ATLS 4320: Advanced Mobile Application Development

Week 3: Table Views

Table Views Overview

https://developer.apple.com/design/human-interface-guidelines/ios/views/tables/
Tables should be used to display large or small amounts of information in the form of a list. A table presents data as a scrolling, single-column list of rows. Additionally, In conjunction with navigation controllers they are used to navigate through hierarchical data.

Table views have two styles: *plain or grouped*

- The plain style can also have sections with an index along the right side
- The grouped style must have at least 1 group and each group must have at least 1 item
- Both styles can have a header and footer

The **UITableViewController** class is a view controller that manages a table views

The **UITableView** class displays data in a table view

The **UITableViewDelegate** protocol manages table row configuration and selection, row reordering, highlighting, accessory views, and editing operations.

The **UITableViewDataSource** protocol handles constructing tables and managing the data model when rows of a table are inserted, deleted, or reordered.

Table views can have an unlimited number of rows but only display only a few rows at a time.
In order to show data in the tables quickly, table views don't load all the rows, only the ones that need to be displayed at the time. Then as rows scroll off the screen they are automatically placed in a queue to be reused and dynamically refreshed with the appropriate data. Maximum efficiency and minimum work as a developer — thanks Apple!

The `dequeueReusableCell(withIdentifier: for:)` method dequeues an existing cell if one is available or creates a new one

- The reuse identifier is a string used to identify a cell that is reusable

Reusing cells is the best way to guarantee smooth Table View scrolling performance.

Table Rows
Each row in a table is a cell managed by the **UITableViewCell** class.
There are four standard table cell styles enum UITableViewCell.CellStyle or you can create a custom one.

1. **Basic**(default): a simple cell style that has a single title left aligned and an optional image. Good option for displaying items that don't require supplementary information. (UITableViewCellStyleDefault)

2. **Subtitle**: left-aligns the main title and puts a gray subtitle left aligned under it. It also permits an image in the default image location. This style works well in a table where rows are visually similar as the additional subtitle helps distinguish rows from one another. (UITableViewCellStyleSubtitle)
3. **Right Detail** (Value 1): left-aligns the main title with black text and right-aligns the subtitle with smaller blue text on the same line. (UITableViewCellStyleValue1)
    - Used in Settings
4. **Left Detail** (Value 2): puts the main title in blue and right-aligns it at a point that's indented from the left side of the row. The subtitle is left-aligned at a short distance to the right of this point. This style does not allow images. (UITableViewCellStyleValue2)
    - Used in contacts

Search

A search bar lets you search through a large collection of values by typing text into a field. A search bar can be displayed alone, or in a navigation bar or content view.
https://developer.apple.com/design/human-interface-guidelines/ios/bars/search-bars/

The **UISearchController** class incorporates a search bar, UISearchBar, into a view controller that has searchable content.

A search controller works with two custom view controllers that you provide. The first view controller is part of your app's main interface and displays your searchable content. The second view controller is created when you initialize a controller to display the search results. When the user interacts with a search bar, the search controller automatically displays a new view controller with the search results that you specify.

- In iOS11 and later you place the search bar in the navigation bar
- In iOS10 and earlier you place the search bar in the table's header view
- The searchResultsUpdater property is provided the search results from the search controller

The search results updater object must adopt the **UISearchResultsUpdating** protocol which has methods that update the search results as users enter data into the search bar. Its one method handles the search bar interaction and is required `updateSearchResults(for:)` It is automatically called whenever the search bar becomes the first responder or changes are made to the text in the search bar. Perform any required filtering and updating inside of this method.

scrabbleQ

File -> New Project
Single View App
iPhone
Name it scrabbleQ, make sure Storyboard is selected for User Interface and Swift is selected for language (no need for core data or tests, so uncheck those).
Save it somewhere nice

Go into Main.storyboard. The initial scene is a View Controller but we want a Table View Controller.
Click on the scene and **delete** it. Then **drag** onto the canvas a Table View Controller.
In the attributes inspector **check** Is Initial View Controller.

We want our class to be the controller so go into ViewController.swift and **change** its super class to `UITableViewController`.

Now go back into MainStoryboard and select the view controller and **change** its class to ViewController.
Select the table view and note that it has the class UITableView.
In the Connections inspector **check** that the dataSource and delegate for the table view are set to View Controller. If not, drag from the circles to the View Controller icon.
In the attributes inspector see that the table view's style is Plain. Try changing it to grouped and see how that looks.
Select a table view cell and in the attributes inspector you can see that the Table View Cell style is custom. We'll look at the others in a minute.
Select the Table View Cell and in the attributes inspector **make** the identifier "ScrabbleCell".

If you run it at this point you should see a blank table view.

Add `q-words-no-u.plist` from the class example repo into your project and *make sure Copy Items if Needed is checked* as well as your project target.

We'll create a class to load the data as we did for the Picker data and use PropertyListDecoder to decode the plist. We're doing this again because of our MVC design pattern.

File -> New -> File -> Swift file
DataController.swift

Create an array to hold the words and a constant for the filename.

```
class DataController {
    var qNoUWords: [String]?
    let filename = "q-words-no-u"
}
```

Add an enum with all the types of errors we may run into

```
enum DataError: Error {
    case NoDataFile
    case CouldNotDecode
    case NoWords
}
```

Then we'll **create** a method to load the plist data.

```
func loadWords() throws {
    //check if we have the file
```

```
    if let pathURL = Bundle.main.url(forResource: filename, withExtension:
"plist") {
        //plist decoder object
        let plistDecoder = PropertyListDecoder()

        do {
            //try to get the data and decode into array of strings
            let data = try Data(contentsOf: pathURL)
            qNoUWords = try plistDecoder.decode([String].self, from: data)
        } catch {
            print(error)
            throw DataError.CouldNotDecode
        }
    } else {
        //could not find file
        throw DataError.NoDataFile
    }
}
```

Add method to return the array of words.

```
//send all the words back in array of strings
func getWords() throws -> [String] {
    //check to make sure we have the words via conditional unwrapping
    if let words = qNoUWords {
        //got words!
        return words
    } else {
        //don't have any words :( throw error
        throw DataError.NoWords
    }
}
```

Now let's use this class and methods. Go into ViewController.swift and **add** an empty array that will hold the words and another variable to hold an instance of our DataController class

```
//hold our words
var words = [String]()
//connect to data controller
var data = DataController()
```

In viewDidLoad() we'll **call the methods** in the DataController class to get our words array populated.

```
override func viewDidLoad() {
    super.viewDidLoad()
    //make sure we load the data first
    do {
```

```
        try data.loadWords()
        words = try data.getWords()
    } catch {
        //should probably handle the error more robustly...
        print("could not load data")
    }
}
```

Now we **implement** the required methods for the UITableViewDataSource protocol

```
//customize the number of rows in each section (group) -- only one section in
this example
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return words.count
}

//displays the cells
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "ScrabbleCell", for:
indexPath)
    //we know there's a text label so we can safely set
    cell.textLabel?.text = words[indexPath.row]
    return cell
}
```

Your identifier string here MUST match the identifier you used in interface Builder. If they don't match, you will get an exception error: 'unable to dequeue a cell with identifier *WrongIdentifier* - must register a nib or a class for the identifier or connect a prototype cell in a storyboard'

You should now see your table view with the list of words.

Don't worry that the table scrolls under the status bar because table views are usually in navigation controllers and that will fix the problem.

Now let's **add** an image.
Drag scrabble_q_tile.png into Assets.xcassets
In Main.storyboard select the table view cell and **change** the style to basic and under image choose scrabble_q_tile.png.
Now when you run it you'll see the image.

Now **change** the Table View Cell style to subtitle.
Notice this adds a detail label.
Change its text to say "Q no U".
Now when you run it you'll see subtitles as well.

Now what if you want to do something when the user selects a row. It's a UITableViewDelegate method that handles this.

```
//Delegate Method
//alert the selected word when selected
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    let alert = UIAlertController(title: "Word selected", message: "You selected
\(words[indexPath.row])", preferredStyle: .alert)
    //create the action button
    let okAction = UIAlertAction(title: "OK", style: .default, handler: nil)
    alert.addAction(okAction) //add the button
    present(alert, animated: true, completion: nil)
    tableView.deselectRow(at: indexPath, animated: true) //make sure the row gets
deselected
}
```

Search

Let's add the ability to search our table view by creating a new view controller class to handle search and its results.
File -> New File
iOS -> Cocoa Touch class
SearchResultsController
Subclass UITableViewController
Leave Also create xib file unchecked
Save it in your project and target

Go into your new file and **adopt** the UISearchResultsUpdating protocol

```
class SearchResultsController: UITableViewController, UISearchResultsUpdating
```

(you'll get an error until we conform to the protocol)

Add variables
SearchResultsController needs access to the list of words that the main view controller is displaying, so we'll need an array to store those words as well as an array to store the results of a search.

```
//arrays for all the words and the ones matching the filter
var allWords = [String]()
var filteredWords = [String]()
```

The file already contains some template code that provides a partial implementation of the UITableViewDataSource protocol and some commented-out methods UITableViewController subclasses often need. We're not going to use most of them so you can delete them if you want.

Since we won't have a scene for this view controller in our storyboard we need to register our cell reuse identifier programmatically. We get a default tableView since we are subclassing

UITableViewController. Any customization of the table (changing style, etc.) would be done in `viewDidLoad()`. **Add to viewDidLoad()**

```
override func viewDidLoad() {
    super.viewDidLoad()

    //no storyboard scene so programmatically register the cell reuse identifier
    tableView.register(UITableViewCell.self, forCellReuseIdentifier:
"ScrabbleCell")
}
```

The `UISearchResultsUpdating` protocol only has 1 method and it's required. Let's **implement** it.

```
//implement the search
func updateSearchResults(for searchController: UISearchController) {
    let searchString = searchController.searchBar.text //get the text entered
into search bar
    filteredWords.removeAll() //empty array of filtered words

    //make sure we got a value
    if searchString?.isEmpty == false {

        //closure to filter through all words
        let searchFilter: (String) -> Bool = {word in
            let range = word.range(of: searchString!, options: .caseInsensitive)
            //range will be nil if the character sequence is no present in the
given word
            return range != nil
        }

        //use the closure to filter through all words
        filteredWords = allWords.filter(searchFilter)
    }
    //update the table with relevant words
    tableView.reloadData()
}
```

The Array filter method takes in a closure that takes an element of the sequence as its argument and returns a Boolean value indicating whether the element should be included in the returned array. *It does this for each item in the array.* We use the range method to see if a word contains the search string and if it does, the closure returns true. The filter method returns a new array of the words that matched the filter (returned true).

Then we need to implement the `UITableViewDataSource` methods to display the table view cells. We want the table rows to show the search results. Since SearchResultsController is a subclass of UITableViewController it automatically acts as the table's data source.

```
// MARK: - Table view data source
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return filteredWords.count
}

//delegate
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "ScrabbleCell", for:
indexPath)
    cell.textLabel?.text = filteredWords[indexPath.row]
    return cell
}
```

Note that this method is included with a return of 0. You either need to **change** it to return 1 section or **comment/delete** it as 1 is the default if the method isn't present.

```
override func numberOfSections(in tableView: UITableView) -> Int
    return 1
}
```

If you wanted to do something when the user selects a row from the search results, you would implement the `override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)` function to handle the action. This would probably match the action that happens when an item is selected from the original, all words tableview for the most predictable behaviour.

Now we have to **set up** ViewController.swift to add the search bar.
In ViewController.swift **add** an instance of UISearchController above `viewDidLoad()`

```
//add instance of search controller
var searchController = UISearchController()
```

Update `viewDidLoad()` to implement and configure the search bar.

```
override func viewDidLoad() {
    super.viewDidLoad()
    //make sure we load the data first
    do {
        try data.loadWords()
        words = try data.getWords()
        //create an instance of our custom results controller
        let resultsController = SearchResultsController()
        //set the all words property in instance based on loaded data
```

```
        resultsController.allWords = words
        //tell the search controller to use our
        searchController = UISearchController(searchResultsController:
resultsController)
        //add some placeholder text
        searchController.searchBar.placeholder = "Filter"
        searchController.searchBar.sizeToFit() //make it fit the parent view
        //add a header that consists of the search bar that belongs to our search
controller
        tableView.tableHeaderView = searchController.searchBar
        //tell it which object will be updating the results
        searchController.searchResultsUpdater = resultsController
    } catch {
        //should probably handle the error more robustly...
        print("could not load data")
    }
}
```

Each time the user types something into the search bar, `UISearchController` uses the object stored in its `searchResultsUpdater` property to update the search results. In our case this is the instance of our `SearchResultsController` — made possible by conforming to `UISearchResultsUpdating` and implementing `func updateSearchResults(for searchController: UISearchController)`

Now you should be able to search through the data in your table view which is very useful for tables with a lot of data or tables with dynamic data supplied by the user.

Grouped

Now let's look at the table view grouped style. We'll just modify our app a little bit to show how the grouping works. Quick and dirty.

First, we'll **add** a second plist that just has words containing the letter "q". (File->New->File->Property List)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
      <string>Quit</string>
      <string>Quite</string>
      <string>Aqua</string>
      <string>Quiver</string>
      <string>Quip</string>
      <string>Equal</string>
```

```
      <string>Equip</string>
</array>
</plist>
```

Second, we'll make a couple changes to our DataController.class to accommodate the new file.
Add a variable for the qWords

```
var qWords: [String]?
```

Change the filenames variable to an array

```
let filenames = ["q-words-no-u", "q-words"]
```

Create a for loop in the `loadWords()` method to handle the new file

```
for filename in filenames {
        if let pathURL = Bundle.main.url(forResource: filename, withExtension:
"plist") {
            //plist decoder object
            let plistDecoder = PropertyListDecoder()

            do {
                //try to get the data and decode into array of strings
                let data = try Data(contentsOf: pathURL)
                if filename == filenames[0] {
                    qNoUWords = try plistDecoder.decode([String].self, from:
data)
                } else if filename == filenames[1] {
                    qWords = try plistDecoder.decode([String].self, from: data)
                }
            } catch {
                print(error)
                throw DataError.CouldNotDecode
            }
        } else {
            //could not find file
            throw DataError.NoDataFile
        }
    }
```

Create a new method to get the qWords

```
func getQWords() throws -> [String] {
    //check to make sure we have the words via conditional unwrapping
    if let words = qWords {
        //got words!
        return words
    } else {
        //don't have any words :( throw error
```

```
            throw DataError.NoWords
        }
    }
```

Now we'll modify our ViewController to accommodate the new data into a grouped table view

Add a variable to hold the new property list data and some constants to store the indexes of each group.

```
//hold our words
var qNoUWords = [String]()
var qWords = [String]()
//constants
let qSection = 0
let qNoUSection = 1
```

Make necessary changes in viewDidLoad() to populate qWords array

```
override func viewDidLoad() {
    super.viewDidLoad()
    //make sure we load the data first
    do {
        try data.loadWords()
        qNoUWords = try data.getQNoUWords()
        qWords = try data.getQWords()
        //create an instance of our custom results controller
        let resultsController = SearchResultsController()
        //set the all words property in instance based on loaded data
        resultsController.allWords = qNoUWords + qWords
        //tell the search controller to use our
        searchController = UISearchController(searchResultsController:
resultsController)
        //add some placeholder text
        searchController.searchBar.placeholder = "Filter"
        searchController.searchBar.sizeToFit() //make it fit the parent view
        //add a header that consists of the search bar that belongs to our search
controller
        tableView.tableHeaderView = searchController.searchBar
        //tell it which object will be updating the results
        searchController.searchResultsUpdater = resultsController
    } catch {
        //should probably handle the error more robustly...
        print("could not load data")
    }
}
```

Change numberOfSections to 2 since we'll now have two sections

```
//how many sections do we want?
override func numberOfSections(in tableView: UITableView) -> Int {
    return 2
}
```

Change numberOfRowsInSection to be dependent on which group and the length of arrays

```
//customize the number of rows in each section (group)
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    //switch based on section parameter
    if section == qSection {
        return qWords.count
    } else {
        return qNoUWords.count
    }
}
```

Same with cellForRowAt

```
//displays the cells
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

    //get the section
    let section = indexPath.section

    var word: String
    //get word based on section
    if section == qSection {
        word = qWords[indexPath.row]
    } else {
        word = qNoUWords[indexPath.row]
    }

    let cell = tableView.dequeueReusableCell(withIdentifier: "ScrabbleCell", for:
indexPath)
    //we know there's a text label based on table style so we can safely set
    cell.textLabel?.text = word
    return cell
}
```

Also **change** didSelectRowAt

```
//alert the selected word when selected
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
```

```
    let section = indexPath.section

    var word: String
    //get word based on section
    if section == qSection {
        word = qWords[indexPath.row]
    } else {
        word = qNoUWords[indexPath.row]
    }

    let alert = UIAlertController(title: "Word selected", message: "You selected
\(word)", preferredStyle: .alert)
    //create the action button
    let okAction = UIAlertAction(title: "OK", style: .default, handler: nil)
    alert.addAction(okAction) //add the button
    present(alert, animated: true, completion: nil)
    tableView.deselectRow(at: indexPath, animated: true) //make sure the row gets
deselected
}
```

And **add** two methods to give each group an appropriate heading

```
override func tableView(_ tableView: UITableView, willDisplayHeaderView view:
UIView, forSection section: Int) {
    let headerview = view as! UITableViewHeaderFooterView
    headerview.textLabel?.font = UIFont(name: "Helvetica", size: 20)
    headerview.textLabel?.textAlignment = .center
}

override func tableView(_ tableView: UITableView, titleForHeaderInSection
section: Int) -> String? {
    //tableView.headerView(forSection: section)?.textLabel?.textAlignment =
NSTextAlignment.center
    if section == qSection {
        return "Words with Q"
    } else {
        return "Words with Q w/o U"
    }
}
```

Should be good to go! No worries!