COMP327 Mobile Computing

Session: 2012-2013

Lecture Set 3a - Views, Delegates and Tables

In these Slides...

- **We will cover...**
  - View Contexts
  - View Delegates
  - Displaying lists within Table Views
  - Extending Table Views
  - Customising Cells

Different Views

These slides will allow you to use advanced view paradigms and interface elements that require delegate that adhere to a protocol to make them work.

UNIVERSITY OF LIVERPOOL

Application Contexts

Lecture Set 3a - Views, Delegates and Tables

Application Contexts

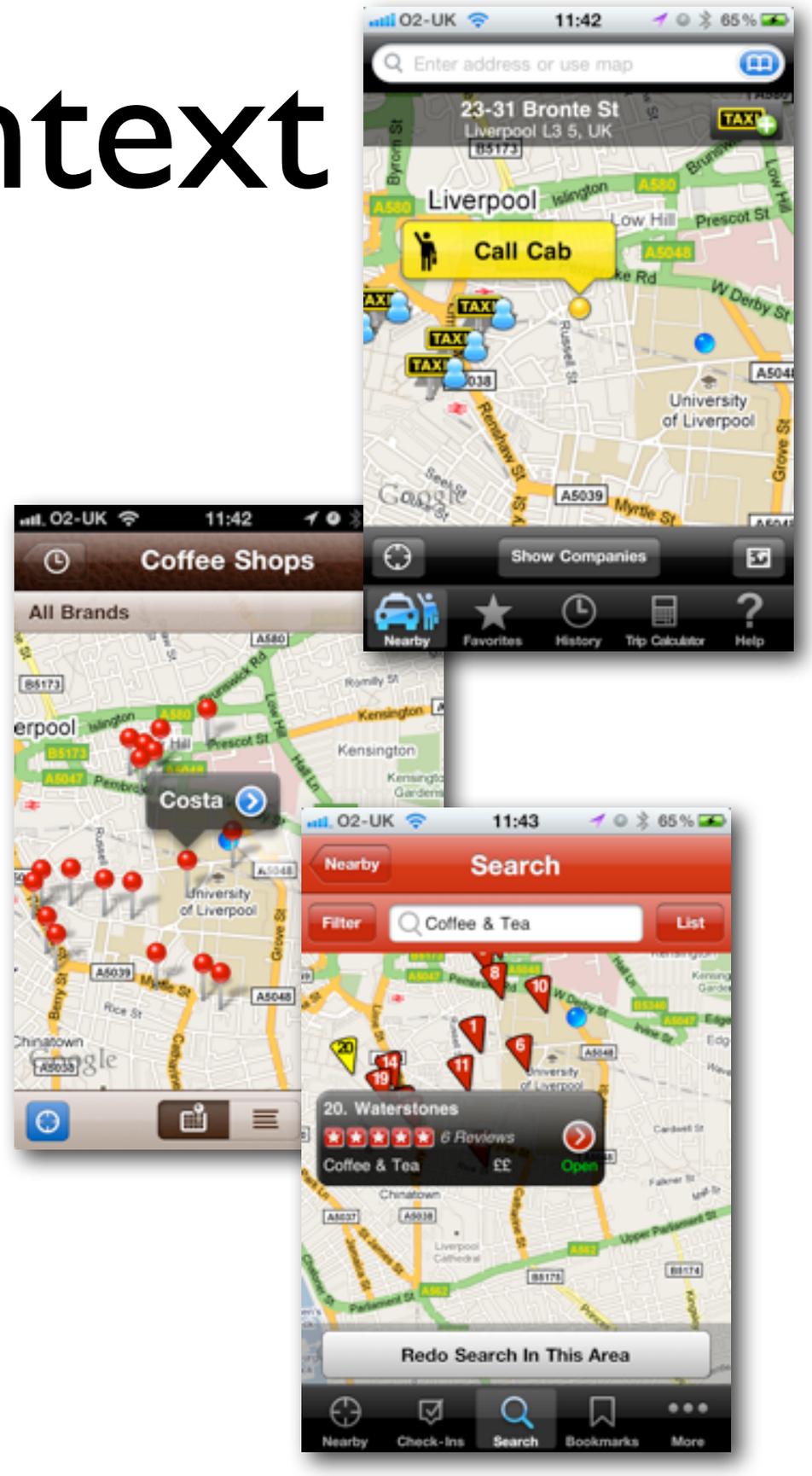
- Applications can be presented in a variety of ways, depending on the:
  - Intended user
  - Task itself
- Desktop Applications often attempt to combine different contexts within a single application
  - This can be confusing or difficult to use for the mobile user.
- Various contexts exist, including:
  - Utility Context
  - Locale Context
  - Informative Context
  - Productivity Context
  - Immersive Context

Utility Context

- The most basic context
  - Short, simple task-based scenarios
    - e.g. calculator, weather forecast, unit conversion, stocks, etc
  - The user provides simple (minimal information)
    - Either by fetching relevant information from a server, or performing a simple calculation, the relevant information is presented to the user.
- The goal
  - provide “at a glance” information
  - focus on presenting only the relevant information unambiguously
  - avoid clutter, use minimal design aesthetic, with focus on the content in view, often with large fonts and sparse layout.

Locale Context

- Emerging due to location-based services
  - Provide information given the user's position,
    - tourist guides, location-based social networks, finder apps
- Normally, a map is presented with the location of items plotted.
- Ordering is based on proximity
- Location of the user is important, and should be emphasised

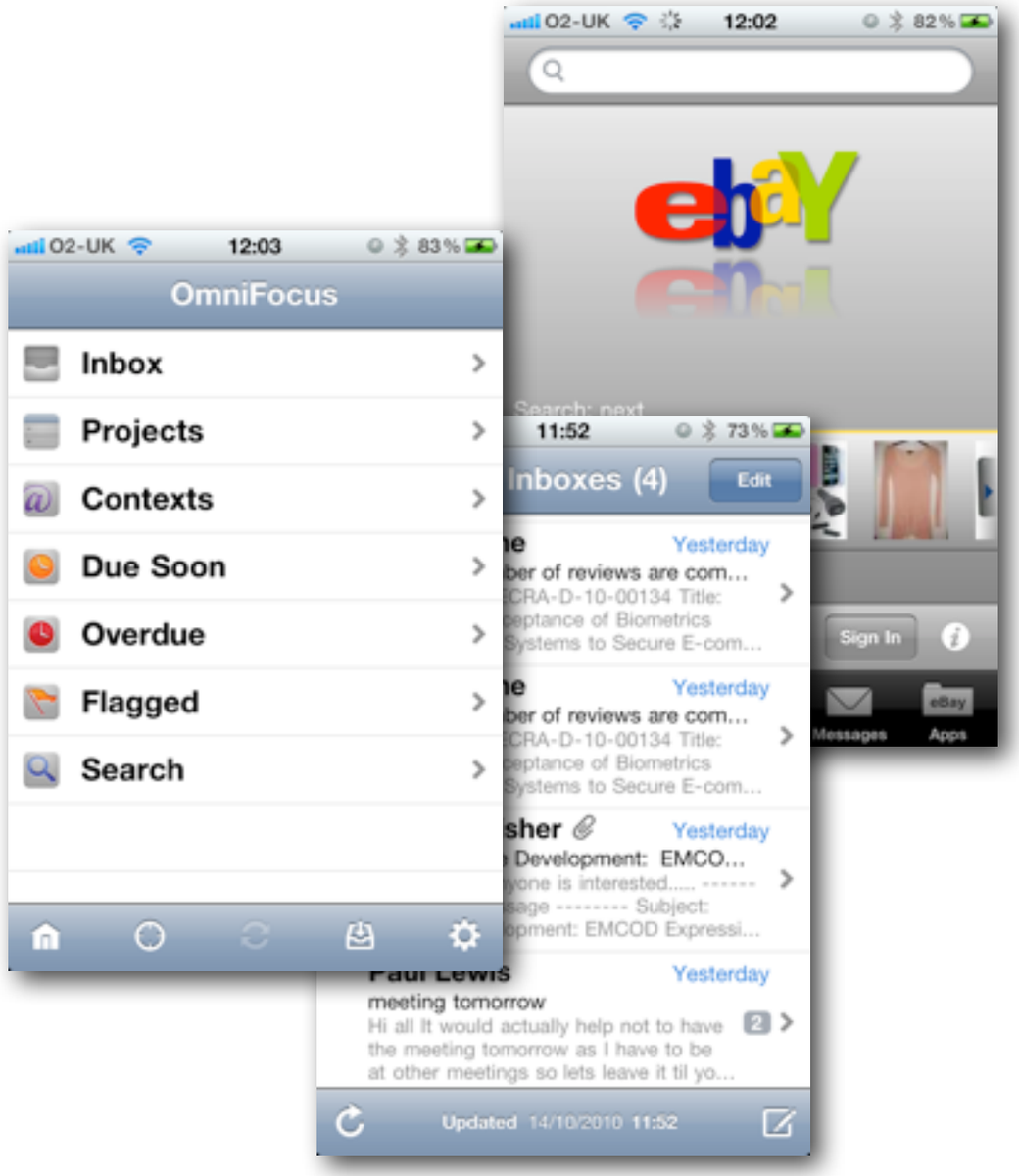


Informative context

- Goal is to provide information, such as a news site
  - User wants to read or browse, but not necessarily interact
  - Used during brief idle times
    - Such as waiting for public transport
  - Need to provide the option to bookmark or save story to read later, or flag favourites
- Avoid having the user enter complex or lengthy data

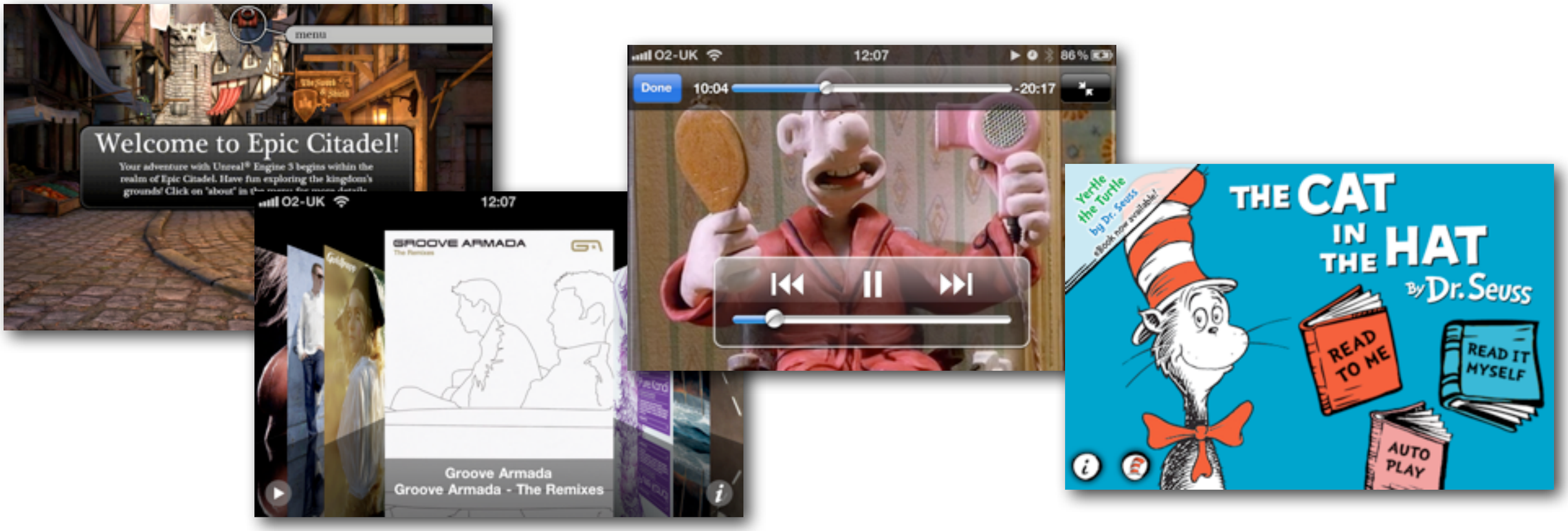
Productivity Context

- Assist with heavier, task-based services
  - User is focussed on performing a task, rather than glancing or consuming information
  - Similar to using a tool on a desktop
    - e.g. managing email, contacts, plans etc
  - Information often presented in a group or hierarchy format
    - should think how the user thinks about information
      - order and priority of tasks, etc
    - e.g. mail apps focus on inbox as a top priority item
      - yet sending email is still important



Immersive Context

- Immersive, full screen applications
  - Media Players, Games, etc
- Consume the users focus, by filling the entire screen
  - Often includes augmented reality applications



UNIVERSITY OF LIVERPOOL

Organising Data on the display's window

Lecture Set 3a - Views, Delegates and Tables

Organising your Content

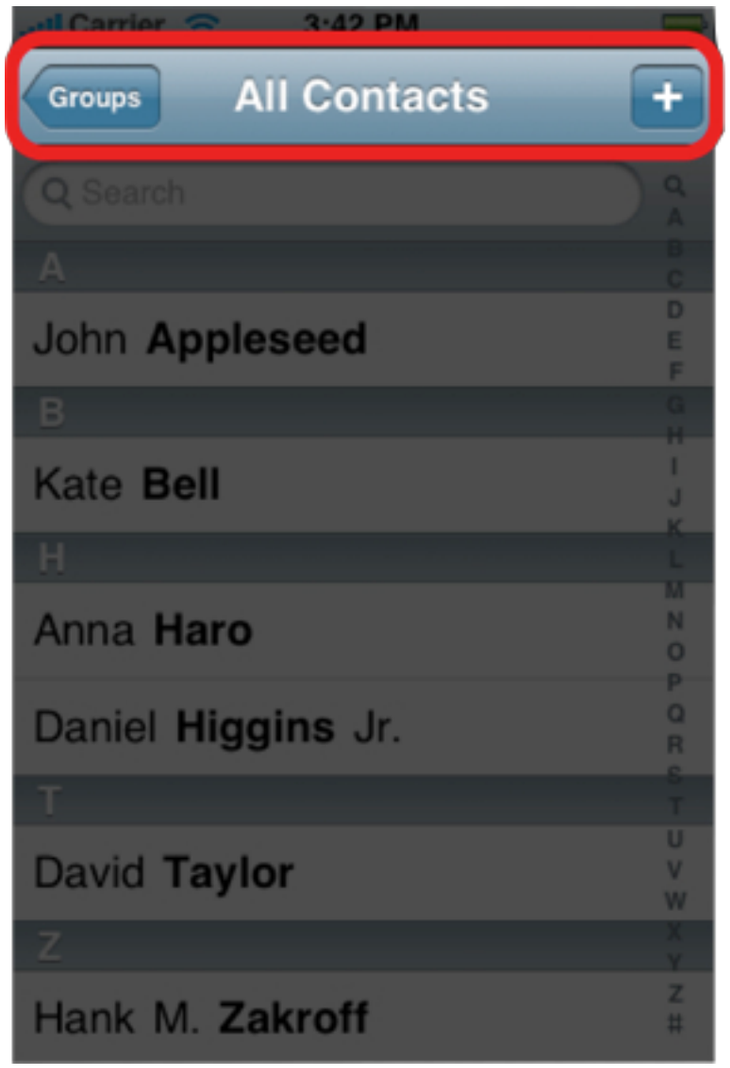
- Often applications can present lots of data

- Aim to deliver
  - Single screens of content
  - One thing at a time

- Focus on your user's content

Patterns for Organising Content

Navigation Bar



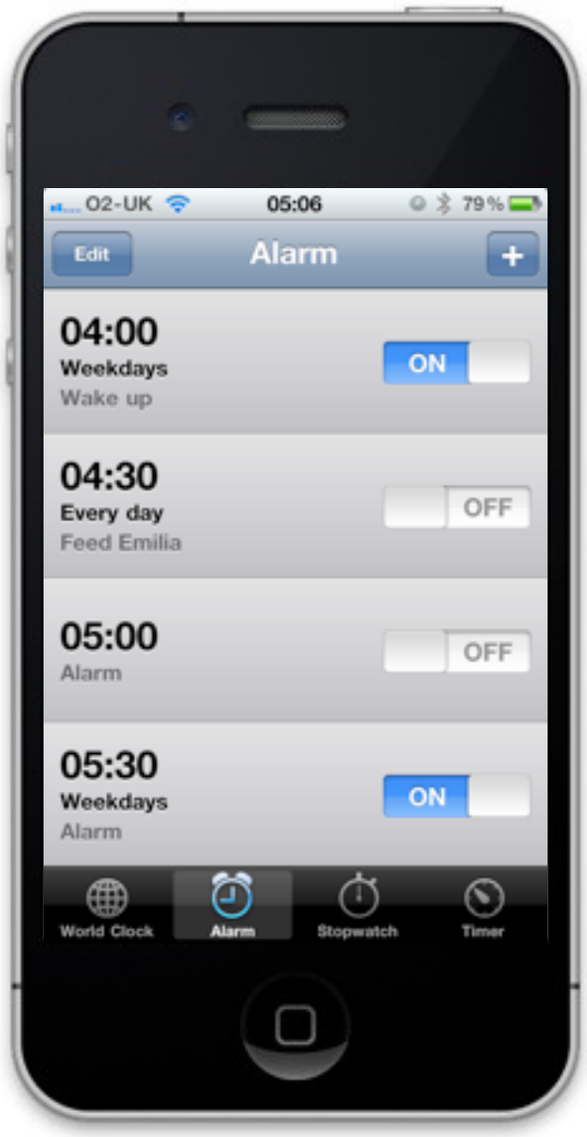
Tab Bar

Navigation Controller

- Often used when there is a hierarchy of content
  - E.g. module browser or Email client
- User needs to drill down into greater detail
  - Also need to manage where in the hierarchy they are
    - Provide cues in the top navigation bar
  - Allow them to navigate back up the content
    - Provide a “back” button

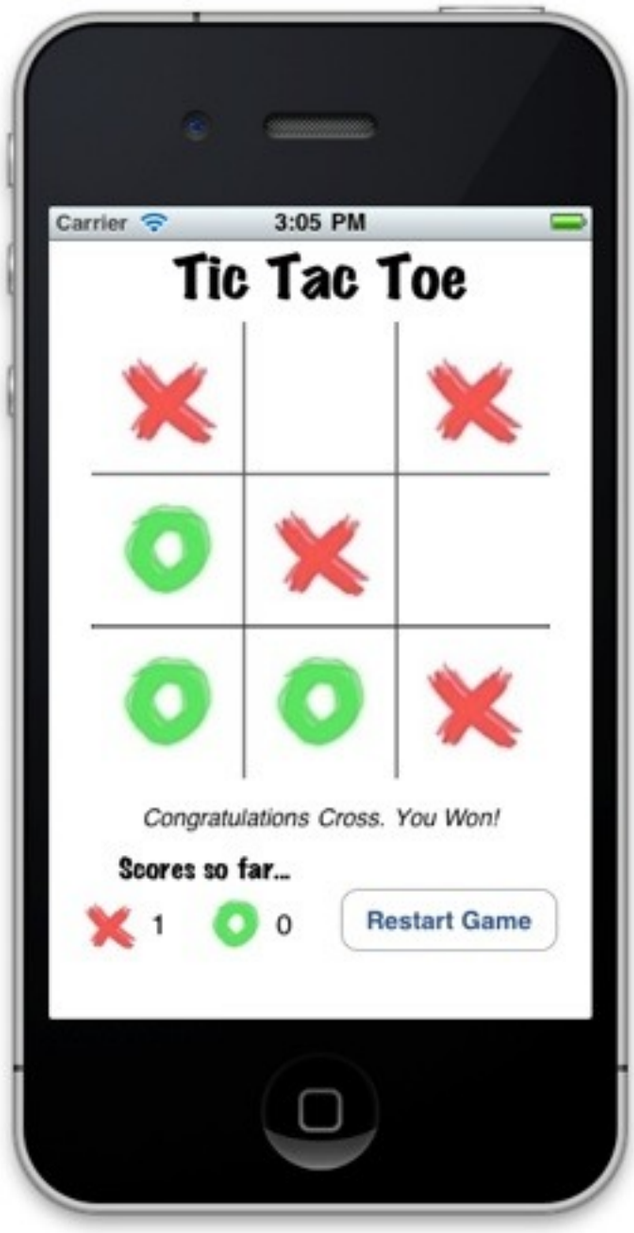
Tab Bar

- Often used when a single application has many self contained modes or views
  - User may want to switch between different functionality, whilst in the same view
- Tab bar allows the user to select one mode at a time



Problem: Managing a Screenful

- The controller manages the views, your data, and the application logic
  - Apps are full of these
  - Can provide encapsulated functionality for a view or set of views
    - e.g. Tic Tac Toe (opposite)
- App can be built by plugging individual views together
  - Some flows are very common
    - Navigation-based
    - Tab-Bar based



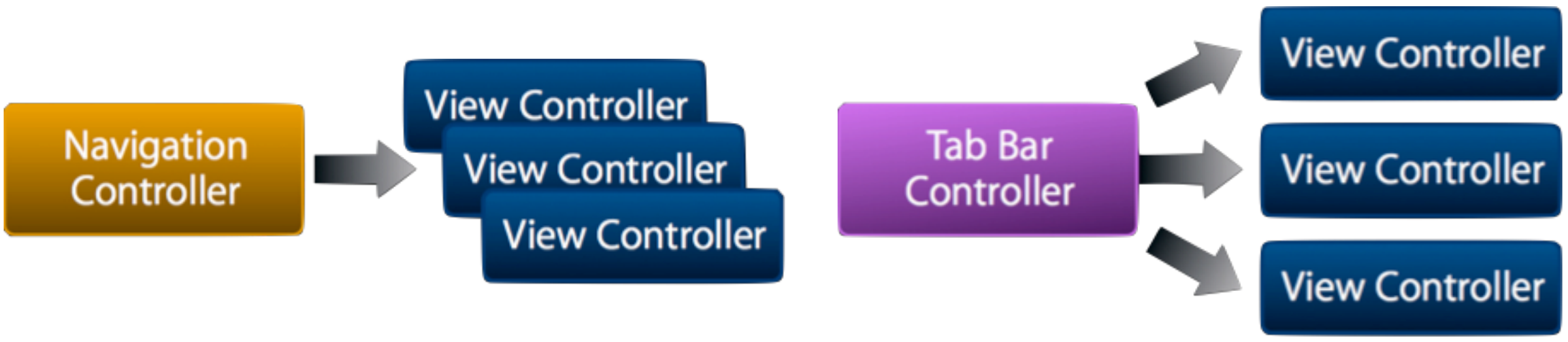
View Controller provides a well defined starting point

UIViewController

- Provides a basic building block within apps
  - Manages a screenful of content
  - Includes links to a view
  - Subclass this class to add to your application logic

Managing a number of Views with controllers

- Create your own UIViewController for each screenful
  - Plug them together using one of the existing **composite** view controllers



Your View Controller

- The UIViewController superclass has a view property
- Loads lazily
- On demand when requested by the application
- Can also be purged on demand (e.g. low memory)
- Never call **`-loadView`**
  - Let Cocoa do it when it wants
  - Don't assume it is called when the instance is created
  - That is what **`-(id)initWithNibName`** is for

View Controller Lifecycle

- `- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil`
  - Perform some initial set up, but nothing view related

- `- (void)viewDidLoad`
  - View has been loaded so it can now be customised

- `- (void)viewWillAppear:(BOOL)animated`
  - View is about to show on the screen. Good place to load data, etc

- `- (void)viewWillDisappear:(BOOL)animated`
  - View is about to leave the screen. Good place to save data, scroll position, etc

UNIVERSITY OF LIVERPOOL

Using protocols and delegates to support views

Lecture Set 3a - Views, Delegates and Tables

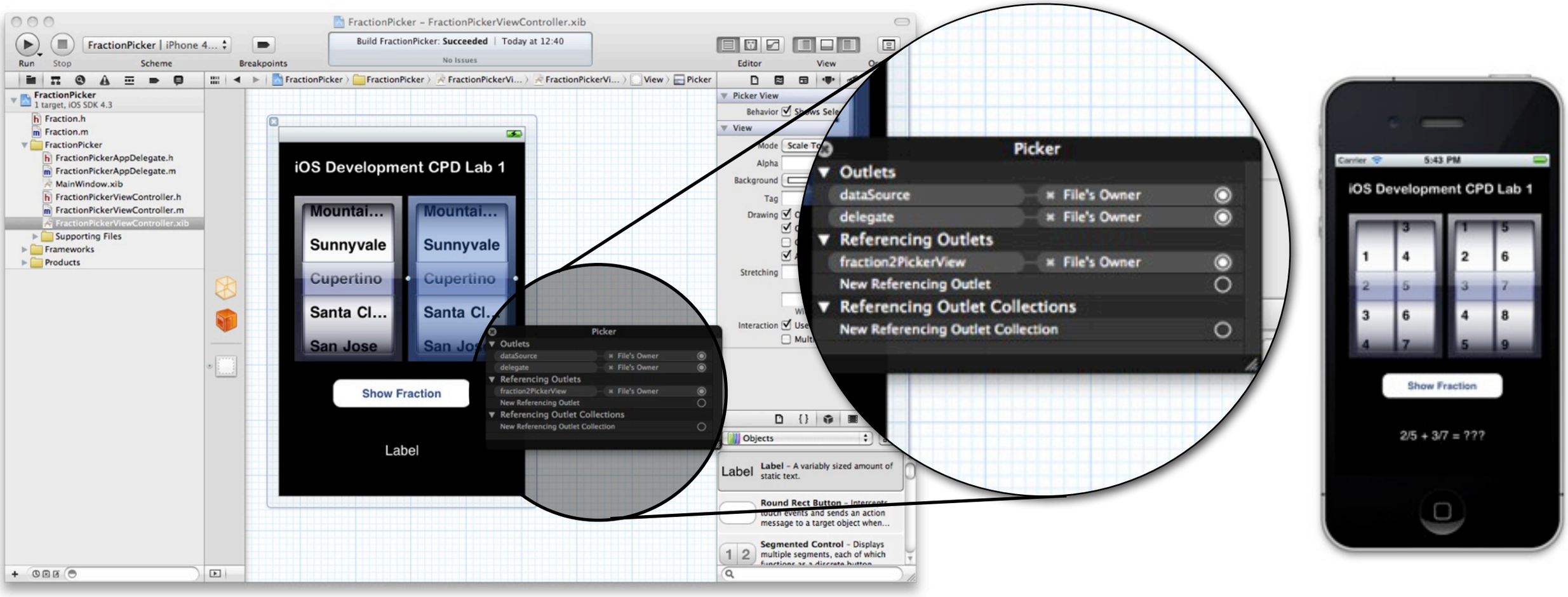
Extending other views

- Managing some interface elements can be trivial
  - Properties can be used to set or query the UI objects state
  - Other methods can be used to modify the object in some way
  - e.g. UISliders, UISwitches, Segmented Controls
- Other interface elements need to be extended in some way to facilitate the correct functionality
  - Determining how much data appears in an interface object
  - Determining what data appears
  - Determining what should happen if an element is selected.
  - For example:
    - Image Pickers
    - Table Views
    - Picker Views (e.g. from Day 1's tutorial)
- Delegation typically used in these cases
  - Delegate has to adopt some protocol
    - For the element itself
    - For the data source


```
- (void)viewDidLoad {
    [super viewDidLoad];

      self.view.backgroundColor = [UIColor viewFlipsideBackgroundColor];
      // Update the game size, assuming that the set value is valid
      if (gameSize>3)
            [gameSizeControl setSelectedSegmentIndex:gameSize-3];

      // Update the other controls
      [playerAIDifficultyControl setContinuous:FALSE];
      [playerAIDifficultyControl setValue:playerAIDifficulty animated:NO];
      [playerAIDifficultyControl setMaximumValue:1.0];
      [playerAIDifficultyControl setMinimumValue:0.0];

      [playerAISettingControl setOn:playerAISetting animated:NO];

}
```

Example: UIPickerView

- UIPickerViews provide a way of selecting a value from a list of possible values
  - Multiple lists, or components can be defined and displayed
    - UIDatePicker is a custom subclass of UIPicker, designed to display times and dates
- However, the object needs a delegate to provide the data for each component, and to define the number of components (**dataSource**)
- Also needs a delegate to determine what happens when a value is selected (**delegate**)



Adopting the UIPicker protocols

- The view controller owning the UIPickerView should adopt the two protocols
  - UIPickerViewDelegate
    - These methods return height, width, row title, and the view content for the rows in each component.
      - It must also provide the content for each component's row, either as a string or a view.
    - Typically the delegate implements other optional methods to respond to new selections or deselections of component rows.
  - UIPickerViewDataSource
    - The data source provides the picker view with the number of components and the number of rows in each component.

UIPickerViewDelegate

The didSelectRow method is called when a row is selected in one of the components. As both rows and components are 0-indexed, these can be used to index some data model that relates to the value selected.

In this example, we used the NSIntegers simply as integers, rather than as indexes into some model.

```
// ================================================================
#pragma mark - UIPickerViewDelegate methods
// ================================================================


- (void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)row
inComponent:(NSInteger)component
{
    // Don't care which component changed value.
    NSInteger myNumerator = 1+[pickerView selectedRowInComponent:NUMERATOR];
    NSInteger myDenominator = 1+[pickerView selectedRowInComponent:DENOMINATOR];

     ...
}

(NSString *)pickerView:(UIPickerView *)pickerView
             titleForRow:(NSInteger)row forComponent:(NSInteger)component
{
    // Both pickers just display numbers from 1 to 9
    return [NSString stringWithFormat:@"%d", 1+row];
}
```

The titleForRow method is called when creating the interface element (or when the element is being updated with new data). It is called for each element in each component.

In this case, we simply create a string based on the number of the row +1 (as we don't want to show the value zero).

UIPickerViewDataSource

```
// =============================================================================
#pragma mark - UIPickerViewDataSource methods
// =============================================================================

// Returns the number of components (or “columns”) that the picker view should display.
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    // Ignore the value of pickerView, as we assume only one picker for now
    return NUMBER_OF_PICKER_COMPONENTS;
}

// Returns the number of rows for the component.
- (NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:(NSInteger)component
{
    // Ignore the value of pickerView, as we assume only one picker for now
    return MAX_NUMBER_ON_PICKER;
}
```

The **numberOfComponentsInPickerView** method is called to determine the number of “wheels” in the picker - in this case the value was 2.

The **numberOfRowsInComponent** method is called to determine the number of rows (or values) in each component. This will determine the number of times the titleForRow (in the UIPickerViewDelegate) will be called for a given component.

This value is often generated from the number of elements in some data model.

UNIVERSITY OF LIVERPOOL

Table Views

Lecture Set 3a - Views, Delegates and Tables

Table Views

- Display lists of content
  - Single column, multiple rows
    - Vertical scrolling
    - Large data sets
- Powerful and ubiquitous in iPhone applications
  - Two table-view styles
- Content can be a single list or consist of multiple sections
  - Each with headers and footers

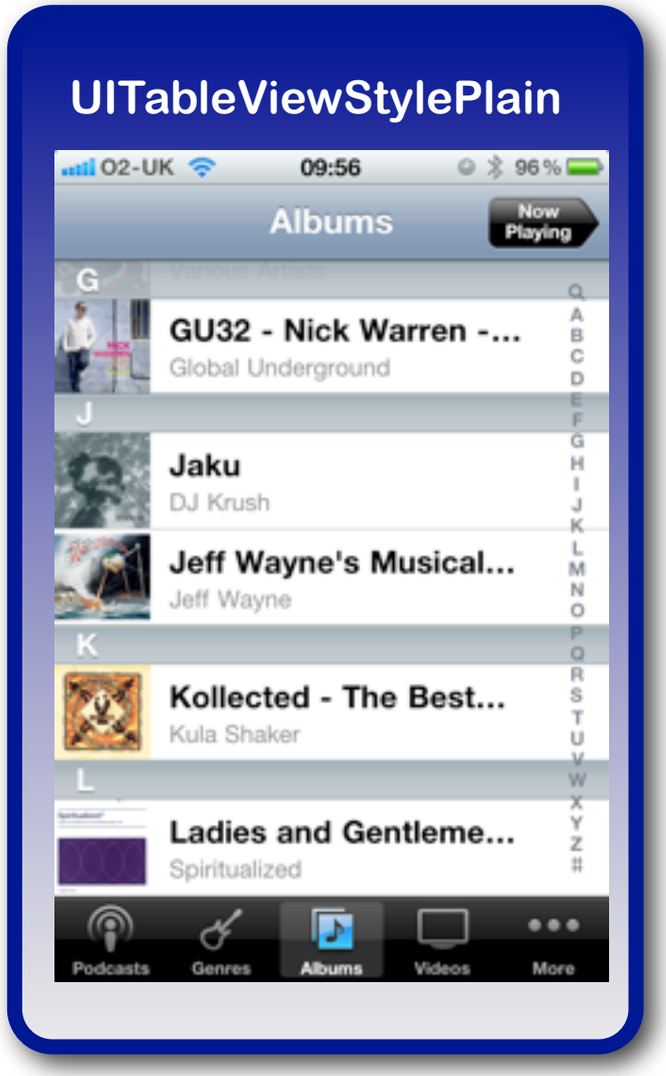


Table View Anatomy

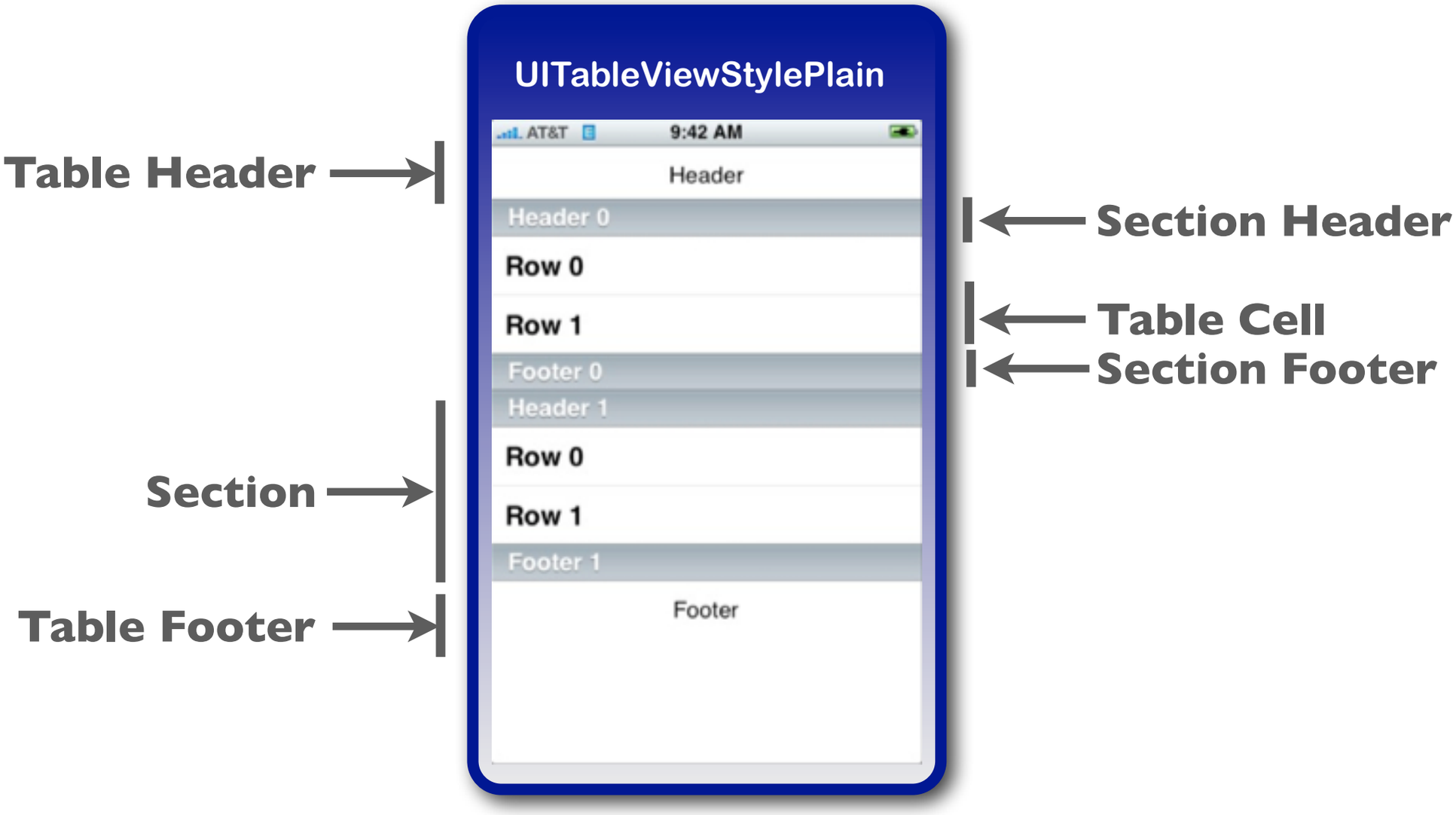
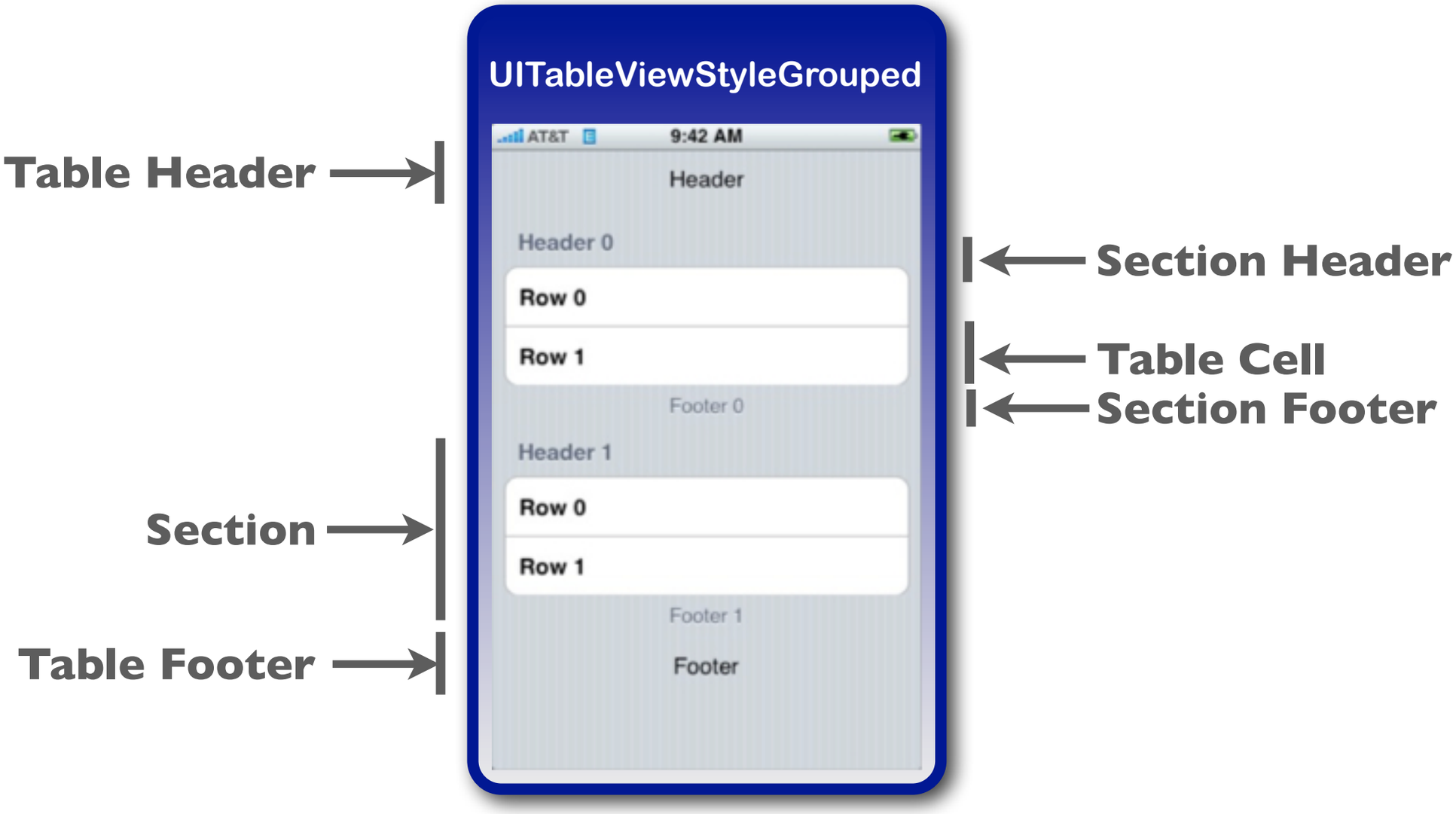


Table View Anatomy



Loading and displaying data in a table view

- The naive approach is to load everything at once!
  - Time and resource costs are huge!
    - Possibly will involve processing data that is unnecessary, such as entries near the bottom of the list

- Much better to take a lazy approach
  - Only load data as it is needed!
  - Assign responsibility of providing data to the table as and when it is necessary
  - Create a **Data Source delegate**
    - Similar to the approach used with UIPickerView

UITableViewDataSource Protocol

- A number of methods are defined in the protocol for passing data to a table
- The most important ones are illustrated in the code fragments below
  - **numberOfSectionsInTableView:**
    - This is optional (defaults to return 1), but determines how many sections in the table
  - **tableView:numberOfRowsInSection:**
    - This is **required**. States how many rows should appear in the table for each section

Cell Reuse

When asked for a cell, it would be expensive to create a new cell each time

Cells are therefore reused!!!

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.

    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.

    return [myStrings count];
}
```

UITableViewDataSource Protocol

- tableView:cellForRowAtIndexPath:
  - This provides the **cells for the table view** as needed

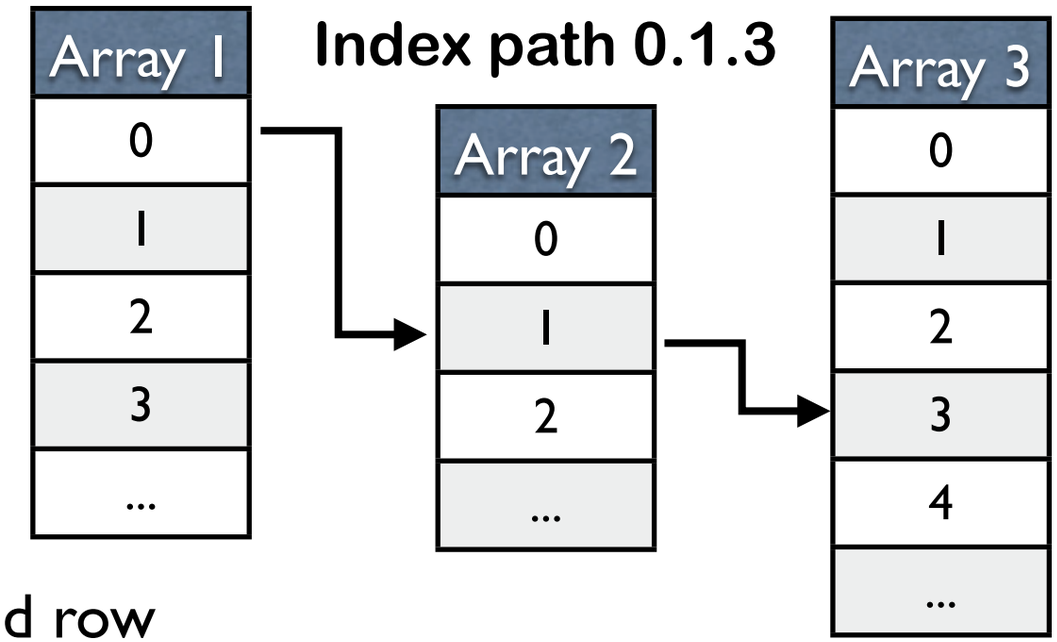
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
 reuseIdentifier:CellIdentifier];
    }

    // Configure the cell...
    [[cell textLabel] setText:[myStrings objectAtIndex:[indexPath row]]];

    return cell;
}
```

- Data is requested and used to define a cell
  - when the row becomes visible
  - when an update is explicitly requested
    - by calling the method reloadData or reloadRowsAtIndexPaths:withRowAnimation:

NSIndexPath

```
@interface NSIndexPath (UITableView)
// A UITableView specific category for
// extending the use of NSIndexPath

+ (NSIndexPath *)indexPathForRow:(NSUInteger)row
                       inSection:(NSUInteger)section;

@property(nonatomic,readonly) NSUInteger section;
@property(nonatomic,readonly) NSUInteger row;

@end
```

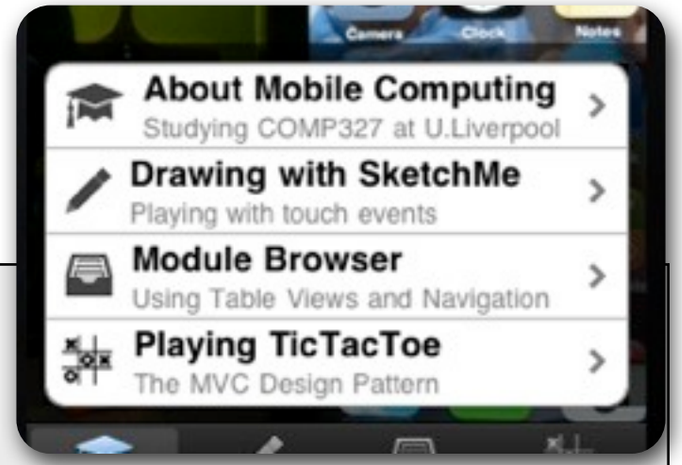
- This Foundation class represents the path to a specific node in a tree of nested array collections
  - Known as an **index path**



- Used by tables to index by section and row
  - A category on this class exists to simplify access to rows and sections within an index path (see above)

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    ...
    [[cell textLabel] setText:[myStrings objectAtIndex:[indexPath row]]];
    ...
}
```

UITableViewDataSource Example




```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [moduleInfoArray count];
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:CellIdentifier];
    }

    // Configure the cell...
    [[cell textLabel] setText:[[moduleInfoArray objectAtIndex:[indexPath row]] objectForKey:TITLE]];
    [[cell detailTextLabel] setText:[[moduleInfoArray objectAtIndex:[indexPath row]] objectForKey:SUBTITLE]];
    [cell setAccessoryType:UITableViewCellAccessoryDisclosureIndicator];

    NSString *path = [[NSBundle mainBundle] pathForResource:[[moduleInfoArray objectAtIndex:[indexPath row]]
                                                objectForKey:IMAGE] ofType:@"png"];
    UIImage *theImage = [UIImage imageWithContentsOfFile:path];
    cell.imageView.image = theImage;

    return cell;
}
```

Datasource: Headers and Footers

- The data source can also provide the text for headers and footers for each section


```
- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section
{
    // Configure the cell...
    NSDictionary *house = [myKingQueensModel objectAtIndex:section];
    return [house objectForKey:@"title"];
}

- (NSString *)tableView:(UITableView *)tableView titleForFooterInSection:(NSInteger)section
{
    // Configure the cell...
    NSDictionary *house = [myKingQueensModel objectAtIndex:section];
    return [house objectForKey:@"epoch"];
}
```

UITableView Delegate

- A second delegate is used to customise the **appearance** and **behaviour** of a table
  - Defines methods for:
    - Configuring rows for the table
    - Managing Accessory Views
      - i.e. arrow icons to the right of a cell
    - Managing Selections
      - Both determining when a cell is about to be selected
        - so it can be disabled if the selection is determined to be invalid
      - And determining what cell has been selected
    - Modifying the height or view of Section Headers / Footers
    - Editing Table Rows or Reordering Table Rows

UITableView Delegate

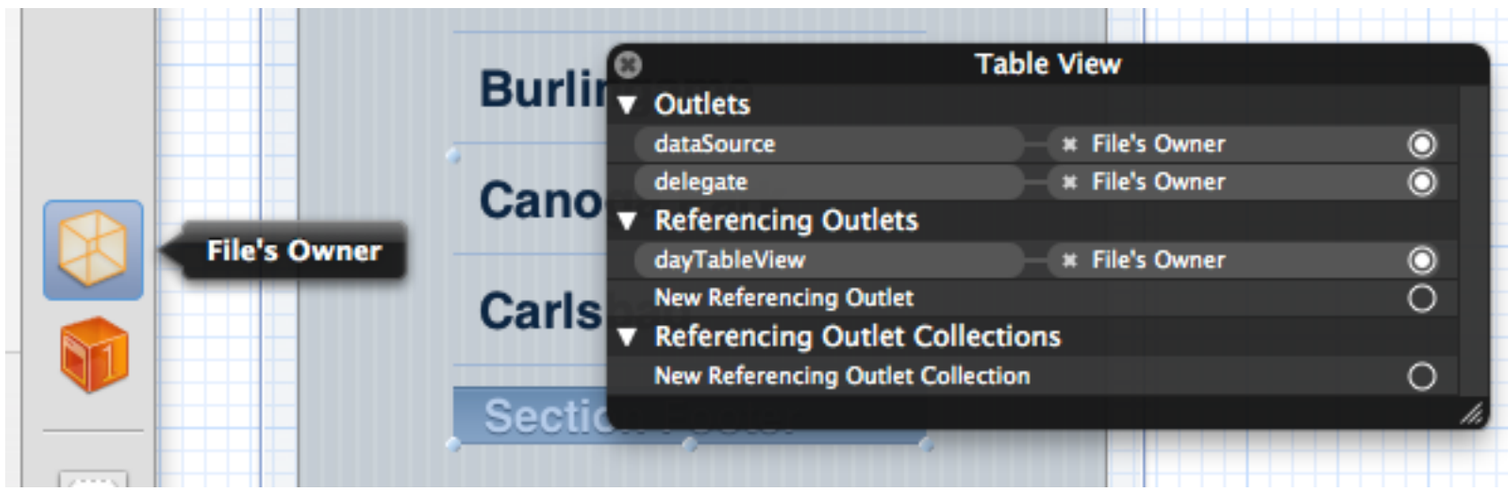
- **tableView:didSelectRowAtIndexPath**
  - This is typically included to determine what happens when a cell is selected
    - The cell selected is determined by the indexPath
  - As tableViews are commonly used within UINavigationControllers, stub code is included (but commented) to provide navigation logic.

```
#pragma mark Table view delegate

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    // Navigation logic may go here.

    NSString *sourceStr = [[moduleInfoArray objectAtIndex:[indexPath row]]
                                            objectForKey:CONTENTPATH];
    NSLog(@"Opening file %@", sourceStr);

   ...

}
```

This method (defined in the UITableView's delegate, determines what should happen when a cell is selected. The cell (and section in which it exists) is defined by the indexPath.

UITableViews and their delegates

- Tables can be added to nib files using UITableViews
  - Need to associate the table-view delegate and dataSource to some file containing the relevant delegate methods
    - This is typically the File's owner



UITableView class

- The class of the table itself
  - Defines a number of methods for the class
    - Including:
      - Table configurations (mainly through properties)
      - Accessing cells and sections
      - Scrolling the table view
      - Managing Selections
      - Inserting and deleting cells
      - Managing the editing of table cells
      - Reloading the Table View and Data
      - Accessing the drawing areas of the Table View

Tableview methods example

This method can be used to insert an array of rows within a table, and specify how the animation may appear, for example, fade, slide from the right or left, top or bottom, etc. Similar methods exist for deleting rows, etc.

This will cause the datasource delegate to be called, to retrieve the appropriate data. It is therefore imperative that the data also exists for these new rows.

```
- (void)add:(id)sender
{
    [myEditableKingQueensModel insertObject:[self createNewKingQueen] atIndex:0];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0 inSection:0];

    [[self tableView] insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                            withRowAnimation:UITableViewRowAnimationFade];

    [[self tableView] scrollToRowAtIndexPath:indexPath
                            atScrollPosition:UITableViewScrollPositionTop
                                    animated:YES];
}
```

This method will scroll the table to reveal a row, and position it somewhere within the tableview (e.g. at the top, bottom, center etc.)

UITableViewController

- Convenient starting point for view controller with a table view
  - Table view is automatically created
  - No need to create nib file for the view, as the entire view is a table.
  - Often used within UINavigationControllers
  - **Controller is table view's delegate and datasource**
- Takes care of some default behaviours
  - Calls -reloadData the first time it appears
  - Deselects rows when user navigates back
  - Flashes scroll indicators

UNIVERSITY OF LIVERPOOL

Cells within Table Views

Lecture Set 3a - Views, Delegates and Tables

Creating Cells

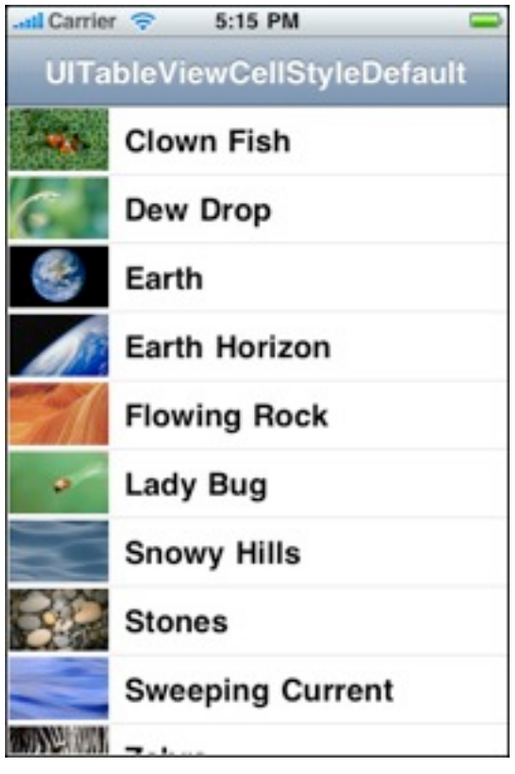
- Cells within table views are typically created, or reused from within **tableView:cellForRowAtIndexPath:**
  - A cell "queue" is used to store unused cells.
    - If no cells exist within the queue, then they are created
    - Otherwise they are retrieved from the queue.
  - As different types of cell can appear in a table, they are identified using a CellIdentifier string

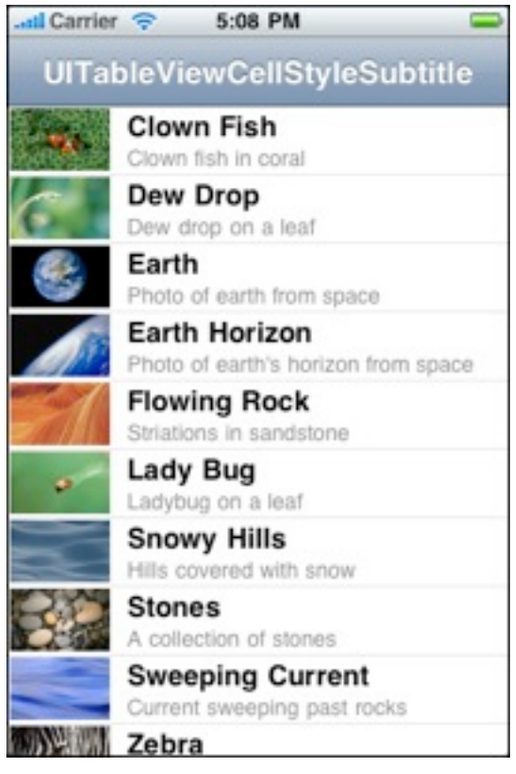
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier];
    }

    // Configure the cell...
    [[cell textLabel] setText:[myStrings objectAtIndex:[indexPath row]]];

    return cell;
}
```

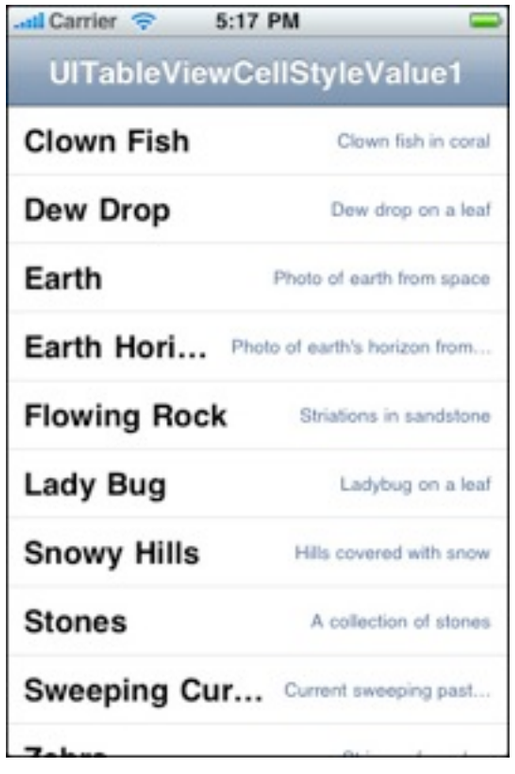
Default Cell Styled

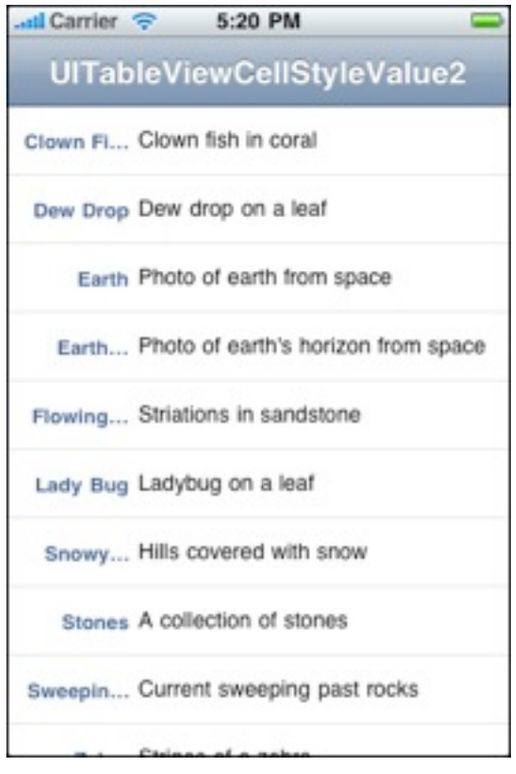
- There are four default cell types
  - The type is given as the style parameter within the init method when creating the cell
-

```
...
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                       reuseIdentifier:CellIdentifier] autorelease];
...
```

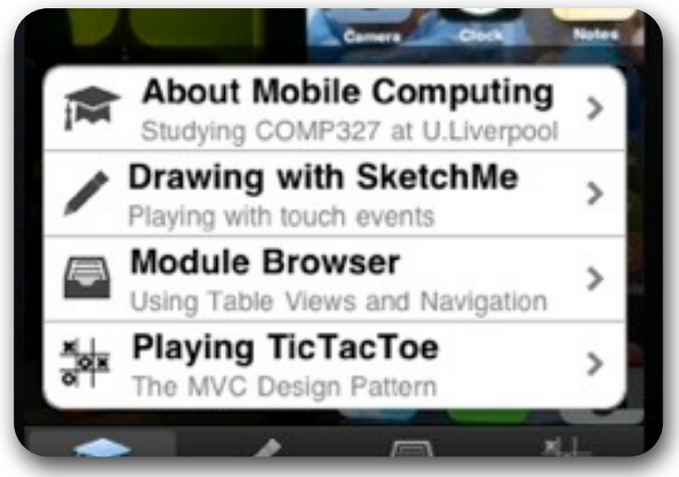








Accessory Views



- There are three standard kinds of accessory views
  - These appear to the right of the cell
  - Selected using the method setAccessoryType: when creating or changing a cell

```
...
[cell setAccessoryType:UITableViewCellAccessoryDisclosureIndicator];
...
```

- **Disclosure Indicator** (*UITableViewCellAccessoryDisclosureIndicator*)
  - Used to indicate that selecting a cell will result in the display of another table view within a data hierarchy (typically using a UINavigationController).
- **Detail disclosure button** (*UITableViewCellAccessoryDetailDisclosureButton*)
  - Used to allow the selection of the disclosure button to mean something different to selecting the cell.
- **Check mark** (*UITableViewCellAccessoryCheckmark*.)
  - Use when a touch on a row should result in the selection of that item (e.g. in pop-up lists). Can be restricted to one or several rows.

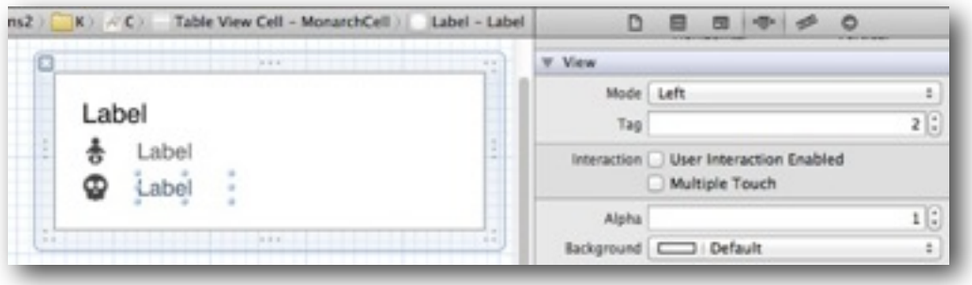
Cell Customisation

- Typically three ways of customising cells:
  - 1) Subclass a UITableViewCell
    - Provides ultimate control, but poor design can affect performance
  - 2) Adding subviews to a cell (programmatically)
    - Achieved by adding additional views to the cell
    - Often indexed by using Tags

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    ...
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                      reuseIdentifier:CellIdentifier];
    }

    birthLabel = [[UILabel alloc]
                      initWithFrame:CGRectMake(170.0, 5.0, 80.0, 15.0)];
    [birthLabel setFont:[UIFont systemFontOfSize:11.0]];
    [birthLabel setTag:BIRTHLABEL_TAG];
    [[cell contentView] addSubview:birthLabel];
    ...
```

Cell Customisation

- 3) Using nib files by hand
  - A nib containing the cell is created using interface builder
  - Tags are used to identify the fields, and the cell is identified with a cell identifier
- Each cell is then loaded, rather than calling alloc/init

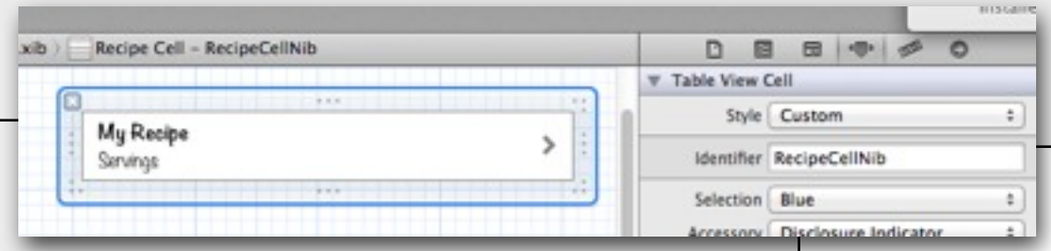



```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"MonarchCell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        [[NSBundle mainBundle] loadNibNamed:@"CustomMonarchCell" owner:self options:nil];
        cell = myCustomMonarchCell;
        [self setMyCustomMonarchCell:nil];
    }

    UILabel *monarchLabel = (UILabel *)[cell viewWithTag:MONARCHLABEL_TAG];

    [monarchLabel setText:@"Elisabeth II"];
}
```

Cell Customisation (new)

- 4) Registering nib files (`dequeueReusableCellWithIdentifier:forIndexPath:`)
  - Introduced in iOS6.
  - A nib containing the cell is created using interface builder, and associated with a custom cell
  - The nib is then registered when the view is loaded
  - Each cell is then loaded automatically if no reusable cell can be found in the queue

```
//  RecipeCell.h

#import <UIKit/UIKit.h>

#define RecipeCellReuseIdentifier   @"RecipeCellNib"

@interface RecipeCell : UITableViewCell

@property (weak, nonatomic) IBOutlet UILabel *detailTextLabel;
@property (weak, nonatomic) IBOutlet UILabel *titleLabel;

@end
```




```
- (void)viewDidLoad
{
    [super viewDidLoad];
     // Do any additional setup after loading the view, typically from a nib.
    ...

    UINib *recipeCellNib = [UINib nibWithNibName:@"RecipeCell" bundle:[NSBundle mainBundle]];
    [[self tableView] registerNib:recipeCellNib forCellReuseIdentifier:RecipeCellReuseIdentifier];

    ...
}
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    RecipeCodeCell *cell = [tableView dequeueReusableCellWithIdentifier:RecipeCodeCellReuseIdentifier
                                                                forIndexPath:indexPath];
    ...

    return cell;
}
```

UNIVERSITY OF LIVERPOOL

Questions?

Lecture Set 3a - Views, Delegates and Tables