // Create an option menu as an action sheet
    let optionMenu = UIAlertController(title: nil, message: "What do you want
to do?", preferredStyle: .ActionSheet)

    // Add actions to the menu
    let cancelAction = UIAlertAction(title: "Cancel", style: .Cancel, handler:
nil)
    optionMenu.addAction(cancelAction)

    // Display the menu
    self.presentViewController(optionMenu, animated: true, completion: nil)

}
```

The above code creates an option menu by instantiating a `UIAlertController` object. When a user taps any rows in the table view, this method will be called automatically to bring up an action sheet showing a 'What do you want to do' message and a cancel button. Try to run the project to have a quick test. The app should now be able to detect a touch.

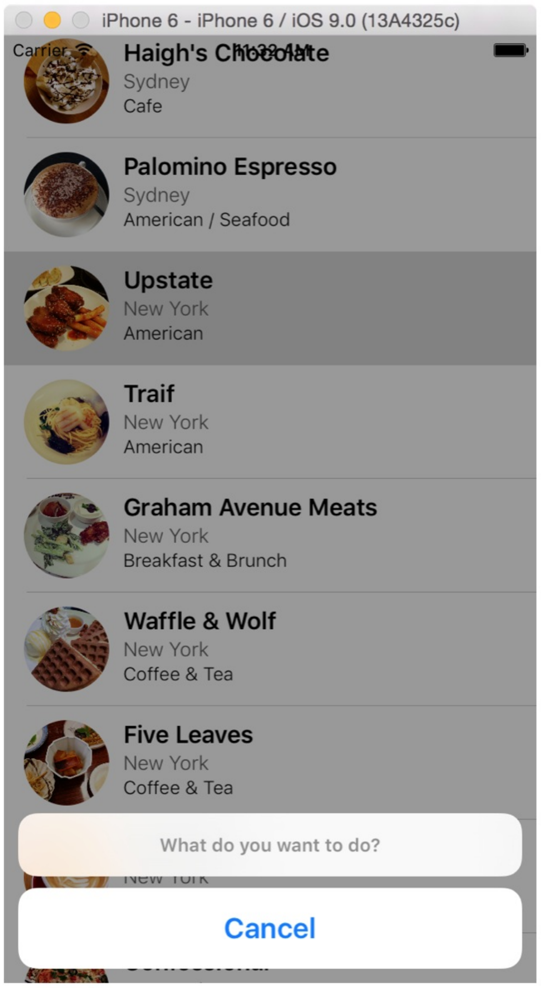


Figure 10-4. Displaying an action sheet

More on UIAlertController

Before we move on, let me talk more about the `UIAlertController` class. The `UIAlertController` class was first introduced in iOS 8 to replace both `UIAlertView` and `UIActionSheet` classes from older versions of iOS SDK. It is designed for displaying alert messages to a user.

Referring to the code snippet in the previous section, you can specify the style of the `UIAlertController` object through the `preferredStyle` parameter. You can either set its value to `.ActionSheet` or `.Alert`. Figure 10-5 displays the sample alert styles.

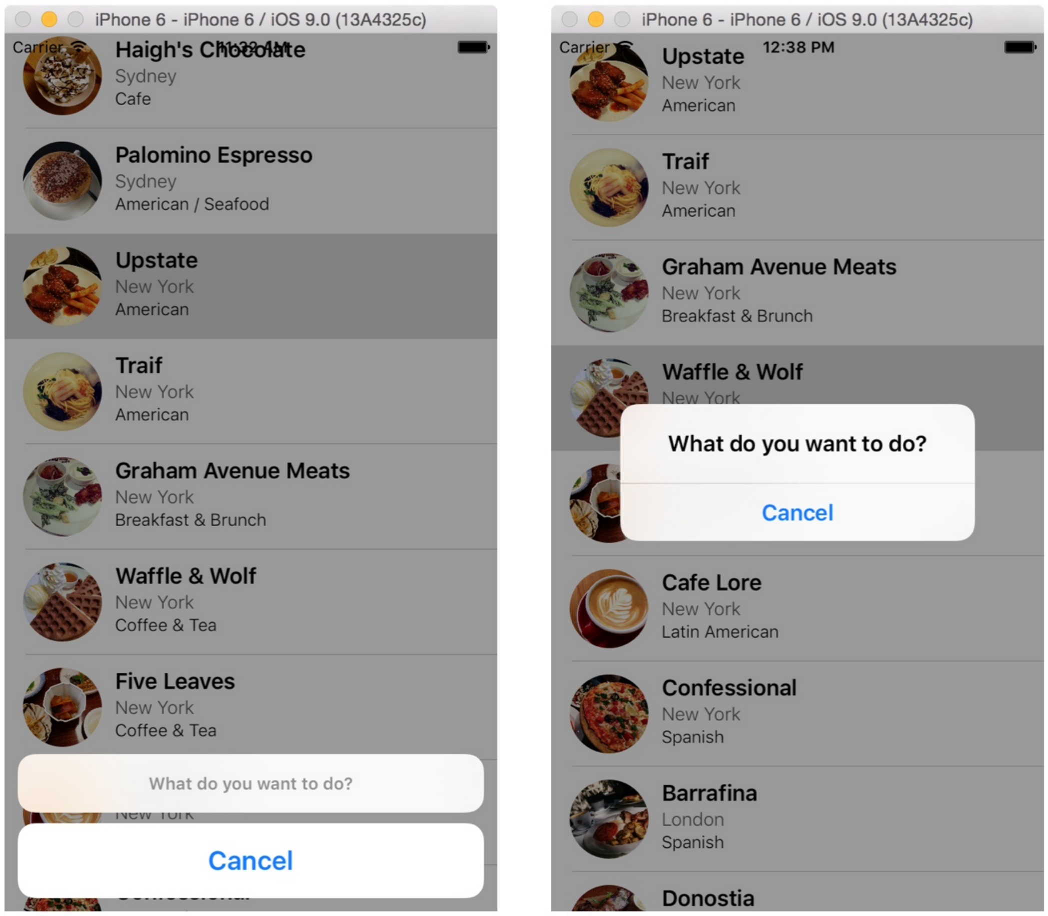


Figure 10-5. ActionSheet (left) and Alert (right)

In addition to displaying a message to a user, you can assign actions to the alert controller to give users a way to respond. To do that, you create a `UIAlertAction` object with your preferred title, style, and the block of code to execute for the action. In the code snippet, we create a `cancelAction` object with the title 'Cancel' and '.Cancel' style. There is nothing to perform when a user selects the cancel action. Thus, the handler is set to `nil`. After the `UIAlertAction` object is created, you can assign it to the alert controller by using the `addAction` method.

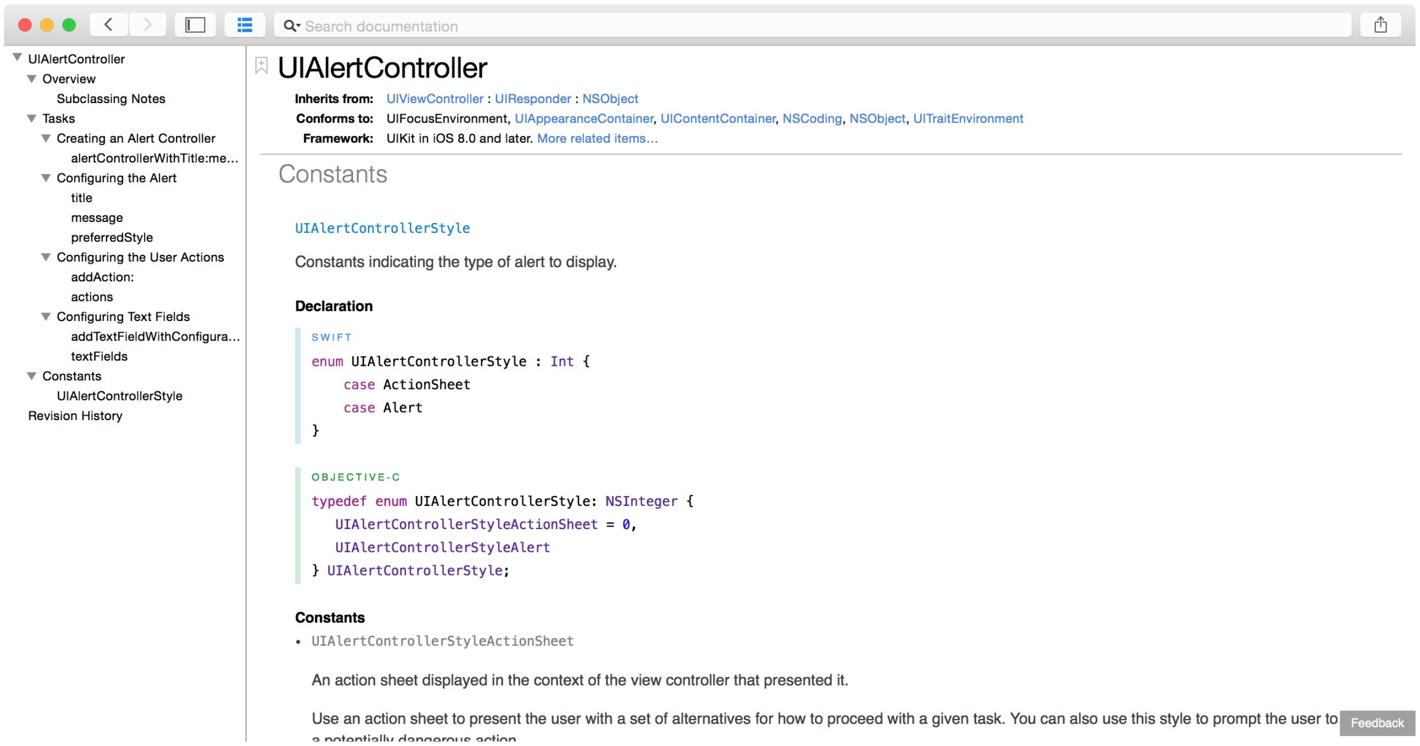
When the alert controller is configured properly, you can simply present it using the `presentViewController` method.

This is how you use the `UIAlertController` class to present an alert. As a beginner, you may have a couple of questions in mind:

- How do I know the available values of the `preferredStyle` parameter when creating a `UIAlertController` object?
- The dot syntax looks new to me. Shouldn't it be written as `UIAlertControllerStyle.ActionSheet` ?

Both are good questions.

For the first question, again the answer is "Refer to the documentation". In Xcode, you can place the cursor over the `preferredStyle` parameter and press `control-command-?` . Xcode will show the method declaration. You can further click `UIAlertControllerStyle` to read the API reference. As you can see in figure 10-6, `UIAlertControllerStyle` is an enumeration, which defines two possible values: `ActionSheet` and `Alert` .



UIAlertController

Inherits from: UIViewController : UIResponder : NSObject
Conforms to: UIFocusEnvironment, UIAppearanceContainer, UIContentContainer, NSCoding, NSObject, UITraitEnvironment
Framework: UIKit in iOS 8.0 and later. More related items...

Constants

`UIAlertControllerStyle`

Constants indicating the type of alert to display.

Declaration

SWIFT

```
enum UIAlertControllerStyle : Int {
    case ActionSheet
    case Alert
}
```

OBJECTIVE-C

```
typedef enum UIAlertControllerStyle: NSInteger {
   UIAlertControllerStyleActionSheet = 0,
   UIAlertControllerStyleAlert
} UIAlertControllerStyle;
```

Constants

- `UIAlertControllerStyleActionSheet`

  An action sheet displayed in the context of the view controller that presented it.

  Use an action sheet to present the user with a set of alternatives for how to proceed with a given task. You can also use this style to prompt the user to

Figure 10-6. UIAlertControllerStyle

```
Quick note: An enumeration is a common type in Swift that defines a list of
possible values for that type. The UIAlertControllerStyle is a good example.
```

We can refer to the values using `UIAlertControllerStyle.ActionSheet` or

`UIAlertControllerStyle.Alert`. So you can write the code like this when creating a UIAlertController:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want to
do?", preferredStyle: UIAlertControllerStyle.ActionSheet)
```

There is nothing wrong with the code above. Swift gives developers a shorthand and helps us type less code. Because the type of the `preferredStyle` parameter is already known (i.e. `UIAlertControllerStyle`), Swift lets you use a shorter dot syntax by omitting `UIAlertControllerStyle`. This is why we instantiate the `UIAlertController` object like this:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want to
do?", preferredStyle: .ActionSheet)
```

The same applies to `UIAlertActionStyle`. The `UIAlertActionStyle` is an enumeration with three possible values: *Default*, *Cancel* and *Destructive*. When creating the `cancelAction` object, we also use the shorthand syntax:

```
let cancelAction = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
```

Adding Actions to the Alert Controller

Now let's add two more actions to the alert controller:

- Call action - Call the selected restaurant. We will populate a fake phone number and display 'Call 123-000-x'.
- "I've been here" action - When selected, this option adds a checkmark to the selected restaurant.

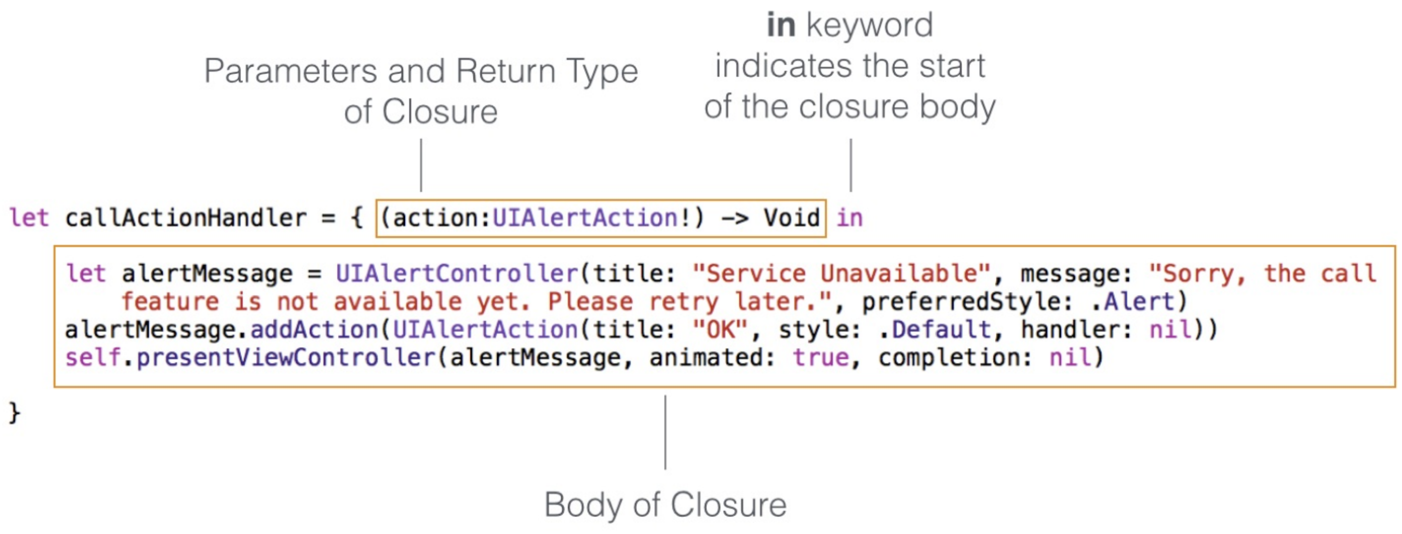
In the `tableView(_:didSelectRowAtIndexPath:)` method, add the following code for the "Call" action. You can insert the code after the initialization of `cancelAction`:

```
let callActionHandler = { (action:UIAlertAction!) -> Void in
    let alertMessage = UIAlertController(title: "Service Unavailable", message:
"Sorry, the call feature is not available yet. Please retry later.",
preferredStyle: .Alert)
    alertMessage.addAction(UIAlertAction(title: "OK", style: .Default, handler:
nil))
    self.presentViewController(alertMessage, animated: true, completion: nil)
}

let callAction = UIAlertAction(title: "Call " + "123-000-\(indexPath.row)",
style: UIAlertActionStyle.Default, handler: callActionHandler)
optionMenu.addAction(callAction)
```

In the above code, you may not be familiar with the `callActionHandler` object. As mentioned before, you can specify a block of code as a handler when creating a `UIAlertAction` object. The block of code will be executed when a user selects the action. Previously, we specify `nil` for the `cancelAction` object. That means we do not have any follow-up action for the Cancel button.

For the `callAction` object, we assign it with `callActionHandler`. The code block displays an alert, telling the user that the call feature is not yet available.

In Swift, this block of code is known as *Closure*. Closures are self-contained blocks of functionality that can be passed around in your code. It is very similar to *blocks* in Objective-C. Like the example above, one way to provide the action closure is to declare it as a constant or variable with the block of code as the value. The first part of the code block is identical to the definition of the handler parameter of `UIAlertAction`. The `in` keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure will begin. Figure 10-7 illustrates the syntax of a closure.


Parameters and Return Type of Closure
in keyword indicates the start of the closure body
Body of Closure

```
let callActionHandler = { (action:UIAlertAction!) -> Void in
    let alertMessage = UIAlertController(title: "Service Unavailable", message: "Sorry, the call
        feature is not available yet. Please retry later.", preferredStyle: .Alert)
    alertMessage.addAction(UIAlertAction(title: "OK", style: .Default, handler: nil))
    self.presentViewController(alertMessage, animated: true, completion: nil)
}
```

Figure 10-7. Closure

The title of the `callAction` object is a fake phone number. It is generated by concatenating '123-000-' with the selected row index. As you can see from the code, Swift allows developers to concatenate strings using the addition operator (+). In addition, you can create a string from an integer or other literals. All you need is to wrap a pair of parentheses, prefixed with a backslash like this:

```
"Call " + "123-000-\(indexPath.row)"
```

```
Quick note: String concatenation has been covered in the Playgrounds chapter.
If you have gone through the exercise in chapter 2, it's time to revisit it.
Alternatively, you can refer to the appendix.
```

Following the implementation of the Call action, add the following lines of code for the "I've been here" action:

```
let isVisitedAction = UIAlertAction(title: "I've been here", style: .Default,
handler: {
            (action:UIAlertAction!) -> Void in

            let cell = tableView.cellForRowAtIndexPath(indexPath)
            cell?.accessoryType = .Checkmark
        })
optionMenu.addAction(isVisitedAction)
```

The above code shows you another way to use closure. You can write the closure inline as a parameter of the handler. This is the preferred way as the code is clearer and more readable.

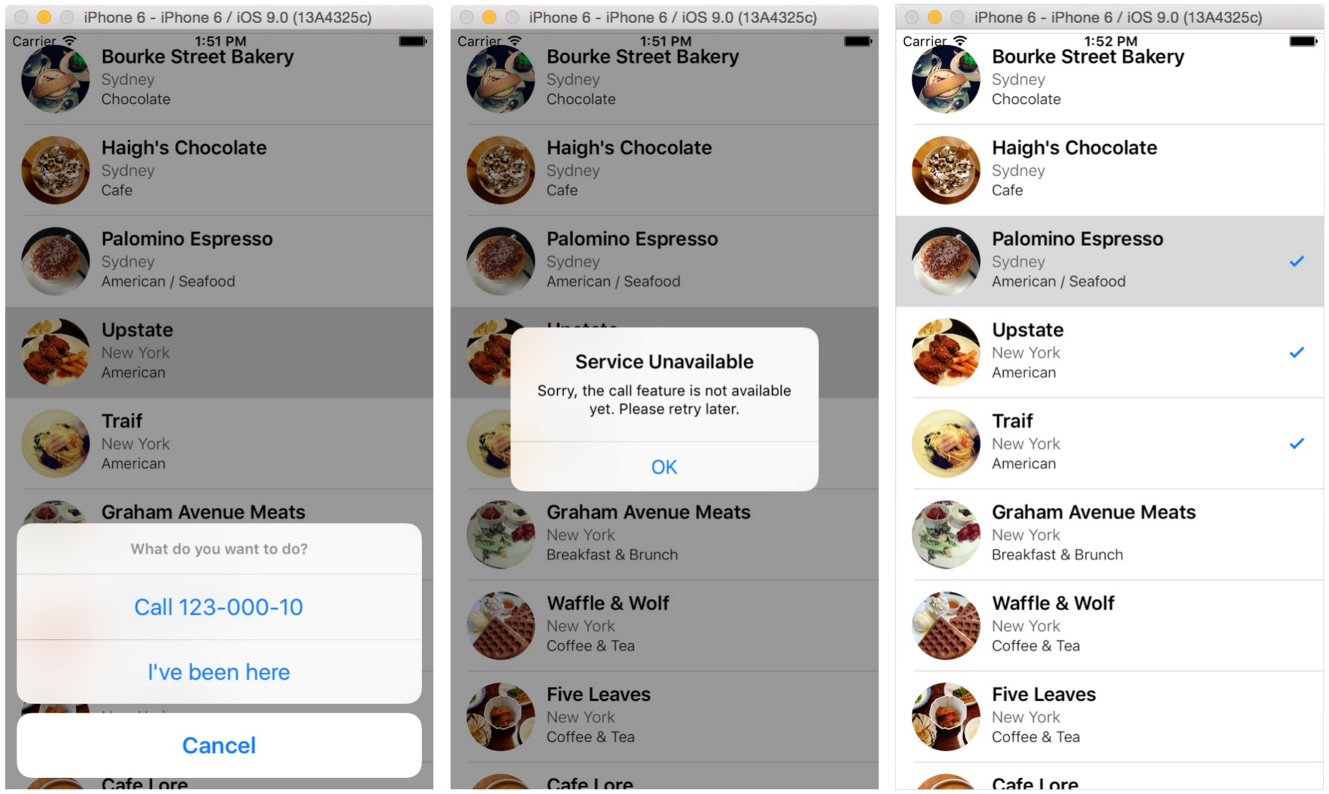
Optionals in Swift

```
You may wonder what the question mark is for. The cell is known as an optional
in Swift. Optional is a new type introduced in Swift. An optional simply means
'there is a value' or 'there isn't a value at all'. The cell returned by
tableView.cellForRowAtIndexPath is an optional. To access the accessoryType
property of the cell, you use the question mark. In this case, Swift will check
if the cell exists and allow you to set the value of accessoryType if the cell
exists. In most cases, the autocomplete feature of Xcode would add the question
mark for you when accessing a property of an optional. To learn more about
Optionals, you can further refer to the appendix.
```

When a user selects the "I've been here" option, we add a checkmark to the selected cell. For a table view cell, the right part is reserved for an accessory view. There are four types of built-in accessory views including *disclosure indicator*, *detail disclosure button*, *checkmark* and *detail*. In this case, we use *checkmark* as the indicator.

The first line of the code block retrieves the selected table cell using `indexPath`, which contains the index of the selected cell. The second line updates the `accessoryType` property of the cell with a check mark.

Compile and run the app. Tap a restaurant and choose one of the actions, it'll either show you a check mark or an alert.



Service Unavailable

Sorry, the call feature is not available yet. Please retry later.

Figure 10-8. Call action and I've been here action

For now, when you select a row, the row is highlighted in gray and stayed as selected. Add the following code at the end of the `tableView(_:didSelectRowAtIndexPath:)` method to deselect the row.

```
tableView.deselectRowAtIndexPath(indexPath, animated: false)
```

We Hit a Bug

The app looks great. If you look at it closely, however, there is a bug in the app. Say, you mark the 'Cafe Deadend' restaurant using the 'I've been here' action. If you scroll down the table, you'll find that another restaurant (e.g. Palomino Espresso) also contains a checkmark. What's the problem? Why did the app add an extra checkmark?

> Like every programmer, I hate bugs especially when facing a project deadline. However it's always the bugs that help me improve my programming skills. You'll hit a lot of bugs too as you continue to learn. Just get used to it.

The problem is due to cell reuse, that we have discussed in the previous chapter. For example, the table view has 30 cells. For performance reason, instead of creating 30 table cells, UITableView may just create 10 cells and reuses them as you scroll through the table. In this case, UITableView reuses the 1st cell (originally used for Cafe Deadend with a checkmark) for displaying another restaurant. In our code, we only update the image view and labels when the table view reuses the same cell. The accessory view is not updated. Thus, the next restaurant reusing the same cell shares the same accessory view. If the accessory view contains a checkmark, that restaurant will also carry a checkmark.

So how can we resolve the bug?

We have to find another way to keep track of the checked items. How about creating another array to save the checked restaurants? In the `RestaurantTableViewController.swift` file, declare a Boolean array:

```
var restaurantIsVisited = [Bool](count: 21, repeatedValue: false)
```

Bool is a data type in Swift that holds a Boolean value. Swift provides two Boolean values: *true* and *false*. We declare the `restaurantIsVisited` array to hold a collection of `Bool` values. Each value in the array indicates whether the corresponding restaurant is marked as 'I've been here'. For example, we can look at to the value of `restaurantIsVisited[0]` to see if Cafe Deadend is checked or not.

The values in the array are initialized to `false`. In other words, the items are unchecked by default. The above line of code shows you a way to initialize an array in Swift with repeated values. The initialization is the same as the following:

```
var restaurantIsVisited = [false, false, false, false, false, false, false,
false, false, false, false, false, false, false, false, false, false, false,
false, false, false]
```

We have to make a couple of changes in order to fix the bug. First, we need to update the value of the Bool array when a restaurant is checked. Add a line of code in the handler of the `isVisitedAction` object:

```
let isVisitedAction = UIAlertAction(title: "I've been here", style: .Default,
handler: {
     (action:UIAlertAction!) -> Void in

     let cell = tableView.cellForRowAtIndexPath(indexPath)
     cell?.accessoryType = .Checkmark
     self.restaurantIsVisited[indexPath.row] = true
})
```

The code is very straightforward. We update the value of the selected item from `false` to `true`. Lastly, add a few lines of code to update the accessory view in the `tableView(_:cellForRowAtIndexPath:)` method before `return cell`:

```
if restaurantIsVisited[indexPath.row] {
    cell.accessoryType = .Checkmark
} else {
    cell.accessoryType = .None
}
```

Here we check if the restaurant to be displayed is marked. If the condition is `true`, we display a checkmark in the cell. Otherwise, just display nothing. We now refer to the `restaurantIsVisited` array to see if the restaurant is marked. So even if the cell is reused, we can display the accessory view correctly.

Now, compile and run the app again. Your bug should now be resolved.

As a side note, you can further simplify the above if condition to a single line of code using the ternary conditional operator (?:):

```
cell.accessoryType = restaurantIsVisited[indexPath.row] ? .Checkmark : .None
```

The ternary conditional operator is an efficient shorthand for evaluating simple condition.

Your Exercise

Presently the app doesn't allow users to uncheck the checkmark. Think about how you can alter the code such that the app can toggle the checkmark. You'll also need to show a different title for the 'I've been there' button if the selected cell is marked. It's not too hard to make the changes.

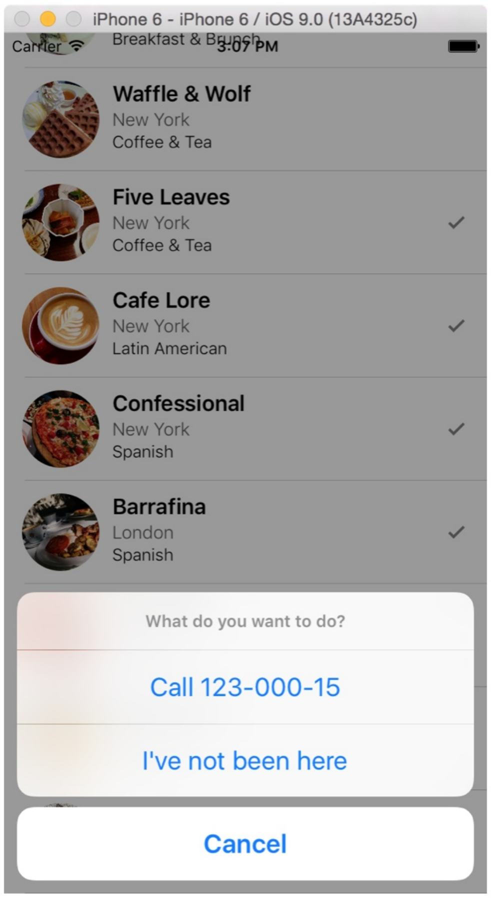


Figure 10-9. Deselecting a restaurant

Take some time to work on the exercise. I'm sure you'll learn a lot.

Summary

At this point, you should have a solid understanding about how to create table view, customize table cells and handle table row selection. You're ready to build a simple table view app on your own (e.g. a simple ToDo app). I always recommend you to create your own project. I don't mean you have to start a big one. If you love travel, create a simple app that displays a list of your favorite destination. If you love music, create your own app that shows a list of your favorite albums. Just play around with Xcode, make mistake and learn along the way.

For reference, you can download the complete Xcode project from https://www.dropbox.com/s/uf84e3bv3z9uatm/FoodPinTableSelection.zip?dl=0.

In the next chapter, we'll continue to explore table view and see how you can delete a table row.