CSCI 4100 Laboratory Five

Contact List

Introduction

In this laboratory you will construct a simple contact list application that illustrates the use of storyboards and table views. This application is shown in figure 1. It has the names and phone numbers of the computing science faculty.

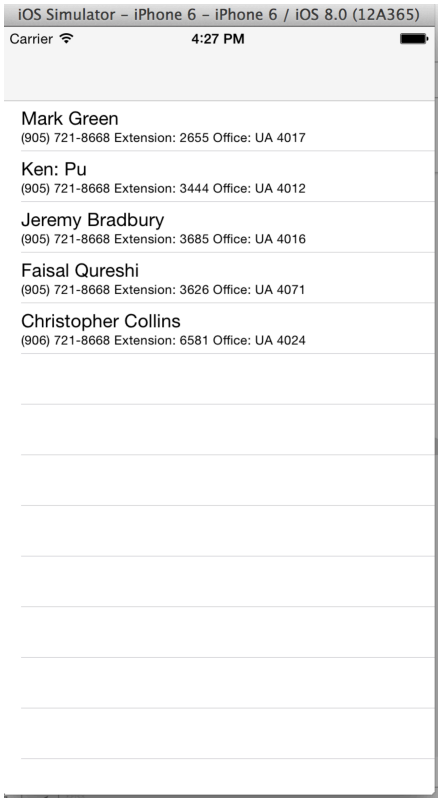

Figure 1 Contact List Application

Start by creating a project for this application. I called my project ContactList. You should use the Single View Application template when you create the project. Remember to select the Swift programming language.

Create the Data Model

Even though this application is relatively simple we will get into the good practice of creating data models for our applications. For this application the data model is quite simple. We will have an array of Contact objects, with each object holding the contact information for an individual faculty member.

Start by creating a new Swift class called Contact, this class should inherit from NSObject. This class will mainly contain data, so it will need a set of variables and an initialization method. We will need the following variables:

- firstName
- lastName
- phoneNumber
- Extension
- location

Declare these variables as class variables and they should all have the String type. Remember that the Swift syntax for variable declarations is different from the one that you are used to.

The initialization method for this class is:

```
    init(first : String, last: String, phone : String, ext :
String, loc: String)  {
        self.firstName = first;
        self.lastName = last;
        self.phoneNumber = phone;
        self.location = loc;
        self.Extension = ext;

    }
```

This completes the data model for this laboratory.

Create the User Interface

The next thing that we need to do is create the user interface. If you look at the storyboard you will see that Xcode has just created a standard view controller for us and not a table view controller. In addition, the ViewController class that Xcode automatically constructs for us inherits from UIViewController. We will need to change this before we start building the user interface. In the ViewController.swift file change UIViewController to UITableViewController.

In the storyboard delete the existing view controller (but don't delete the storyboard). If you deleted the storyboard by mistake you can create a new one in essentially the same way as you create a new class (make sure the name of the new storyboard is main, otherwise it won't be found at runtime). Now drag a table view controller onto the screen and make sure it is linked to the ViewController class. This should be the case by default, but if it's not correct it can be a hard to find bug.

Now we need to design the prototype cell for the table. In this case it is quite simple, we can use one of the standard cell types. The table view controller already has a prototype cell, which we can modify for our purposes. In the properties inspector change style from custom to Subtitle and set the identifier to Cell (see figure 2).

If we just use the storyboard the way it is the user interface won't look right, the table view will flow into the status area at the top of the screen. We can solve this problem by embedding the table view controller in a navigation controller. Start by selecting the table view controller and then select "Embed in" from the "Editor" menu. This will give you a list of choices; select "Navigation Controller" from this list. Make sure that the navigation controller is selected as the initial view controller (see figure 3). This completes the storyboard side of the user interface and we can now turn our attention to adding code to the table view controller.

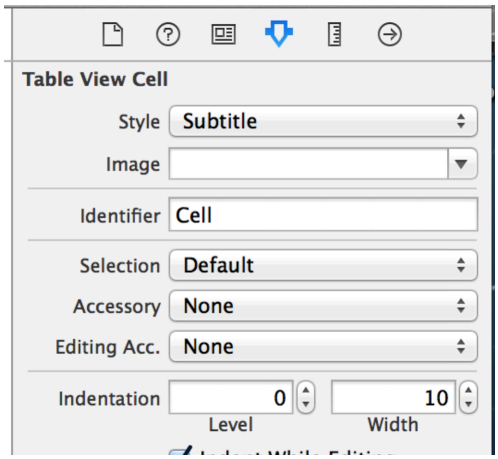

Figure 2 Configuration of Table View Cell

Table View Controller

Now we turn our attention to the ViewController class. To start with we need to initialize our data model. We can do this by creating an array of Contract objects called contacts in the following way:

```
     var contacts : [Contact] = [Contact(first: "Mark", last: "Green", phone: "(905) 721-
8668",ext: "2655", loc: "UA 4017"),
        Contact(first: "Ken:", last: "Pu", phone: "(905) 721-8668", ext: "3444", loc: "UA
4012"),
        Contact(first: "Jeremy", last: "Bradbury", phone: "(905) 721-8668", ext: "3685",
loc: "UA 4016"),
        Contact(first: "Faisal", last: "Qureshi", phone: "(905) 721-8668", ext: "3626",
loc: "UA 4071"),
        Contact(first: "Christopher", last: "Collins", phone: "(906) 721-8668", ext:
"6581", loc: "UA 4024")
    ];
```

This should be a class variable so it can be used by the methods within the class.

There are several functions that we need to implement for a table view controller. These functions tell the table view the number of rows in the table and the content of each row. The first of these functions is:

```
    override func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
        return contacts.count
    }
```

This function returns the number of rows in the table, which we can easily compute as contacts.count. This function looks a bit strange since we are interfacing with the Objective-C code that implements the table view. When these function names are converted to Swift they all override the tableView method, but each one of them as a different set of parameters. Note that the parameter name is considered to be part of the function name.

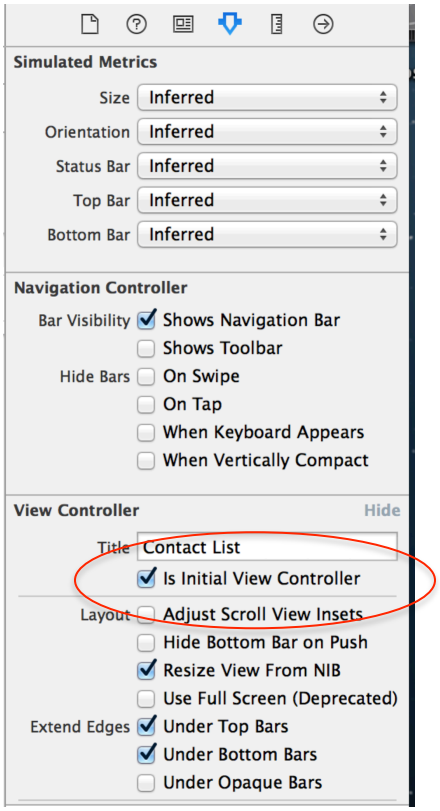

Figure 3 Sekect "Is Initial View Controller"

Now we need to create the table contents. The table view calls one of our procedures for each row in the table. This procedure constructs a cell that contains the on screen representation of the row. In our case this procedure is:

```
     override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("Cell") as UITableViewCell

        let content = contacts[indexPath.row]
        cell.textLabel?.text = "\(content.firstName) \(content.lastName)";
        cell.detailTextLabel?.text = "\(content.phoneNumber) Extension:
\(content.Extension) Office: \(content.location)";

        return cell
    }
```

The table view maintains a pool of cells so it is not constantly allocating and freeing storage as we scroll through the table. To get a new cell we call the

dequeueReusableCellWithIdentifier method in the tableView class. The parameter to this method is the identifier that was set when we created the table view cell in the storyboard (see figure 2).

One of the parameters to this function is indexPath, which contains the row for the table view cell. We can use this to index into our contacts array and store the result in the content constant. We are now ready to set the contents for the cell. The standard cell that we have chosen has two fields; a textLabel and a detailTextLabel. These labels are displayed as two rows with the textLabel on the top with a larger font. Both of the textLabels have a text field that contains the text to be displayed. But, there is an issue here, both the labels are optional values, so we need to add the "?" when we dereference them. This is a good example of the string formatting that we can do in Swift.

How did I know that the standard cell had two labels and the names of these labels? We can use the Xcode help system to determine this information. First, highlight the UITableViewCell somewhere in the code. Then on the pane on the right click the button that has a "?" inside a circle. The pane should change to the view shown in figure 4. Close to the bottom of this pane you will see a link for the reference page for this class. When you click this link you will get a new window that is similar to figure 5. Note that this page has both the Swift and Objective-C information for this class.

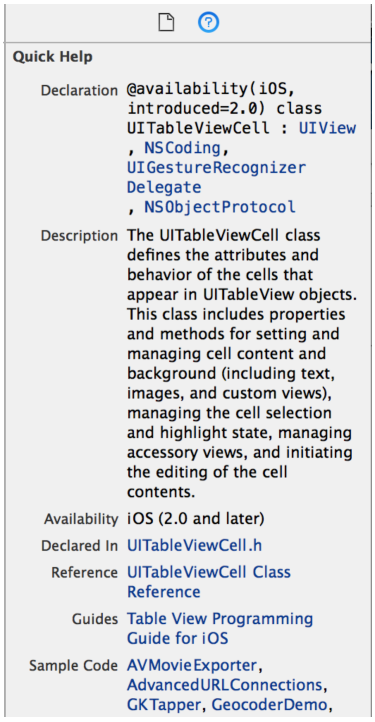

Figure 4 Quick Help for UITableViewCell

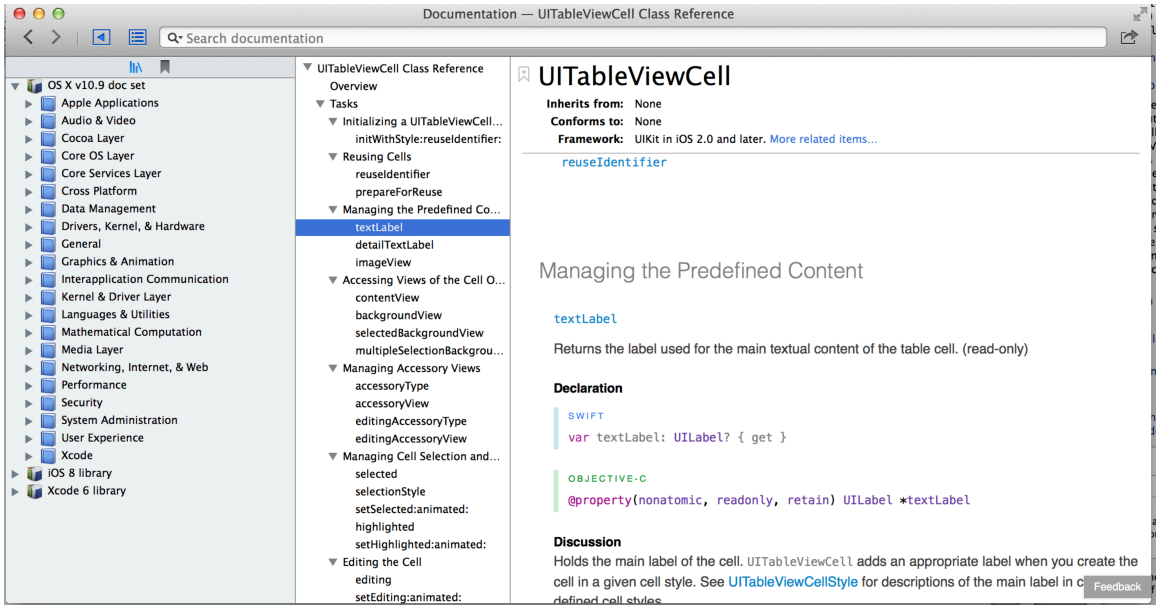

Figure 5 Reference page for UITableViewCell

Laboratory Report

To get things going quickly we used a standard table view cell, which doesn't look too bad. Improve the appearance of this application by replacing the standard cell with a cell of your own design. There are a few things you need to do to make this work, and we will walk through the first part of it. Start by making a copy of your project and modify the copy (just in case you do something wrong). Now change the type of the cell back to Custom from Subtitle. You might want to make the prototype cell large so you can fit multiple widgets into it. Now create a new Swift class, we will call it ContactsCell and it should be a subclass of UITableViewCell. Now select the prototype cell and in the identity inspector set the class to ContactsCell (you will probably need to type this). Next in the attributed inspector, make sure the identifier property is still Cell (Xcode will probably change this one you). Now drag a text field widget onto the prototype cell and place it somewhere near the top. You will probably need to resize it. Finally start the Assistant editor (click the button that looks like a guy with a bow tie in the upper right corner of the Xcode window) and make sure that the ContactsCell.swift file is shown in the right pane. Control drag from the text field widget to the ContactsCell class and create a new outlet called Name. You will need to repeat this process for all the widgets that you add to the prototype cell.

In the ViewController class change the second tableView function to:

```
    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("Cell") as ContactsCell

        let content = contacts[indexPath.row]
        cell.Name.text = content.lastName

        return cell
    }
```

There are basically two changes here. First, we cast cell to ContactsCell; our new table view cell class. Second, we use the Name outlet in this new class to display the last name of the faculty members.

Produce a nice looking prototype cell and show the results to your TA or email your code and a screen shot to the TA later.