Advanced Mobile Application Development
Week 3: Table Views

Table Views
https://developer.apple.com/ios/human-interface-guidelines/views/tables/
Tables should be used to display large or small amounts of information the form of a list. A table presents data as a scrolling, single-column list of rows.
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 view
https://developer.apple.com/reference/uikit/uitableviewcontroller

The **UITableView** class displays data in a table view
https://developer.apple.com/reference/uikit/uitableview

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 placed in a queue to be reused.
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.
https://developer.apple.com/reference/uikit/uitableviewcell
There are four standard table cell styles 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/ios/human-interface-guidelines/bars/search-bars/
The **UISearchController** class incorporates a search bar, UISearchBar, into a view controller that has searchable content https://developer.apple.com/documentation/uikit/uisearchcontroller
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 a table you can assign the search bar to the tableHeaderView property
- The searchResultsUpdater property is provided the search results from the search controller

The **UISearchResultsUpdating** protocol must be adopted
https://developer.apple.com/documentation/uikit/uisearchresultsupdating. Its one method handles the search bar interaction and is required updateSearchResults(for:).

scrabbleQ
File | New Project
Single View App
iPhone

Go into MainStoryboard. 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 "CellIdentifier" (note the capital I).

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

Add qwordswithoutu1.plist into your project and make sure Copy Items if Needed is checked as well as your project target.

Go into ViewController.swift and add an array that will hold the letters and words.
```
var words = [String]()
```

We'll get the URL to the plist as we did for the picker data and use PropertyListDecoder to decode the data and assign it to our words array.

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

        // URL for our plist
        if let pathURL = Bundle.main.url(forResource: "qwordswithoutu1",
withExtension: "plist"){
            //creates a property list decoder object
            let plistdecoder = PropertyListDecoder()
            do {
                let data = try Data(contentsOf: pathURL)
                //decodes the property list
                words = try plistdecoder.decode([String].self, from: data)
            } catch {
                // handle error
                print(error)
            }
        }
      }
```

Now we implement the required methods for the UITableViewDataSource protocol

```
//Required methods for UITableViewDataSource
// Customize the number of rows in the section
    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return words.count
    }

// Displays table view cells
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        //dequeues an existing cell if one is available, or creates a new
one and adds it to the table
        let cell = tableView.dequeueReusableCell(withIdentifier:
"CellIdentifier", for: indexPath)
        cell.textLabel?.text = words[indexPath.row]
        return cell
    }
```

Your identifier string here MUST match the identifier you used in interface Builder. 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 MainStoryboard select the table view cell and change the style to basic and under image choose scrabble_q_tile.png.
You can also assign the image programmatically

```
cell.imageView?.image=UIImage(named: "scrabbletile90.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.
Make its text say Q no U. (You can either add it in the label in the storyboard or do it programmatically)
```
cell.detailTextLabel?.text="Q no U"
```
Now when you run it you'll see a subtitle as well.

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

```
    //UITableViewDelegate method that is called when a row is selected
    override func tableView(_ tableView: UITableView, didSelectRowAt 
indexPath: IndexPath) {
        let alert = UIAlertController(title: "Row selected", message: "You 
selected \(words[indexPath.row])", preferredStyle: .alert)
        let okAction = UIAlertAction(title: "OK", style: .default, handler: 
nil)
        alert.addAction(okAction)
        present(alert, animated: true, completion: nil)
        tableView.deselectRow(at: indexPath, animated: true) //deselects the 
row that had been choosen
    }
```

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)
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.

```
    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. Add to viewDidLoad()

```
//register our table cell identifier
tableView.register(UITableViewCell.self, forCellReuseIdentifier:
"CellIdentifier")
```

The `UISearchResultsUpdating` protocol only has 1 method and it's required

```
    //UISearchResultsUpdating protocol required method to implement the
search
    func updateSearchResults(for searchController: UISearchController) {
        let searchString = searchController.searchBar.text //search string
        filteredWords.removeAll(keepingCapacity: true) //removes all
elements
        if searchString?.isEmpty == false {
            //closure that will be called for each word to see if it matches
the search string
            let searchfilter: (String) -> Bool = { name in
                //look for the search string as a substring of the word
                let range = name.range(of: searchString!,
options: .caseInsensitive)
                return range != nil //returns true if the value matches and
false if there's no match
            } //end closure
            let matches = allwords.filter(searchfilter)
            filteredWords.append(contentsOf: matches)
        }
        tableView.reloadData() //reload table data with search results
    }
```

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. 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.

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

    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
"CellIdentifier", 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
    }
```

Now we have to set up our main ViewController to add the search bar.
In ViewController.swift add an instance of UISearchController

```
    var searchController : UISearchController!
```

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

```
        //search results
        let resultsController = SearchResultsController() //create an
instance of our SearchResultsController class
        resultsController.allwords = words //set the allwords property to
our words array
        searchController = UISearchController(searchResultsController:
resultsController) //initialize our search controller with the
resultsController when it has search results to display

        //search bar configuration
        searchController.searchBar.placeholder = "Enter a search term"
//place holder text
        searchController.searchBar.sizeToFit() //sets appropriate size for
the search bar
        tableView.tableHeaderView=searchController.searchBar //install the
search bar as the table header
        searchController.searchResultsUpdater = resultsController //sets the
instance to update search results
```

Each time the user types something into the search bar, UISearchController uses the object stored in its searchResultsUpdater property to update the search results.

Now you should be able to search through the data in your table view. Very useful for tables with a lot of data.

Grouped
Now let's look at the table view grouped style. (scrabbleQgrouped)
This uses qwordswithoutu2.plist which is a dictionary with keys that are letters and values that are an array of words. The keys are used to group the table and create an index.
In the storyboard the only difference is in the attributes inspector for the table view the style is Grouped.

In ViewController.swift there's a Dictionary for all the words and an Array for the letters

In viewDidLoad we load the plist into the Dictionary and then extract the keys into the letters array and sort them.
The Array class has a method called sort that sorts the array and assigns it back to itself. It takes a closure for how to do the sort. This is the shorthand way to sort Strings in ascending order. For more complex sorts you can write a function.

```
    override func viewDidLoad() {
        super.viewDidLoad()
        // URL for our plist
        if let pathURL = Bundle.main.url(forResource: "qwordswithoutu2",
withExtension: "plist"){
            //creates a property list decoder object
            let plistdecoder = PropertyListDecoder()
            do {
                let data = try Data(contentsOf: pathURL)
                //decodes the property list
                allwords = try plistdecoder.decode([String:[String]].self,
from: data)
                letters = Array(allwords.keys)
                // sorts the array
                letters.sort(by: {$0 < $1})
            } catch {
                // handle error
                print(error)
            }
        }

        //search results
        let resultsController = SearchResultsController() //create an
instance of our SearchResultsController class
        resultsController.allwords = allwords
        resultsController.letters = letters
        searchController = UISearchController(searchResultsController:
resultsController) //create a search controller and initialize with our
SearchResultsController instance

        //search bar configuration
        searchController.searchBar.placeholder = "Enter a search term"
//place holder text
     tableView.tableHeaderView=searchController.searchBar //install the
search bar as the table header
        searchController.searchResultsUpdater = resultsController //sets the
instance to update search results
    }
```

The delegate methods now all need to take into account the section.

Need to calculate the number of words for a given section.

```
    override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        let letter = letters[section]
        let letterSection = allwords[letter]!
        return letterSection.count
    }
```

Need to get the section before the word.

```
    override func tableView(_ tableView: UITableView, cellForRowAt
indexPath: IndexPath) -> UITableViewCell {
        let section = indexPath.section
        let letter = letters[section]
        let wordsSection = allwords[letter]!
        //configure the cell
        let cell = tableView.dequeueReusableCell(withIdentifier:
"CellIdentifier", for: indexPath)
        cell.textLabel?.text = wordsSection[indexPath.row]
        return cell
    }

    override func tableView(_ tableView: UITableView, didSelectRowAt
indexPath: IndexPath) {
        let section = indexPath.section
        let letter = letters[section]
        let wordsSection = allwords[letter]!
        let alert = UIAlertController(title: "Row selected", message: "You
selected \(wordsSection[indexPath.row])", preferredStyle: .alert)
        let okAction = UIAlertAction(title: "OK", style: .default, handler:
nil)
        alert.addAction(okAction)
        present(alert, animated: true, completion: nil)
        tableView.deselectRow(at: indexPath, animated: true) //deselects the
row that had been choosen
    }
```

Need to display a section header

```
    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
    }
```

The number of sections is not 1 so we have to implement these UITableViewDatasource methods.

Returns the number of sections. (The default is 1 so we didn't need to implement it earlier)

```
    override func numberOfSections(in tableView: UITableView) -> Int {
        return letters.count
    }
```

Sets the header value for each section.

```
    override func tableView(_ tableView: UITableView,
titleForHeaderInSection section: Int) -> String? {
        //tableView.headerView(forSection:
section)?.textLabel?.textAlignment = NSTextAlignment.center
        return letters[section]
    }
```

Adds a section index

```
    override func sectionIndexTitles(for tableView: UITableView) ->
[String]? {
        return letters
    }
```

An optional method that allows you to configure a custom view for the section headers.

```
    override func tableView(_ tableView: UITableView, viewForHeaderInSection
section: Int) -> UIView? {
        let headerview = UITableViewHeaderFooterView()
        var myView:UIImageView
        if section == 0 {
            myView = UIImageView(frame: CGRect(x: 10, y: 8, width: 40,
height: 40))
        } else {
            myView = UIImageView(frame: CGRect(x: 10, y: -10, width: 40,
height: 40))
        }
        let myImage = UIImage(named: "scrabbletile90")
        myView.image = myImage
        headerview.addSubview(myView)
        return headerview
    }
```

We only have 4 different sections so it doesn't look great, but this is especially helpful for really long lists.