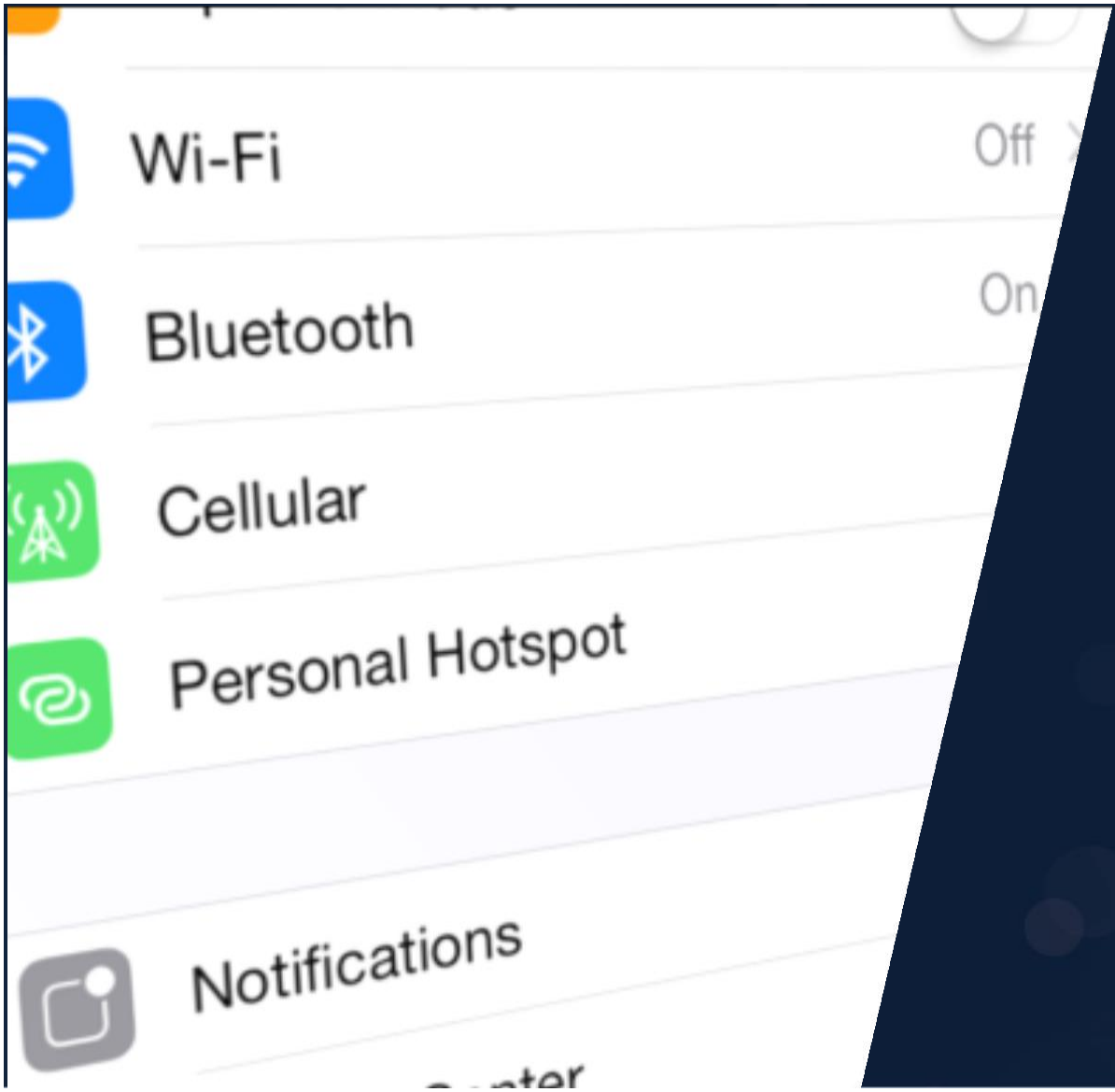
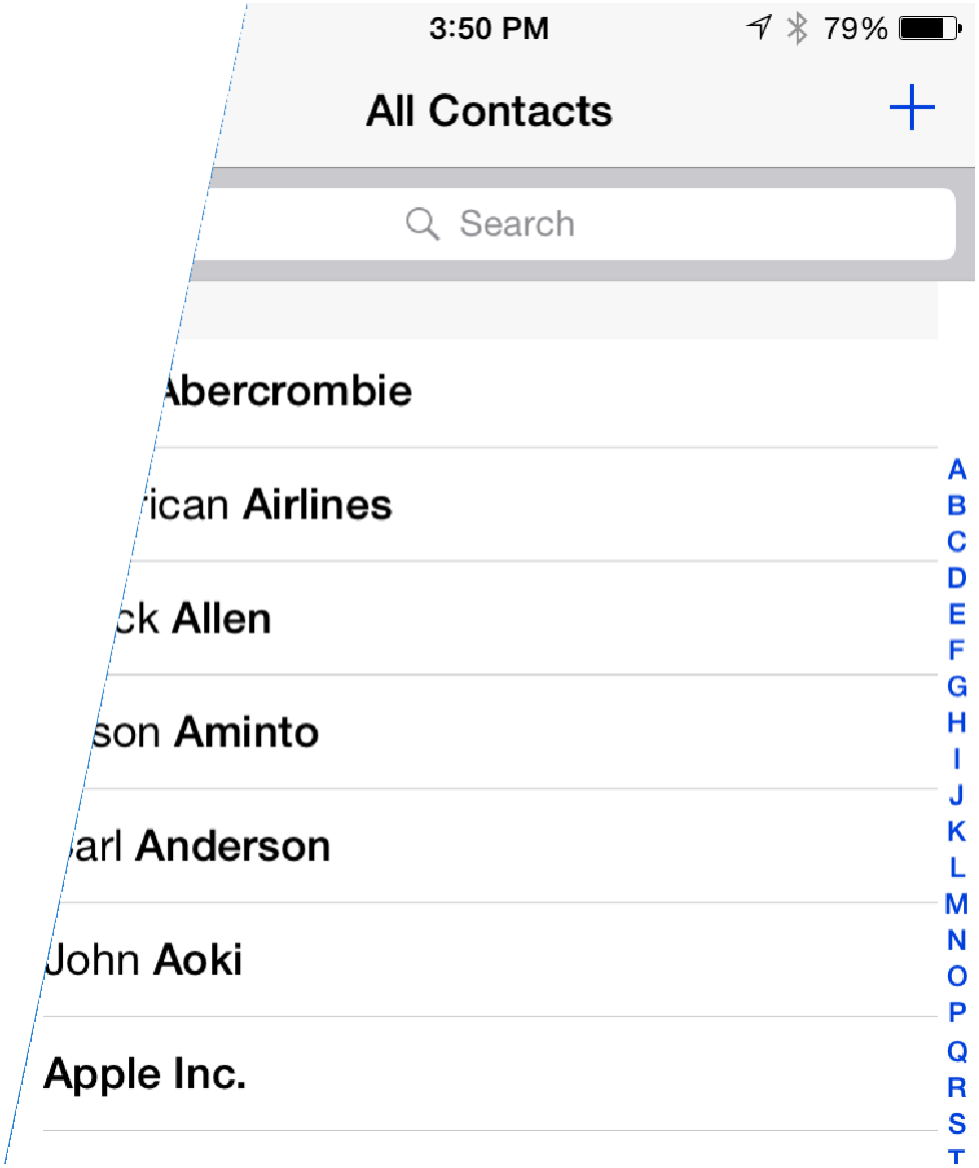


Fundamentals of TableViews

Objectives

1. Explore Table Views
2. Utilize built-in cell styles
3. Add selection behavior
4. Implement cell reuse



Explore Table Views

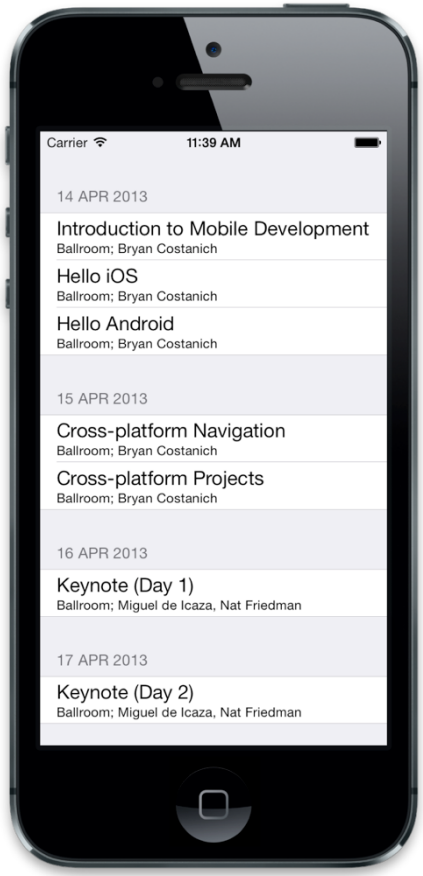
Tasks

1. Describe the Table View and its common uses
2. Identify the classes which make up Table View
3. Add a Table View to your UI
4. Fill the Table View with data

What is a Table View?

- Table Views are a built-in control in iOS to present a scrollable, selectable list of rows – similar to a **`ListBox`** in Windows or **`ListView`** in Android

- Table Views are highly customizable and very common in iOS applications



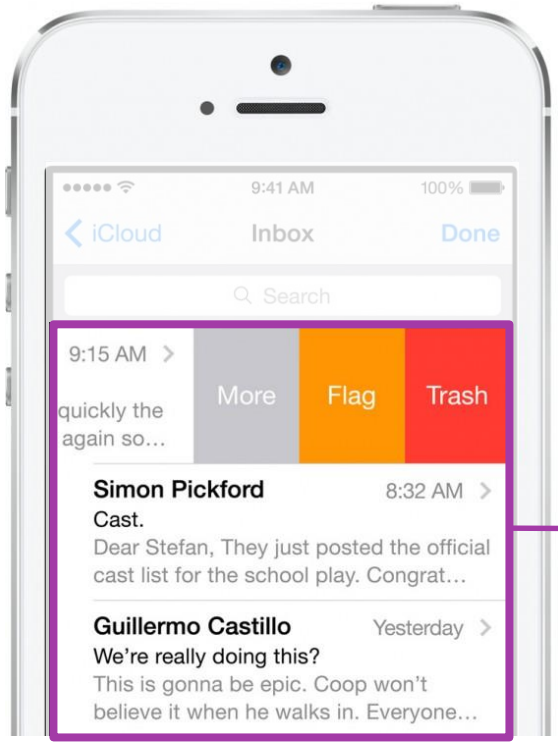
Demonstration

Examine how Table View is used in different applications

Components of a TableView

- Table View consists of several related classes, starting with **`UITableView`**

- Common to have the Table View take up the entire screen excluding any navigation bars, particularly on smaller devices

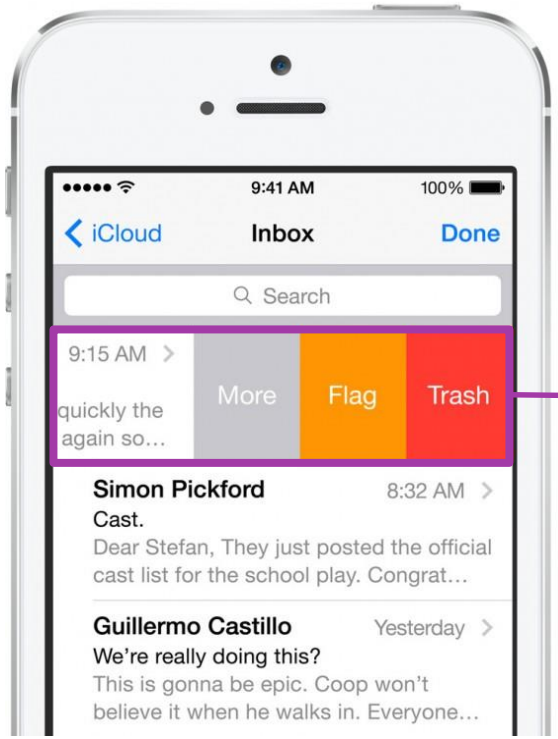


`UITableView` is the main **`UIView`** class which visualizes the scrollable table

Components of a TableView

- Each row in the Table View is a **`UITableViewCell`**

  - System has pre-defined cell styles, or you can create custom cells to display any type of data desired
  - Rows can be either fixed or variable height



`UITableViewCell` is a **`UIView`** that renders a single row of data in the table and provides the selection and interactivity

Adding a Table View to your UI

- Can add a Table View into your UI in code, or through the designer

Adding a Table View in code

- Can instantiate the **`UITableView`** and add as a subview to a screen

```
private UITableView tableView;

public override void ViewDidLoad()
{
    base.ViewDidLoad();

    tableView = new UITableView(View.Frame);
    Add(tableView);
}
```

Adding a Table View in code

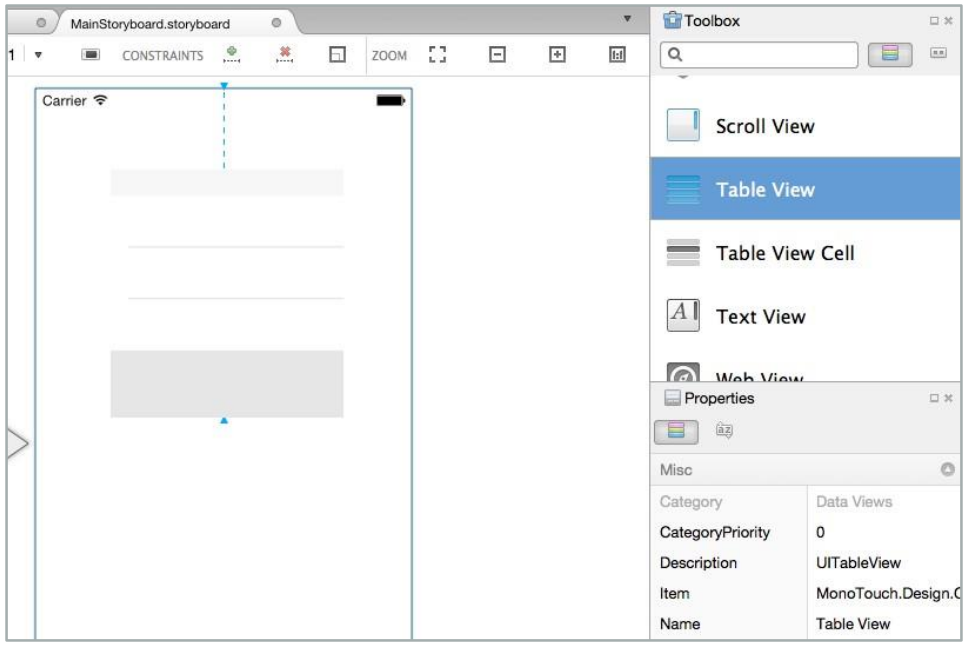
- Can also use a **`UITableViewController`** – this is a standard view controller with a built-in **`UITableView`**; can use a derived version of this class as a root view controller, or navigate to one for secondary pages

```
public class MyTableViewController : UITableViewController
{
   public override void ViewDidLoad()
   {
      base.ViewDidLoad();
      TableView.ContentInset = new UIEdgeInsets(20,  0, 0, 0);
   }
}
```

Has property to access created Table View

Using the Storyboard Designer

- Toolbox contains both Table View and Table View Controller elements which can be dragged onto designer surface



Can then set properties in the designer to control the visualization and behavior of the Table View

Group Exercise

Add a Table View to an application

Supplying data to the Table View

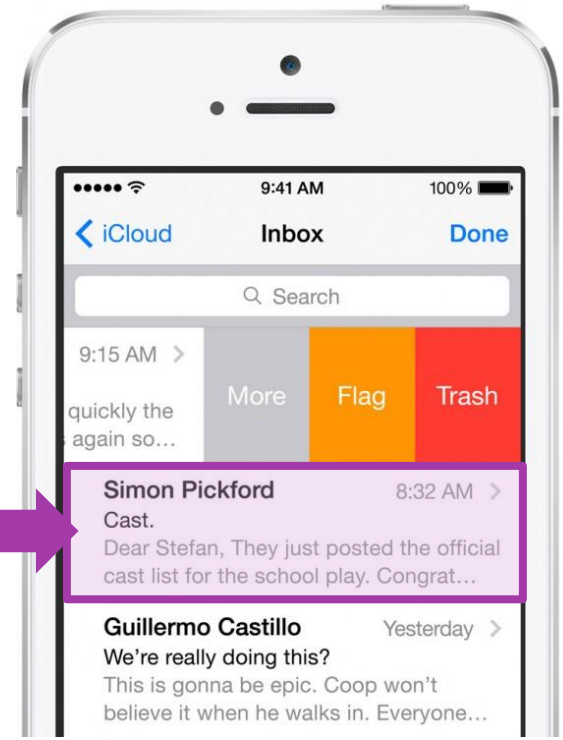
- A *data source* converts the apps internal data into visual rows that are displayed in the Table View

```
class Email
{
    string To;
    string From;
    string Subject
}
```

Data

Creates and populates the `UITableViewCell` to display a single item of data

`UITableViewDataSource`

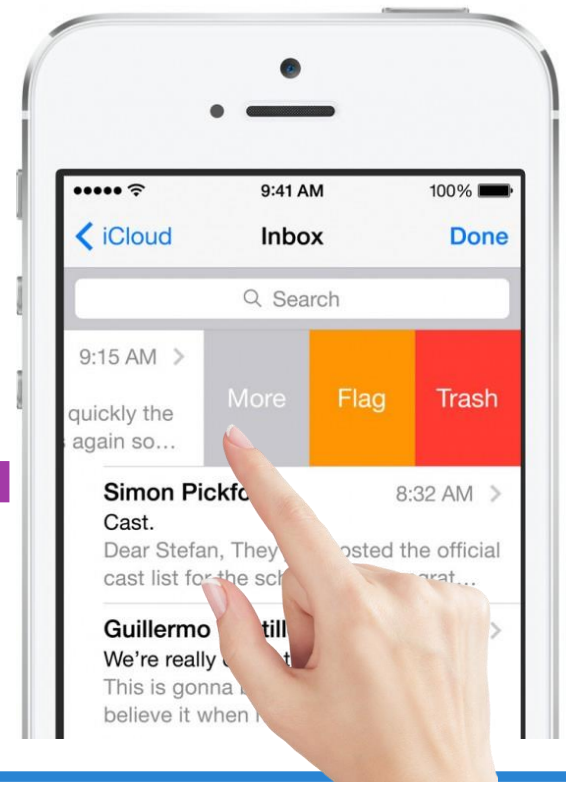


Interacting with the TableView

- Table View *delegate* receives notifications about user interactions

Must implement delegate class and connect to Table View to receive notifications

`UITableViewDelegate`



Separate data and behavior

- Table View needs both a data source and a delegate – can derive from these two abstract classes to provide data and behavior

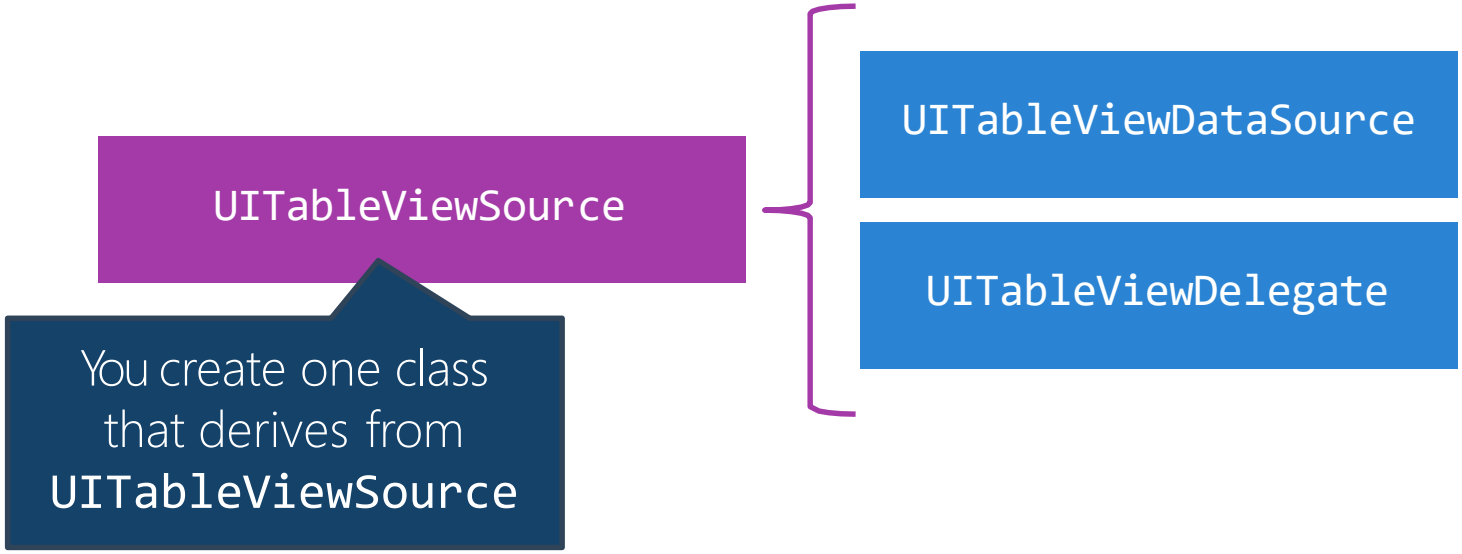
`UITableViewDataSource`

`UITableViewDelegate`

These are both *protocols* in iOS; in Xamarin, they are mapped to two abstract base classes you must extend

Sharing data and behavior together

- Alternatively, Xamarin provides a single abstract base class which implements both protocols; can provide data and behavior in a single derived class



Assigning the Table View source

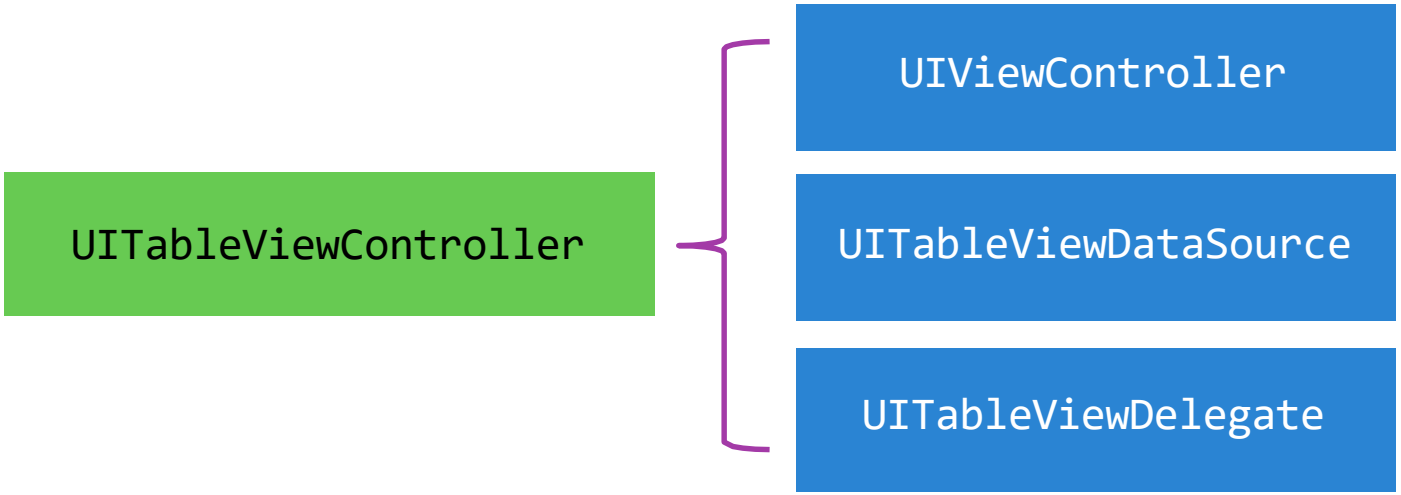
- An implementation of **`UITableViewSource`** must be assigned to the Table View **`Source`** property to be used as the protocol implementation

```
public class MyTableViewSource : UITableViewSource
```

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    this.tableView.Source = new MyTableViewSource();
    ...
}
```

UITableViewController

- Built-in **`UITableViewController`** also has support to implement the delegate and data source – can simply override the methods directly on the controller



Providing data to the Table View

- Table Views can be used to display any type of data – strings, custom types, etc.

```
public class MyTableViewController : UITableViewController
{
    private string[] names = { "Moe", "Curly", "Larry", "Shemp" };
    ...
}
```

The data for the table is most often held in an array or list by the data source class or the view controller

Note that we are using the Table View Controller approach here, but the same exact steps and overrides are used no matter which class implements the data source

Providing data to the Table View

- Two methods *must* be implemented by the data source to provide the data

`RowsInSection`

`GetCell`

Providing data to the Table View

- `RowsInSection` provides the total number of rows we want to display

```
public class MyTableViewController : UITableViewController
{
    ...
    public override nint RowsInSection(UITableView tableview, nint section)
    {
        return names.Length;
    }
}
```

The Table View supports different sections (groups), for this class we will assume a single section – check out iOS215 to learn about sections

Providing data to the Table View

- `RowsInSection` provides the total number of rows we want to display

```
public class MyTableViewController : UITableViewController
{
    ...
    public override nint RowsInSection(UITableView tableview, nint section)
    {
        return names.Length;
    }
}
```

Supports automatic casting from regular integer types

A `nint` is an integer that changes its size when you compile it for 32 bit or 64 bit

Providing data to the Table View

- **GetCell** returns a unique **UITableViewCell** for a index position

```
public class MyTableViewController : UITableViewController
{
    ...

    public override UITableViewCell GetCell(UITableView tableView,
                                            NSIndexPath indexPath)
    {
        string data = names[indexPath.Row];
        var cell = new UITableViewCell(CGRect.Empty );
        ...
        return cell;
    }
}
```

Index is described by a **NSIndexPath** which contains Row and Section properties that uniquely identify a cell in a table

Providing data to the Table View

- **`GetCell`** returns a unique **`UITableViewCell`** for a index position

```
public class MyTableViewController : UITableViewController
{
    ...

    public override UITableViewCell GetCell(UITableView tableView,
                                            NSIndexPath indexPath)
    {
        string data = names[indexPath.Row];
        var cell = new UITableViewCell(CGRect.Empty);
        cell.TextLabel.Text = data;
        return cell;
    }
}
```

`TableViewCell` has built-in sub-views to display the specific details for the given cell

Individual Exercise

Populating a Table View

Summary

1. Describe the Table View and its common uses
2. Identify the classes which make up Table View
3. Add a Table View to your UI
4. Fill the Table View with data

Utilize built-in cell styles

Tasks

1. Built-in cell styles
2. Changing the cell style

Built-in cell styles

- There are four built-in cell styles which support the most common data displays

Default

Support a single line of text stretched across the view with an optional image on the left

Built-in cell styles

- There are four built-in cell styles which support the most common data displays

Default

Subtitle

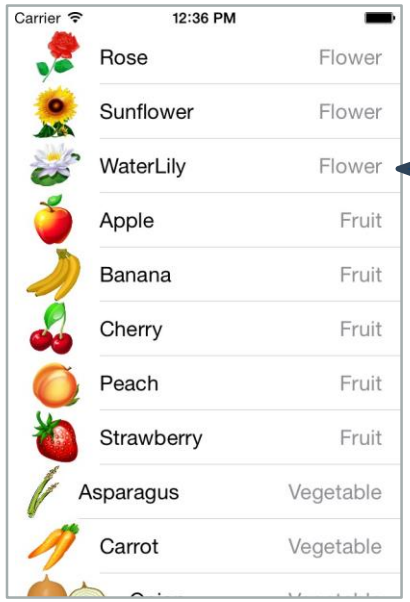
Add a detail text line under the main text label

Built-in cell styles

- There are four built-in cell styles which support the most common data displays

Default

Subtitle

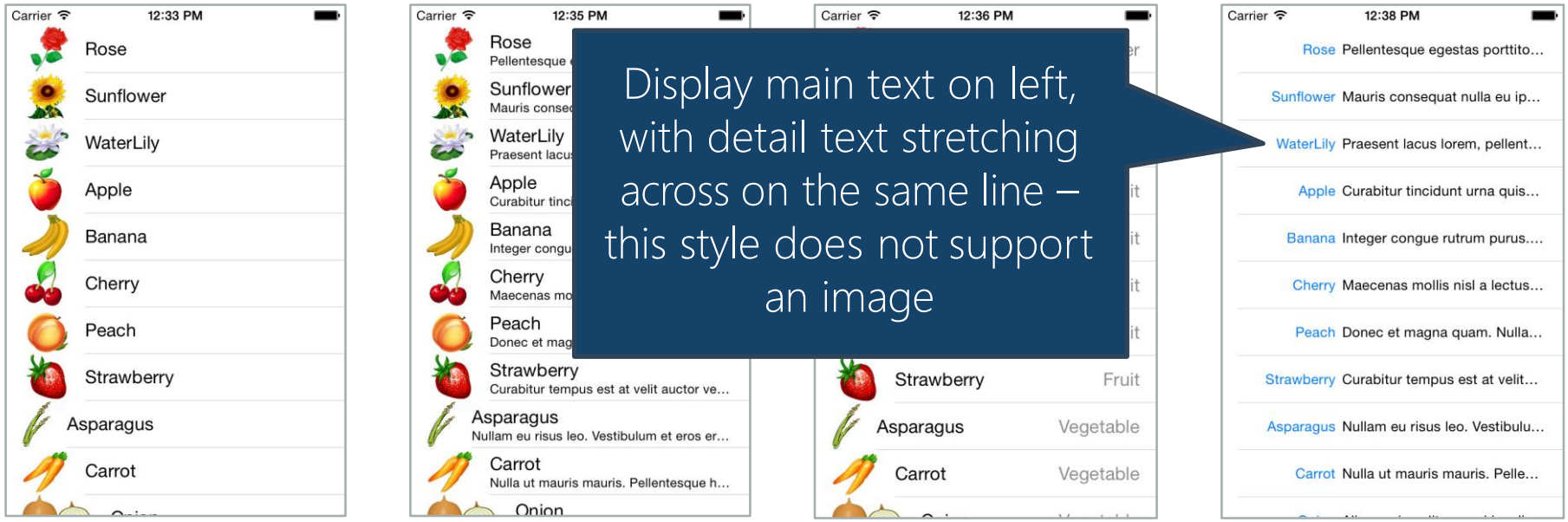


Value1

Display a small detail text element to the right of the main text

Built-in cell styles

- There are four built-in cell styles which support the most common data displays

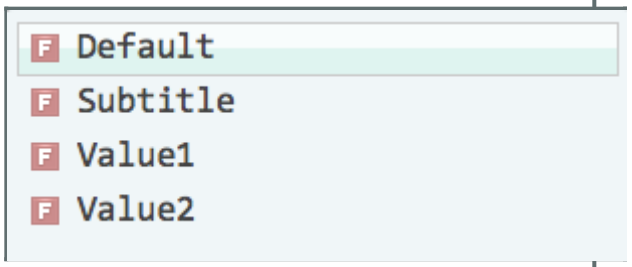


Default Subtitle Value1 Value2

Specify the style of the cell

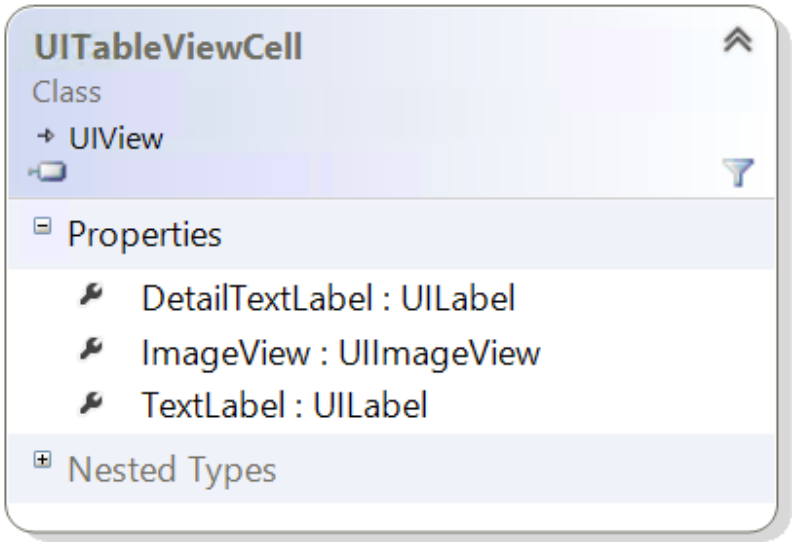
- Constructor for **`UITableViewCell`** takes the style as the first parameter

```
public  override UITableViewCell  GetCell(UITableView  tableView,
                                          NSIndexPath  indexPath)
{
    var data =  plants[indexPath.Row];
    var cell = new  UITableViewCell(UITableViewCellStyle.

    ...


    return  cell;
}
```



Configure the Table View cell contents

- `UITableViewCell` has three subviews which will be assigned based on the style of the cell, or left `null` if it does not apply



`TextLabel` is always available

`DetailTextLabel` is available with all styles except `Default`

`ImageView` is available with all styles except `Value2`

Configure the Table View cell contents

- Should set values into subviews as part of the **`GetCell`** implementation

```
public override UITableViewCell GetCell(UITableView tableView,
                                        NSIndexPath indexPath)
{
    var data = plants[indexPath.Row];
    var cell = new UITableViewCell(
                UITableViewCellStyle.Subtitle, null);

    cell.TextLabel.Text = data.Name;
    cell.DetailTextLabel.Text = data.Description;
    cell.ImageView.Image = UIImage.FromBundle(data.Image);
    return cell;
}
```

Remember that not all **`UIViews`** are created for each style – they will be **`null`** if they are not available

Group Exercise

Using the built-in cell styles

Summary

1. Built-in cell styles
2. Changing the cell style

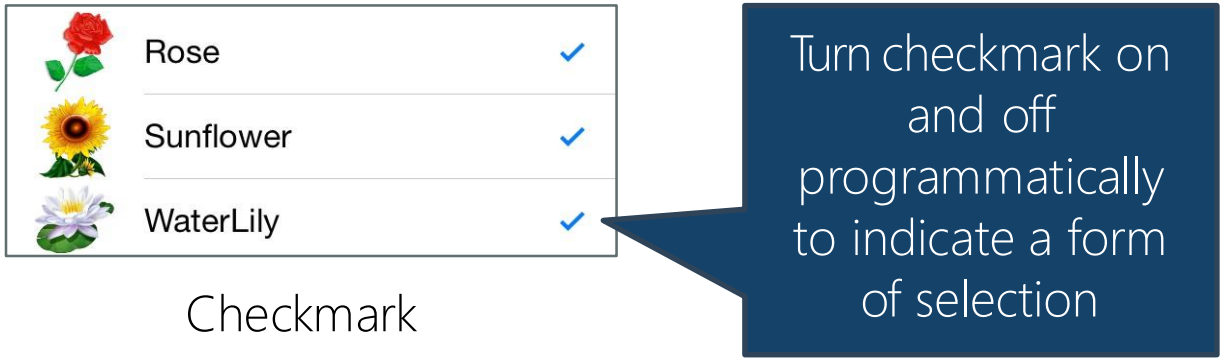
Add selection behavior

Tasks

1. Adding an accessory view
2. Working with the delegate methods
3. Responding to the accessory tap

Setting an accessory style

- Table View cells can include an optional *accessory indicator* on the right side of the cell that indicates some type of interactivity

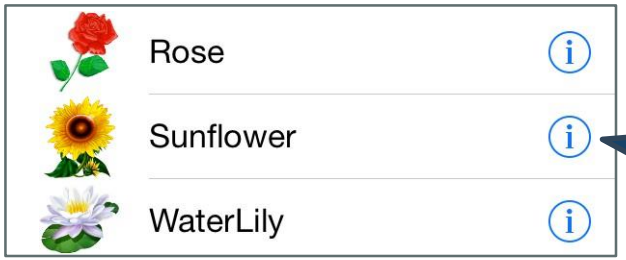


Checkmark

Setting an accessory style

- Table View cells can also include an optional *accessory indicator* that indicates some type of built-in interactivity – selection, navigation, etc.

Checkmark



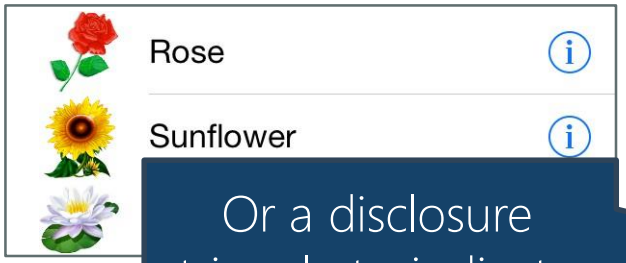
DetailButton

Displays a button which can be tapped independent of the row itself

Setting an accessory style

- Table View cells can also include an optional *accessory indicator* that indicates some type of built-in interactivity – selection, navigation, etc.

Checkmark



DetailButton

Or a disclosure triangle to indicate navigation to a details page

DisclosureIndicator

One other accessory style – DetailDisclosureButton, combines the disclosure indicator and the detail button together – allowing for two different interactions on the row

Setting an accessory style

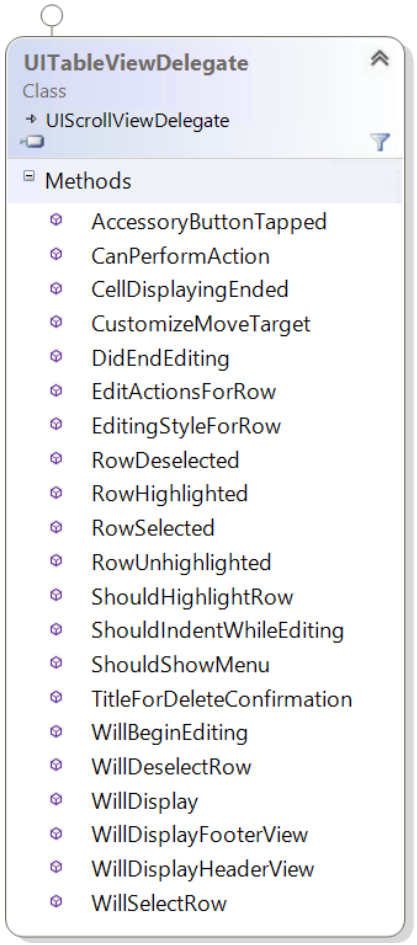
- Accessory property controls the accessory indicator display; it defaults to None and should be set in the **`GetCell`** implementation

```
public override UITableViewCell GetCell(UITableView tableView,
                                        NSIndexPath indexPath)
{
    var cell = new UITableViewCell(...);
     ...

    cell.Accessory = UITableViewCellAccessory.
    return cell;
}
```

- Checkmark
- DetailButton
- DetailDisclosureButton
- DisclosureIndicator
- None

Managing interactions

- `UITableViewDelegate` protocol provides notifications for interactions with the Table View
  - Row activation
  - Editing actions
  - Swipe actions
  - Moving Rows
  - …



Working with Row Selection

- `RowSelected` override is called when a row is tapped – this normally is used to activate some action (e.g. selection, navigation, feature, etc.)

```
public class MyTableViewController : UITableViewController
{
    public  override  void RowSelected(UITableView tableView,
                                       NSIndexPath  indexPath)
    {
        // TODO: indexPath is the row being tapped
    }
}
```

As with the data source methods, the same overrides are performed no matter where the delegate implementation is located

Individual Exercise

Using the accessory styles and row selection

Summary

1. Working with the delegate methods
2. Adding an accessory view
3. Responding to the accessory tap

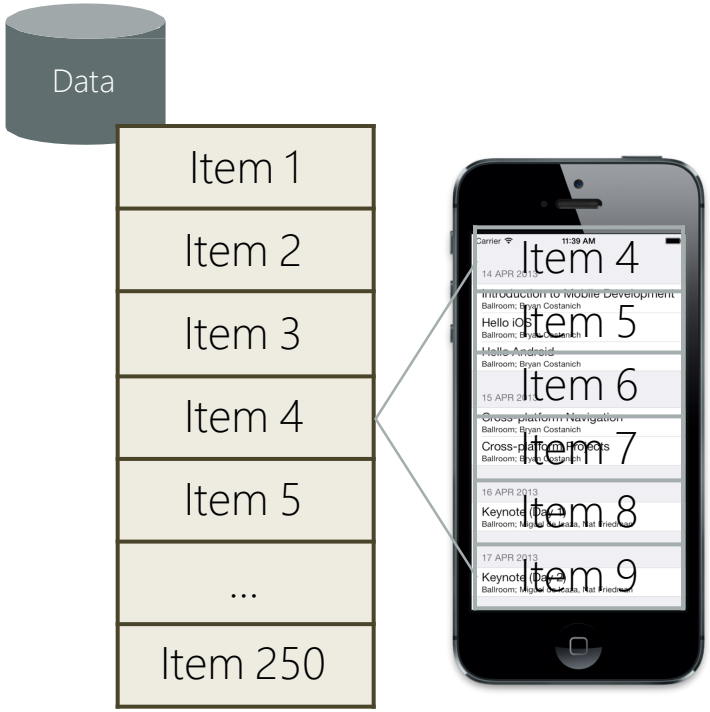
Implement Cell Reuse

Tasks

1. What is cell reuse?
2. Participating in cell reuse
3. Managing the cell data

What is Cell reuse?

- `UITableViewCell`s require more memory than just the data being displayed
- Often cannot display all the available data at one time
- iOS tries to optimize memory by only creating enough cells to display what is visible and then *reuse* the cells as you scroll through the data



Participating in cell reuse

- Key to cell reuse is a *reuse identifier* – this is a custom string that uniquely identifies the style of the cell, which is assigned once when the cell is created

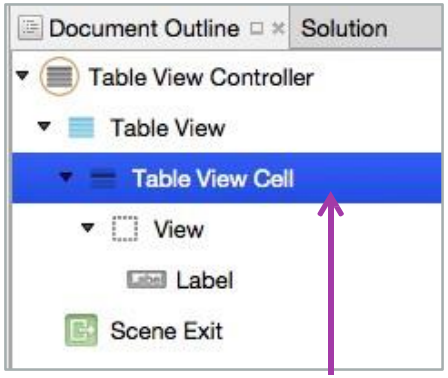
- Two ways to assign the reuse identifier

Storyboard Designer

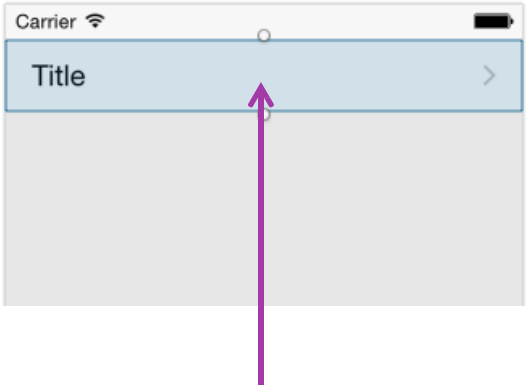
Cell constructor

Reuse identifierin the Designer

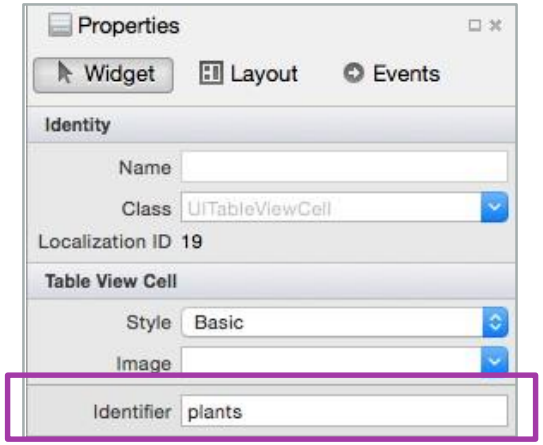
❖ Can design cells in the Storyboard designer – called *prototype cells*



Select the Table View Cell in the document outline



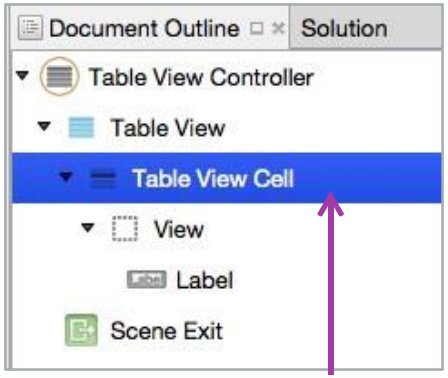
.. or by clicking on the prototype cell in the Table View



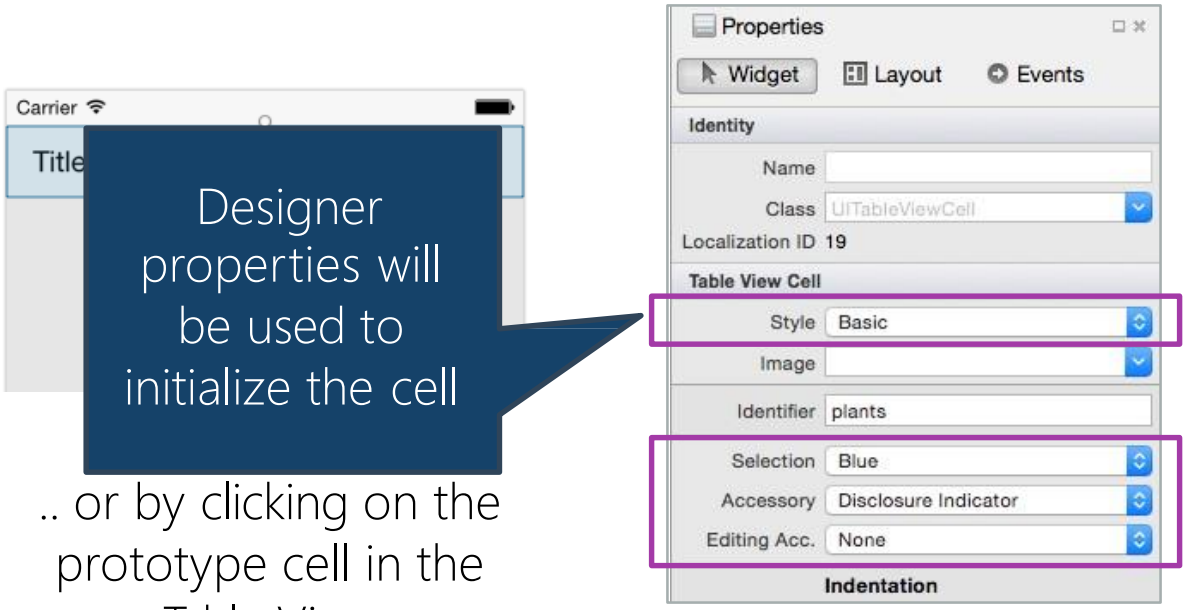
... then set the reuse identifier for this prototype cell

Reuse identifierin the Designer

- Design cells in the Storyboard designer – called *prototype cells*



Select the Table View Cell in the document outline



.. or by clicking on the prototype cell in the Table View

Reuse identifier in code

- If you don't want to use the designer, you can also pass a reuse identifier when you create a new cell; this assigns the cell to a unique "pool"

```
var cell = new
        UITableViewCell(UITableViewCellStyle.Default,
                                        "plants");
```

Pass the reuse identifier as a constructor parameter to the table view cell – the value does not matter, as long as each unique cell style has a unique id

Getting a designer-registered cell

- Use the **`DequeueReusableCell`** method to retrieve an existing Table View cell instead of always creating one

```
public override UITableViewCell GetCell(UITableView tableView, ...)
{
    var cell = tableView.DequeueReusableCell("plants");
    ...

    return cell;
}
```

Pass the reuse identifier so iOS knows the *style* of the cell it is looking for; iOS will look in the pool and return a cell, or create one for you if you used the designer

Getting a code-registered cell

- If you do not use the designer, then you need to test the return value from **`DequeueReusableCell`** – it returns **`null`** if no cell is in the pool, in which case you must create a new cell for that identifier

```
public override UITableViewCell GetCell(UITableView tableView, ...)
{
    var cell = tableView.DequeueReusableCell("plants");
    if (cell == null) {
       cell = new UITableViewCell(UITableViewCellStyle.Subtitle, "plants");
    }
    ...

    return cell;
}
```

This code is <u>not necessary</u> when you use the designer to register the cell

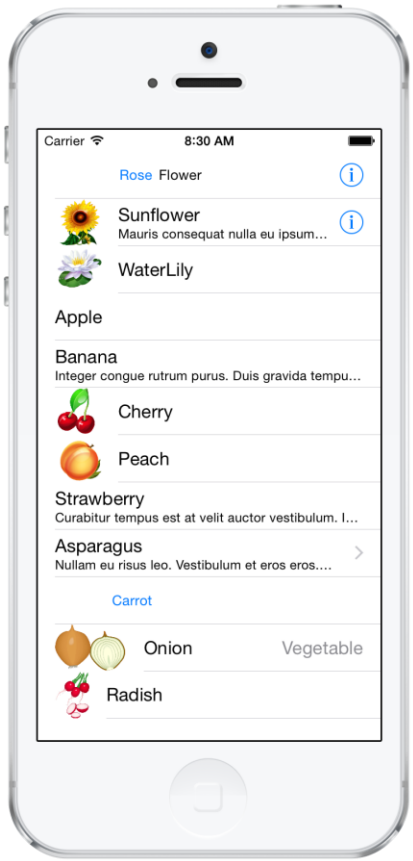
Setting values on the cell

- Fill in the details for the provided cell – must always set or clear values to ensure stale values from previous rows are not displayed

```
public override UITableViewCell GetCell(UITableView tableView, ...)
{
    var cell = tableView.DequeueReusableCell("plants");
    ...
    cell.TextLabel.Text = data.Name;
    ...
    return cell;
}
```

Cell reuse in action

- Cells are created initially for the first screen of data based on the reuse identifier passed to `DequeueReusableCell`
- Once a full "screen" of data is present (plus one or two edge cells), cells are reused as you scroll
- Keeps the number of in-memory objects to a minimum, and reduces allocs and deallocs



Group Exercise

Implement cell reuse

Summary

1. What is cell reuse?
2. Participating in cell reuse
3. Managing the cell data

Next Steps

- This class has shown you how to add a Table View into your application and efficiently populate it with data
- iOS115 looks at further customizations of the Table View
- iOS215 examines adding editing interactions to the TableView

Thank You!