Lecture 7:
Table Views

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest
Content

- UITableView
- Creating Table View MVCs
- UITableViewDataSource
- UITableViewDelegate
UITableView

Very important class for displaying data in a table

- One-dimensional table.
- It’s a subclass of UIScrollView.
- Table can be a static or dynamic list of items.
- Lots and lots of customization via a dataSource protocol and a delegate protocol.
- Very efficient even with very large sets of data.
UITableView

Displaying multi-dimensional tables

- Usually done via a `UINavigationController` containing multiple MVC’s where View is `UITableView`.
- Or, via the `UICollectionView` from iOS 6.0. Collection views provide the same general function as table views, except that a collection view is able to support more layouts.
- Collection views support customizable layouts that can be used to implement multi-column grids, circular layouts, and many more.

Kinds of `UITableViews`

- Plain or Grouped.
- Static or Dynamic.
- Divided into sections or not.
- Different formats for each row in the table (including completely customized).
UITableView

UITableViewStylePlain

UITableViewStyleGrouped

- Airplane Mode
- Wi-Fi
- Bluetooth
- Mobile Data
- VPN
- Notifications
- Control Centre
- Do Not Disturb
- General
- Display & Brightness
- Wallpaper
Plain Style

Table Header

<table>
<thead>
<tr>
<th>Header of Section 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
</tr>
<tr>
<td>Row 1</td>
</tr>
<tr>
<td>Row 2</td>
</tr>
<tr>
<td>Footer 0</td>
</tr>
<tr>
<td>Header of Section 1</td>
</tr>
<tr>
<td>Row 0</td>
</tr>
<tr>
<td>Row 1</td>
</tr>
<tr>
<td>Footer 1</td>
</tr>
</tbody>
</table>

Table Footer
UITableView

Grouped Style

Table Header

Header of Section 0

Row 0
Row 1
Row 2

Footer 0

Header of Section 1

Row 0
Row 1

Footer 1

Table Footer
<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barcelona</td>
<td>Spain</td>
</tr>
<tr>
<td>Bucharest</td>
<td>Romania</td>
</tr>
<tr>
<td>Istanbul</td>
<td>Turkey</td>
</tr>
<tr>
<td>Madrid</td>
<td>Spain</td>
</tr>
<tr>
<td>Mersin</td>
<td>Turkey</td>
</tr>
<tr>
<td>Milano</td>
<td>Italy</td>
</tr>
<tr>
<td>Nice</td>
<td>France</td>
</tr>
<tr>
<td>Paris</td>
<td>France</td>
</tr>
<tr>
<td>Toulouse</td>
<td>France</td>
</tr>
<tr>
<td>Italy</td>
<td></td>
</tr>
<tr>
<td>Milano</td>
<td>Italy</td>
</tr>
<tr>
<td>Rome</td>
<td>Italy</td>
</tr>
<tr>
<td>Romania</td>
<td></td>
</tr>
<tr>
<td>Bucharest</td>
<td>Romania</td>
</tr>
<tr>
<td>Spain</td>
<td></td>
</tr>
<tr>
<td>Barcelona</td>
<td>Spain</td>
</tr>
<tr>
<td>Madrid</td>
<td>Spain</td>
</tr>
<tr>
<td>Turkey</td>
<td></td>
</tr>
</tbody>
</table>
Creating Table View MVCs

`UITableViewController` is the iOS class used as the base class for MVC’s that display `UITableViews`.

Just drag out a `UITableViewController` in Interface Builder.
Creating Table View MVCs

`UITableViewController` is the iOS class used as the base class for MVC's that display `UITableViews`. 
Creating Table View MVCs

Choose “New File ...” from the File menu to create a custom subclass of UITableViewController.
Creating Table View MVCs

Choose “New File ...” from the File menu to create a custom subclass of `UITableViewController`.

Be sure to set the superclass to `UITableViewController`.
Creating Table View MVCs

Choose “New File ...” from the File menu to create a custom subclass of `UITableViewController`.

Be sure to set it in your storyboard too!
Creating Table View MVCs

You can customize both the look of the table view and its cells from Interface Builder.

Click on the table view (not the table view controller) to see its properties in the Inspector.
Creating Table View MVCs

You can customize both the look of the table view and its cells from Interface Builder.

We will talk about the prototype for a Dynamic Table View in a moment, but for now, switch to a Static Table View.
Creating Table View MVCs

You can customize both the look of the table view and its cells from Interface Builder.

This table view has the Plain style. Let's change it to Grouped.
Creating Table View MVCs

And you can change the look of each cell as well.

Click on a cell that you want to change and set its attributes in the Inspector.
Creating Table View MVCs

And you can change the look of each cell as well.

Change Style to Basic, Right Detail, Left Detail, Subtitle and notice the cell layout each time.
Creating Table View MVCs

And you can change the look of each cell as well.

Double-click to edit.
Creating Table View MVCs

And you can change the look of each cell as well.

Show disclosure accessory. This should be on whenever clicking on a row in the table brings up another MVC.
Creating Table View MVCs

And you can change the look of each cell as well.

Show checkmark accessory. This can be used to show multiple selection in the table (requires some other API use).
Creating Table View MVCs

And you can change the look of each cell as well.

Show detail disclosure accessory. This is an active control. Use it to show auxiliary info. Clicking on the row should still do the “main thing” for this row.
Creating Table View MVCs

User taps on the blue detail disclosure below?

This will be sent to your `UITableViewController`:

```swift
func tableView(_ tableView: UITableView, accessoryButtonTappedForRowWith indexPath: IndexPath)
```
Creating Table View MVCs

Notice that some cell styles can have an image. You can set this in the code as well (more in a moment on this).

Let's set an image.
Creating Table View MVCs

Notice that some cell styles can have an image. You can set this in the code as well (more in a moment on this).
Creating Table View MVCs

In the Custom style, you can drag out views and wire them up as outlets!
Creating Table View MVCs

In the Custom style, you can drag out views and wire them up as outlets!
Creating Table View MVCs

In the Custom style, you can drag out views and wire them up as outlets!

CTRL + drag to associate an action to this button.
Creating Table View MVCs

In the Custom style, you can drag out views and wire them up as outlets!

```swift
class MyTableViewController: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Uncomment the following line to press
        // self.clearsSelectionOnViewWillAppear

        // Uncomment the following line to display
        // self.navigationItem.rightBarButtonItem
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be
    }

    // MARK: - Target Actions

    // MARK: - Table view data source

    /*
    override func numberOfSections(in tableView
        #warning Incomplete implementation,
        return 0
    }*

    override func tableView(_ tableView: UITableView,
        #warning Incomplete implementation,
        return 0
    }

    * override func tableView(_ tableView: UITableView,
        let cell = tableView.dequeueReusableCell(withIdentifier:,
        // Configure the cell...
    }
```
Creating Table View MVCs

In the Custom style, you can drag out views and wire them up as outlets!

class MyTableViewController: UITableViewDataSource, UITableViewDelegate {

   override func viewDidLoad() {
      super.viewDidLoad()

      // Uncomment the following line to prevent
      // self.clearsSelectionOnViewWillAppear

      // Uncomment the following line to disp
      // self.navigationItem.rightBarButtonItem

   }

   override func didReceiveMemoryWarning() {
      super.didReceiveMemoryWarning()

      // Dispose of any resources that can be
   }

   // MARK: - Target Actions

   @IBAction func makeThingsHappenNow() {
      print("We make things happen")
   }

   // MARK: - Table view data source

   /*
    * override func numberOfSections(in tableView
    *     #warning Incomplete implementation, return 0
    *
    * override func tableView(_ tableView: UITableView,
    *     #warning Incomplete implementation, return 0
    *
    */
Creating Table View MVCs

Move the entry point arrow to the table view controller, and let's run the application.

Tap the detail disclosure accessory and notice the log printed on the console.
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

Switch to a Dynamic Table with Prototype Cells.
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

Now click on the Prototype to edit it. All cells in this table will be like this Prototype (though we’ll set the contents to be different in code).
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

Now click on the Prototype to edit it. All cells in this table will be like this Prototype (though we’ll set the contents to be different in code).

You should see Table View Cell properties appear in the Attributes Inspector.
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

Let’s change the Prototype’s style to be Subtitle, for example.
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

The reuse identifier is a very important field! It is the name that we will reference in our code to identify this prototype (more on this in a moment).
Creating Table View MVCs

All of the above examples were “static” cells (setup in the storyboard). If you switch to dynamic mode, then the cell you edit is a “prototype” for all cells in the list.

Pick a name that is meaningful. “My Table View Cell” would probably not be that great. Something like “Photo Description” (if this were a list of photos) would be better.
UITableView Protocols

How do we connect to all this stuff in our code?

- A UITableView has two important properties: its delegate and its dataSource.
- The delegate is used to control how the table is displayed.
- The dataSource provides the data that is displayed inside the cells.
- Your UITableViewController is automatically set as the UITableView’s delegate and dataSource.
- Your UITableViewController subclass will also have a property that points to the UITableView:

```swift
var tableView: UITableView! { get set }
```
UITableView Protocols

- To be “dynamic”, we need to be the UITableView’s dataSource.

- Three important methods in this protocol:
  1. How many sections in the table?
  2. How many rows in each section?
  3. Give me a UIView to use to draw each cell at a given row in a given section.

- Let’s cover the last one first.
UITableViewDataSource

How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of UITableViewCell.
- Here is the UITableViewDataSource method to get that cell for a given row in a given section.

```swift
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell
{
return cell
}
```

In a static table, you do not need to implement this method (though you can if you want to ignore what’s in the storyboard).
How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

```swift
override func tableView(_: tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    return cell
}
```

`IndexPath` is just an object with two important properties for use with `UITableView`: row and section.
How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

```swift
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // get a cell to use (instance of UITableViewCell)

    // set properties on the cell to prepare it to display

    return cell
}
```
How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

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

    return cell
}
```

This MUST match what is in your storyboard if you want to use the prototype you defined there!
**UITableViewDataSource**

How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

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

The cells in the table are actually reused. When one goes off-screen, it gets put into a “reuse pool”. The next time a cell is needed, one is grabbed from the reuse pool if available. If none is available, one will be put into the reuse pool if there’s a prototype in the storyboard. Otherwise the dequeue method might return `nil` or generate an `Error`.
UITableViewDataSource

How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

```swift
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
        for: indexPath)

    // set properties on the cell to prepare it to display

    return cell
}
```
How do we control what is drawn in each cell in a dynamic table?

- Each row is drawn by its own instance of `UITableViewCell`.
- Here is the `UITableViewDataSource` method to get that cell for a given row in a given section.

```swift
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
    cell.textLabel?.text = self.data(forRow: indexPath.row, inSection: indexPath.section)
    return cell
}
```

There are obviously other things you can do in the cell besides setting its text (detail text, image, accessory, etc).
UITableViewController

How does a dynamic table know how many rows are there?

- And how many sections, too, of course?

```swift
func numberOfSections(in tableView: UITableView) -> Int
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
```

- Number of sections is 1 by default. In other words, if you don’t implement `numberOfSections(in:)`, it will be 1.
- No default for number of rows in a section.
- This is a required method in this protocol (as is `tableView(_:cellForRowAt:)`).

What about a static table?

- Do not implement these `dataSource` methods for a static table.
- `UITableViewController` will take care of that for you.
There are a number of other methods in this protocol

- But we’re not going to cover all of them.
- They are mostly about getting the headers and footers for sections.
- And about dealing with editing the table (moving/deleting/inserting rows).
There are a number of other methods in this protocol

- Let us continue with our demo and see, for example, how can we delete rows from a Table View.
- We implement the following method to return `true` to allow editing.

```swift
override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
    /* Return false if you do not want the specified item to be editable. */
    return true
}
```
There are a number of other methods in this protocol

- Let us continue with our demo and see, for example, how can we delete rows from a Table View.
- We delete the row from the Table View and also from the Model by implementing this method:

```swift
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath)
{
    if editingStyle == .delete
    {
        // Delete the row from the data source
        let deletedData = self.deleteData(atRow: indexPath.row, inSection: indexPath.section)
        print("We removed : \(deletedData)")

        tableView.deleteRows(at: [indexPath], with: .fade)
    }
}
```
All of the above was the `UITableView`'s `dataSource`. But `UITableView` has another protocol-driven delegate called its `delegate`.

The `delegate` controls how the `UITableView` is displayed. Not what it displays (that’s the `dataSource`’s job).

It is common for `dataSource` and `delegate` to be the same object. Usually the Controller of the MVC in which the `UITableView` is part of the (or is the entire) View.

The `delegate` also lets you observe what the table view is doing. Especially responding to when the user selects a row.

We often will use segues when this happens, but we can also track it directly.
UITableViewDelegate method sent when row is selected

- This is sort of like table view “target/action”.
- You might use this to update a detail view in a split view if master is a table view

```swift
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)
{
    /* Go do something based on information about our data structure corresponding to indexPath.row in indexPath.section */

    // We usually want to deselect the row once you are done
    tableView.deselectRow(at: indexPath, animated: true)
}
```
Table View “Target/Action”

Lots and lots of other delegate methods

- **will/did** methods for both selecting and deselecting rows.
- Providing **UIView** objects to draw section headers and footers.
- Handling editing rows (moving them around with touch gestures).
- **willBegin/didEnd** notifications for editing.
- Copying/pasting rows.
Table View Segues

You can segue when a row is touched, just like from a button. Segues will call `prepare(for:sender:)` with the chosen `UITableViewCell` as sender.

CTRL+drag from your cell to another View Controller to create a segue in Storyboard.
Table View Segues

You can segue when a row is touched, just like from a button. Segues will call `prepare(for:sender:)` with the chosen `UITableViewCell` as sender.

Chose your segue type. In a Navigation Controller, you might want to create a Push segue.
Table View Segues

You can segue when a row is touched, just like from a button. Segues will call `prepare(for:sender:)` with the chosen `UITableViewCell` as sender.

You might want to choose a segue identifier to figure out which segue is triggered in your code!
Table View Segues

You might also want to embed your Table View Controller in a Navigation Controller.
Table View Segues

You might also want to embed your Table View Controller in a Navigation Controller.
Table View Segues

- You can tailor whatever data the MVC needs to whichever cell was selected.
- This works whether dynamic or static.

```swift
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if sender is UITableViewCell {
        let cell = sender as! UITableViewCell
        let indexPath = self.tableView indexPath(for: cell)
        /* Prepare segue.destinationController to display
           * based on information about my data structure
           * corresponding to indexPath.row
           * in indexPath.section */
    }
}
```
What if your Model changes?

- You can:
  
  ```swift
  func reloadData()
  ```

- Causes the table view to call `numberOfSections(in:)` and `tableView(numberOfRowsInSection:)` all over again and then `tableView(cellForRowAt:)` on each visible cell.

- Relatively heavy weight obviously, but if your entire data structure changes, that’s what you need.

- If only part of your Model changes, there are lighter-weight reloaders, for example:
  
  ```swift
  func reloadRows(at indexPaths: [IndexPath],
                  with animation: UITableViewRowAnimation)
  ```
There are dozens of other methods in UITableView

- Setting headers and footers for the entire table.
- Controlling the look (separator style and color, default row height, etc).
- Getting cell information (cell for index path, index path for cell, visible cells, etc). Scrolling to a row.
- Selection management (allows multiple selection, getting the selected row, etc).
- Moving, inserting and deleting rows, etc.
Next Time

View Controller Lifecycle and UIKit:

- View Controller Lifecycle
- Image View
- Web View
- Scroll View