iOS

Additional elements
Table views

• To add a table to an iOS app, we use an instance of UITableView
• It is aUIScrollView subclass, itself a subclass of UIView, so it can either be a full-screen view itself or embedded as a child of another view
• It cannot have arbitrary subviews added to it, as it uses its subviews to present the individual cells within the table view
Two properties

- The most important is the dataSource, which is an object that implements the UITableViewDataSource protocol
  - This protocol defines methods that tell the table how many sections it has (and optionally what their titles are) and how many rows are in a given section
  - The data source also has editing methods that allow for the addition, deletion, or reordering of table contents
- There is also a delegate, an object implementing the UITableViewDelegate protocol
  - This provides method definitions for handling selection of rows and other user-interface events
Two approaches

• The only responsibility of a table is presenting the data and tracking user gestures like scrolling and selection
  — Other objects, often a view controller, are in charge of the rest

• Typically, there are two approaches
  — A UIViewController that implements the UITableViewDataSource and UITableViewDelegate protocols
    • If the table is just a subview then we need to use this
  — A UITableViewController, a subclass of the UIViewController, that also implements the UITableViewDataSource and UITableViewDelegate protocols
    • Useful when the only view presented by the controller is a table
import UIKit

class ViewController: UITableViewController {

View Controller
Providing a data source

- The table in the storyboard does not know how to use our class
  - It expects to create a generic UITableViewController for the table, but we want to use our ViewController
Trivial implementation

• The data source needs to provide a minimum of three things
  — Number of sections
  — Number of rows in a given section
  — A cell for each section and row

• We must implement UITableViewDataSource methods
  — numberOfSectionsInTableView()
  — tableView(numberOfRowsInSection:)
  — tableView(cellForRowAtIndexPath:)
  — And the optional tableView(titleForHeaderInSection:) to see section breaks
import UIKit

class ViewController: UITableViewController {

    override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
        return 5
    }

    override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
        return "Section \(section)"
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return section + 1
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = UITableViewCell(style: UITableViewCellStyle.Default, reuseIdentifier: nil)
        cell.textLabel?.text = "Row \(indexPath.row)"
        return cell
    }
}
First result

- iOS 7 says that view controllers default into full-screen mode
- The idea is that once we start scrolling and content goes under the status bar, the transparency of the status bar gives us a visual cue about information that is about to come
Filling the table

• Since the table can demand the content of any row at any time, we need a data structure that represents the information we want to show: an array

• One approach would be to just create a class for the information we care about

class ParsedInfo: NSObject {
    var infoText : String?
    var userName : String?
    var createdAt: String?
    var userAvatarURL : NSURL?
}
import UIKit

class ParsedInfo: NSObject {

    var infoText : String?
    var userName : String?
    var createdAt : String?
    var userAvatarURL : NSURL?

    init (infoText: String?,
          userName: String?,
          createdAt: String?,
          userAvatarURL : NSURL?) {

        super.init()
        self.infoText = infoText;
        self.userName = userName;
        self.createdAt = createdAt;
        self.userAvatarURL = userAvatarURL;
    }

    override init () {
        super.init()
    }
}

Some fake data

import UIKit

let defaultAvatarURL = NSURL(string: "https://abs.twimg.com/sticky/default_profile_images/" + "default_profile_6_200x200.png")

class ViewController: UITableViewController {

    var parsedInfos : [ParsedInfo] = [
        ParsedInfo(infoText:"First message", userName:"@lbaresi", createdAt:"2015-01-13 16:44:30 EDT", userAvatarURL: defaultAvatarURL),

        ParsedInfo(infoText:"Second message", userName:"@mbaresi", createdAt:"2015-01-14 16:44:30 EDT", userAvatarURL: defaultAvatarURL),

        ParsedInfo(infoText:"Third message", userName:"@nbaresi", createdAt:"2015-01-12 16:44:30 EDT", userAvatarURL: defaultAvatarURL)
    ]
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return 1
}

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

override func tableView (_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = UITableViewCell(style: UITableViewCellStyle.Default, reuseIdentifier: nil)
    let parsedInfo = parsedInfos[indexPath.row]
    cell.textLabel?.text = parsedInfo.infoText
    return cell
}
So how do we refresh contents?

- The array doesn’t have a way to tell the table that its contents have changed
  - The controller must keep the view and model in sync
- UITableViewController offers methods to notify the table of distinct edits
  - insertRowsAtIndexPaths() or removeRowsAtIndexPaths()
  - We can do a full reload of the table with reloadData()

```swift
func reloadInfos() {
    self.tableView.reloadData()
}
```
Table Cell styles

- When we create the UITableViewCell in tableView, our init method takes a style argument
- Four cell styles are defined in enum UITableViewCellStyle
  - we’ve been using UITableViewCellStyle.Default
- The cell class itself defines certain subviews:.textLabel, detailTextLabel, imageView, and accessoryView
  - The style determines if and where those subviews are laid out
  - if an imageView is set, it appears on the left side of the cell, and an accessoryView (usually a “show details” type button) appears on the right
Reuse

• The idea is to create a new UITableViewCell in every call to tableView
  — A cell may only appear for an instant before it goes off the screen and is no longer needed
  — Creating views is fairly expensive, so if we can avoid doing that frequently, it will make our app faster and more responsive
• UITableView provides a method dequeueReusableCellWithIdentifier
  — The idea is to create a cell in the storyboard as a sort of template and identify it with a known string
  — The code will ask for a cell by this name
Custom Table Cells

• We need to create a subclass of UITableViewCell.
override func tableView (_ tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("CustomInfoCell") as! ParsedInfoCell
    let parsedInfo = parsedInfos[indexPath.row]
    cell.userNameLabel.text = parsedInfo.userName
    cell.infoTextLabel.text = parsedInfo.infoText
    cell.createdAtLabel.text = parsedInfo.createdAt

    if parsedInfo.userAvatarURL != nil {
        if let imageData = NSData(contentsOfURL: parsedInfo.userAvatarURL) {
            cell.avatarImageView.image = UIImage(data: imageData)
        }
    }
    return cell
}
Content view controller

• Presents content on the screen using a view or a group of views organized into a view hierarchy
  – Each controller is responsible for managing all the views in a single view hierarchy
  – A single controller should never manage multiple screens
UITableViewController

- A built-in controller designed for managing tabular data
  - Manages a table view and adds support for many standard table-related behaviors
    - A table view presents data in a single-column list of multiple rows and is a means for displaying and editing hierarchical lists of information
    - Has a pointer to the root view of the interface, but it also has a separate pointer to the table view
Table view styles

- It is just an attribute we can set on the table view in the storyboard
UICollectionView

- Represents a view controller whose content consists of a collection view
  - Displays an ordered collection of data
- Similar to a table view displays data using a combination of cell, layout, and supplementary views
  - can display items in a grid or in a custom layout that you design
  - Each cell must be an instance of UICollectionViewCell
UINavigationController

- Presents data organized hierarchically
- Provides methods for managing a stack-based collection of content view controllers
UITabBarController

- Used to divide your app into distinct modes of operation
- The tab bar has multiple tabs, each represented by a child view controller
UITabBarController

• What if we had more than 4 View Controllers?
  — A More button appears
• Everything happens automatically
UIPageViewController

• Implements a page layout
  – One can flip between discrete pages of content as if it were a book
  – Each content page is provided by a content view controller
UI.SplitContainer

- It presents a master-detail interface
  - Changes in the primary view controller (the master) drive changes in a secondary view controller (the detail)
The ContactsUI Framework

The Contacts UI framework provides controllers that facilitate displaying, editing, selecting, and creating contacts in your app.

Classes

- UIViewController
  - The UIViewController class provides the infrastructure for managing the views of your iOS apps.
- CNContactPickerViewController
  - The CNContactPickerViewController class creates a controller object that manages the contacts picker view.
- CNContactViewController
  - The CNContactViewController class implements the view to display a contact.
- ABUnknownPersonViewController
  - The ABUnknownPersonViewController class prompts the user to complete a partial person record, optionally allowing them to add it to the address book.
More scenes
## Segue types

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface Builder Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content in the detail or master area depending on the content of the screen. If the app is displaying a master and detail view, the content is pushed onto the detail area. If the app is only displaying the master or the detail, the content is pushed on top of the current view controller stack.</td>
</tr>
<tr>
<td>Show Detail</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content in the detail area. If the app is displaying a master and detail view, the new content replaces the current detail. If the app is only displaying the master or the detail, the content replaces the top of the current view controller stack.</td>
</tr>
<tr>
<td>Present Modally</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content modally. There are options to choose a presentation style (UITraditionalPresentationStyle) and a transition style (UIModalTransitionStyle).</td>
</tr>
<tr>
<td>Present as Popover</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content as a popover anchored to an existing view. There is an option to specify the possible directions of the arrow shown on one edge of the popover view (UIPopoverArrowDirection). There is also an option to specify the anchor view.</td>
</tr>
<tr>
<td>Custom</td>
<td><img src="image" alt="Icon" /></td>
<td>A custom segue enabling you to write your own behaviors.</td>
</tr>
<tr>
<td>Push (Deprecated)</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content by pushing it onto the current stack of view controllers.</td>
</tr>
<tr>
<td>Modal (Deprecated)</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content modally on top of the existing screen. The options are the same as Present Modally.</td>
</tr>
<tr>
<td>Popover (Deprecated)</td>
<td><img src="image" alt="Icon" /></td>
<td>Present the content as a popover. The options are the same as Present as Popover.</td>
</tr>
<tr>
<td>Replace (Deprecated)</td>
<td><img src="image" alt="Icon" /></td>
<td>Replace the top view controller on the screen with the new content.</td>
</tr>
</tbody>
</table>
**Show** - Pushes the destination view controller onto the navigation stack, moving the source view controller out of the way (destination slides overtrop from right to left), providing a back button to navigate back to the source - on all devices

*Example:* Navigating inboxes/folders in Mail

**Show Detail** - Replaces the detail/secondary view controller when in a `UISplitViewController` with no ability to navigate back to the previous view controller

*Example:* In Mail on iPad in landscape, tapping an email in the sidebar replaces the view controller on the right to show the new email

**Present Modally** - Presents a view controller in various different ways as defined by the Presentation option, covering up the previous view controller - most commonly used to present a view controller that animates up from the bottom and covers the entire screen on iPhone, but on iPad it's common to present it in a centered box format overtrop that darkens the underlying view controller

*Example:* Tapping the + button in Calendar on iPhone

**Popover Presentation** - When run on iPad, the destination appears in a small popover, and tapping anywhere outside of this popover will dismiss it. On iPhone, popovers are supported as well but by default if it performs a Popover Presentation segue, it will present the destination view controller modally over the full screen.

*Example:* Tapping the + button in Calendar on iPad (or iPhone, realizing it is converted to a full screen presentation as opposed to an actual popover)
Unwind segue

- Can be used to navigate back through segues
  - Choose the view controller that should appear onscreen at the end of an unwind segue
  - Define an unwind action method on the view controller you chose
    - `@IBAction func myUnwindAction(unwindSegue: UIStoryboardSegue)`
  - Navigate to the view controller that initiates the unwind action
  - Control-click the button (or other object) that should initiate the unwind segue
    - This element should be in the view controller you want to dismiss
  - Drag to the Exit object at the top of the view controller scene
  - Select your unwind action method from the relationship panel
UIGestureRecognizer

- We can get notified of the raw touch events
  - Or we can react to certain, predefined “gestures”
- Gestures are recognized by the class UIGestureRecognizer
  - This class is “abstract”
  - TapGestureRecognizer, UIPinchGestureRecognizer, 
    UIRotationGestureRecognizer, UISwipeGestureRecognizer, 
    UIPanGestureRecognizer, UIScreenEdgePanGestureRecognizer, 
    UILongPressGestureRecognizer
- There are two sides to using a gesture recognizer
  - Adding a gesture recognizer to a UIView to ask it to recognize that gesture
  - Providing the implementation of a method to “handle” that gesture when it happens
Internationalization

- The ability of code to adapt to local conventions in different parts of the world
  - This includes things like language, time and date formatting, and currency symbols and separators
- We must create a localization for each locale we want to support
  - A localization is a collection of strings, currency formats, graphics, sounds, and other resources that are specific to one locale
  - We declare supported localizations at the project level
Deployment Target

iOS Deployment Target: 8.1

Configurations

- Name: Debug
  - Based on Configuration File: No Configurations Set
- Name: Release
  - Based on Configuration File: No Configurations Set

Use Release for command-line builds

Localizations

<table>
<thead>
<tr>
<th>Language</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>Development Language</td>
</tr>
<tr>
<td></td>
<td>2 Files Localized</td>
</tr>
</tbody>
</table>

- French (fr)
- German (de)
- Chinese (Simplified) (zh-Hans)
- Chinese (Traditional) (zh-Hant)
- Japanese (ja)
- Spanish (es)
- Spanish (Mexico) (es-MX)
- Italian (it)
- Dutch (nl)
- Korean (ko)
- Portuguese (pt)
.strings files
Our code

```swift
@IBAction func increment(sender: UIButton) {
    var text = NSLocalizedString("Value: ", comment: "english")
    counterLabel.text = text + String(counter++)
}
```
Localizable.strings

```swift
/* Localizable.strings
TestApp

Created by Luciano Baresi on 1/6/15.
Copyright (c) 2015 Luciano Baresi. All rights reserved.
*/

"Value: " = "Valeur: ";
```