# Still busy!

- **Autolayout**
- **ScrollView**
- **TableView**
- **Multithreading**

*... and once again plenty of demos!*

# Autolayout

# Autolayout

- **Autolayout so far:**
  - Using the dashed blue lines to try to tell Xcode what you intend
  - Ctrl-Dragging between views to create relationships (spacing, etc.)
  - The "Pin" and "Arrange" popovers in the lower right of the storyboard
  - Reset to Suggested Constraints (if the blue lines were enough to unambiguously set constraints)
  - Document Outline (see all constraints, resolve misplacements and even conflicts)
  - Size Inspector (look at (and edit!) the details of the constraints on the selected view)
  - Clicking on a constraint to select it then bring up Attributes Inspector (to edit its details)

- **Mastering Autolayout requires experience, practice!**
- **Autolayout can be done from code to (you're probably better off doing it in the storyboard wherever possible)**

# Autolayout

- **What about rotation?**
  - Sometimes rotating changes the geometry so drastically that autolayout is not enough
  - You actually need to reposition the views to make them fit properly
- **RPNCalc**
  - For example, what if we had 20 buttons in out RPN Calculator?
  - It might be better in Landscape to have the buttons 5 across and 4 down versus in Portrait have them 4 across and 5 down
- **View Controllers might want this in other situations too**
  - For example, your MVC is the master of a side-by-side split view
  - In that case, you'd want to draw just like a Portrait iPhone does
- **The solution? Size Classes**
  - Your View Controller always exists in a certain "size class" environment for width and height
  - Currently this is either Compact or Regular (i.e. not compact)

# Autolayout

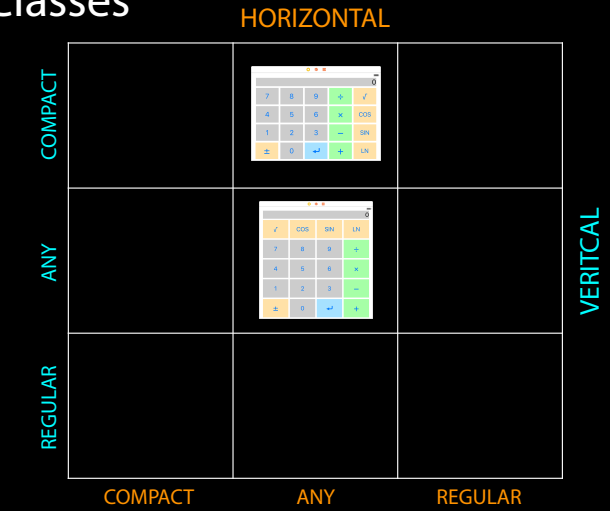
- **iPhone Plus**
  - The iPhone Plus in Portrait orientation is Compact in width and Regular in height
  - In Landscape, it is Compact in height and Regular in width
- **iPhone**
  - Other iPhones in Portrait are also Compact in width and Regular in height
  - But in Landscape, non-plus iPhones are treated as Compact in both dimensions
- **iPad**
  - Always Regular in both dimensions
  - An MVC that is the master in a side-by-side split view will be Compact width, Regular height
- **Extensible**
  - This whole concept is extensible to any "MVC's inside other MVC's" situation (not just split view)
  - An MVC can find out its size class environment via this method in UIViewController …

```
let mySizeClass: UIUserInterfaceSizeClass = self.traitCollection.horizontalSizeClass
```

  - The return value is an enum .Compact or .Regular (or .Unspecified)

# Size Classes


# Size Classes




# ScrollView

# UIScrollView

- What to do when a view is just "too big" to fit on screen?
- Need to pan and zoom without having to implement gestures
- Enter **UIScrollView**: UIView subclass
  - Pans and zooms around a predefined size containing its subviews
  - Two important subclasses of it: **UITableView**, **UITextView**

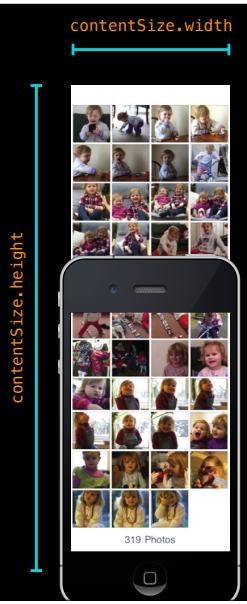

# UIScrollView

```
var contentSize: CGSize
```

- The frame of each of the UIScrollView's subviews is relative to this predefined space.
- A subview with a frame with an origin at (0, 0) would be in the upper left of this space.
- Usually the UIScrollView only has one subview which fills the entire space, i.e. that subview's frame is usually:
  - origin = (0, 0)
  - size = scrollView.contentSize


# UIScrollView

```
var contentSize: CGSize
```

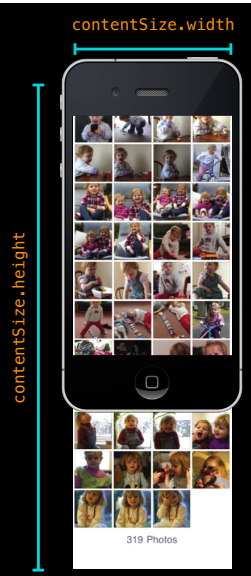



# UIScrollView

```
var contentSize: CGSize
```

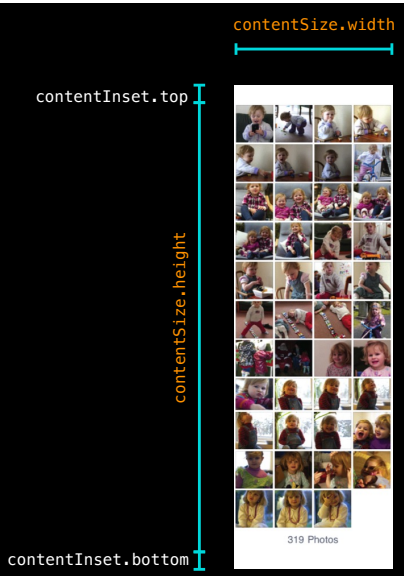
# UIScrollView

```
var contentSize: CGSize
```



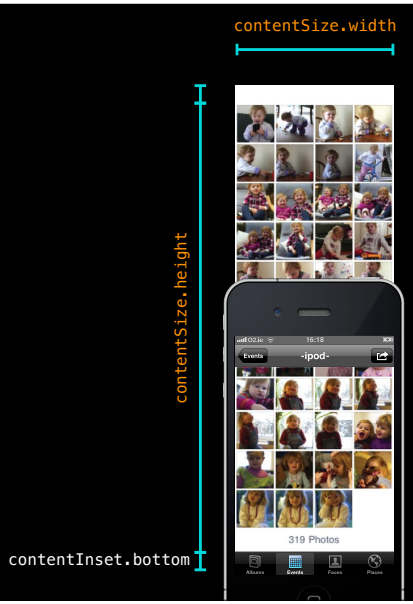

# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets

typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



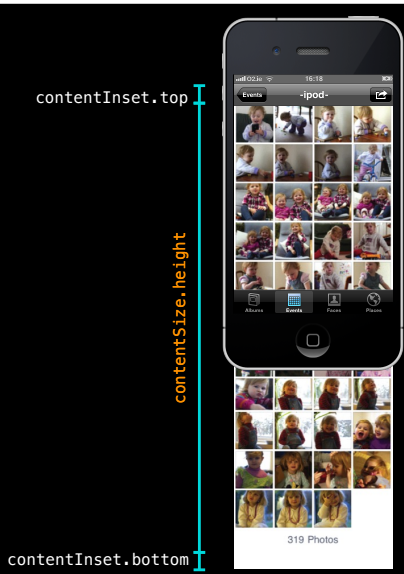

# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets

typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



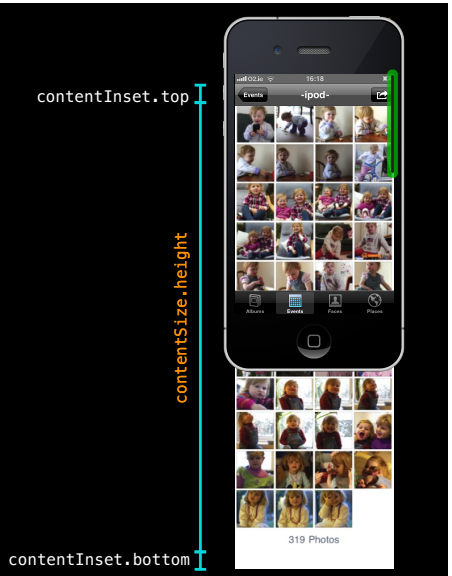

# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets

typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



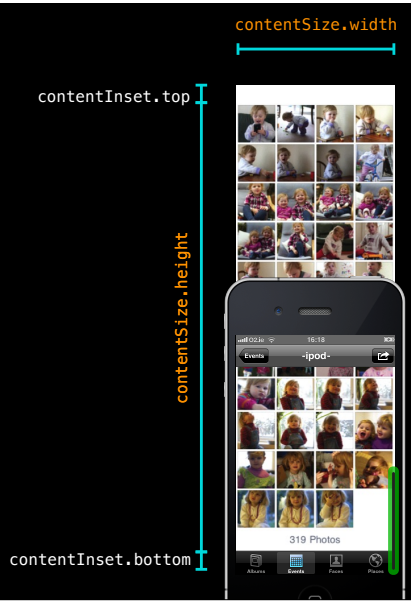
# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets

typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



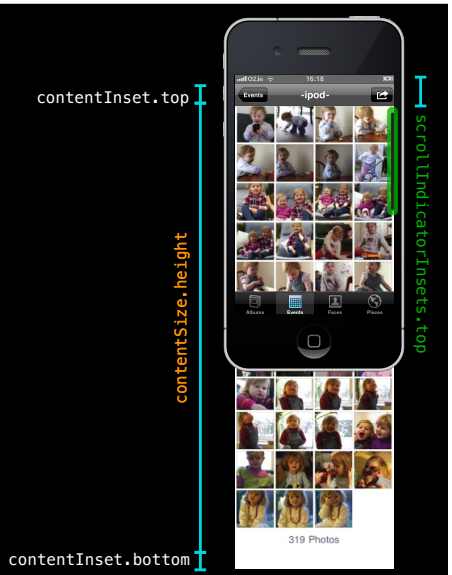

# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets

typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



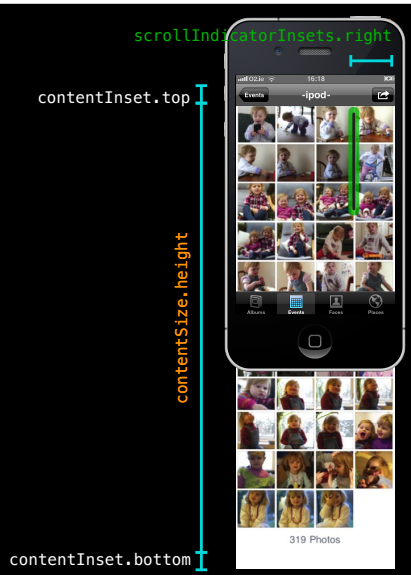

# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets
var scrollIndicatorInsets UIEdgeInsets
typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```




# UIScrollView

```
var contentSize: CGSize
var contentInset: UIEdgeInsets
var scrollIndicatorInsets UIEdgeInsets
typedef struct UIEdgeInsets {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```



# Normal UIView

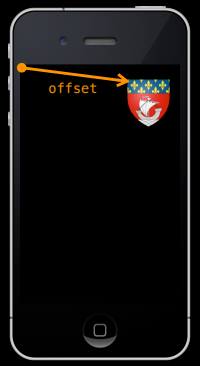
- **Adding subviews to a normal UIView**

```
subview.frame = CGRect(x: offset_x, y: offset_y, width: width, height: height)
view.addSubview(subview)
```

# Normal UIView

- **Adding subviews to a normal UIView**

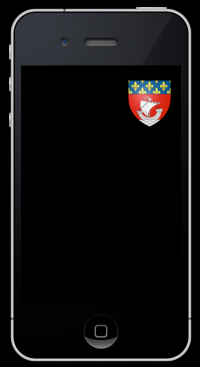
```
subview.frame = CGRect(x: offset_x, y: offset_y, width: width, height: height)
view.addSubview(subview)
```




# Normal UIView

- **Adding subviews to a normal UIView**

```
subview.frame = CGRect(x: offset_x, y: offset_y, width: width, height: height)
view.addSubview(subview)
```



# Normal UIView

- **Adding subviews to a normal UIView**

```
subview.frame = CGRect(x: 0, y: 0, width: width, height: height)
view.addSubview(subview)
```

# Normal UIView

- **Adding subviews to a normal UIView**

```
subview.frame = CGRect(x: offset_x, y: offset_y, width: width, height: height)
view.addSubview(subview)
```

# UIScrollView

- **Adding subviews to a UIScrollView**

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```

# UIScrollView

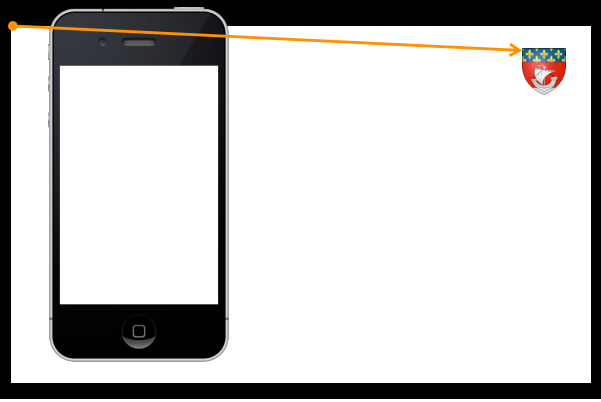
- **Adding subviews to a UIScrollView**

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```


# UIScrollView

- **Adding subviews to a UIScrollView**

```
scrollView.contentSize = CGSize(width: 2200, height: 1400)
subview1.frame = CGRect(x: 2000, y: 200, width: 80, height: 80)
view.addSubview(subview1)
```



# UIScrollView

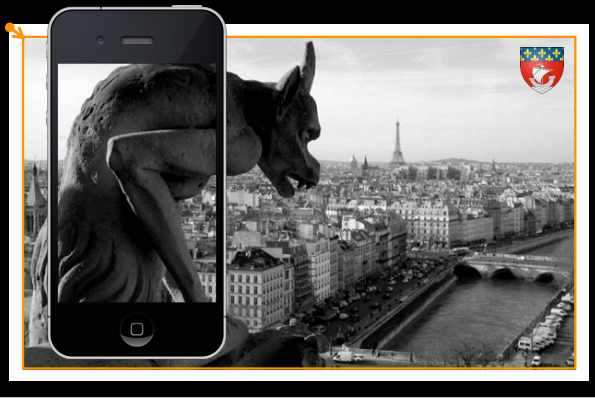
- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSize(width: 2200, height: 1400)
subview1.frame = CGRect(x: 2000, y: 200, width: 80, height: 80)
view.addSubview(subview1)
```

# UIScrollView

- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSize(width: 2200, height: 1400)
subview2.frame = CGRect(x: 100, y: 100, width:2000, height: 1200)
view.addSubview(subview2)
```



# UIScrollView

- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```

# UIScrollView

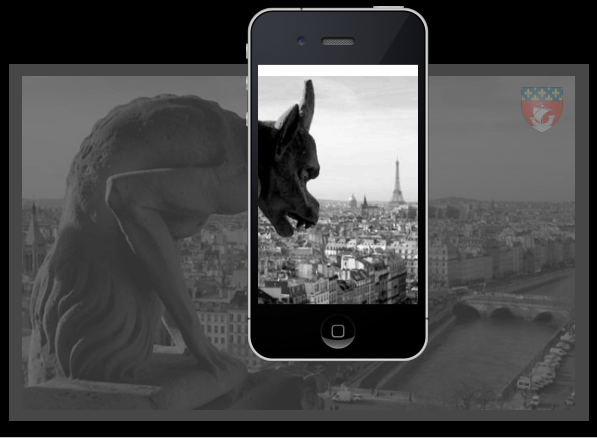
- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```

# UIScrollView

- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```




# UIScrollView

- Adding subviews to a UIScrollView

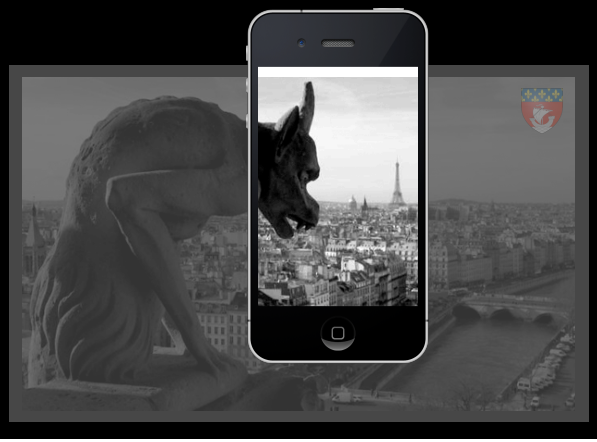
```
scrollView.contentSize = CGSizeMake(2200, 1000);
```



# UIScrollView

- Adding subviews to a UIScrollView

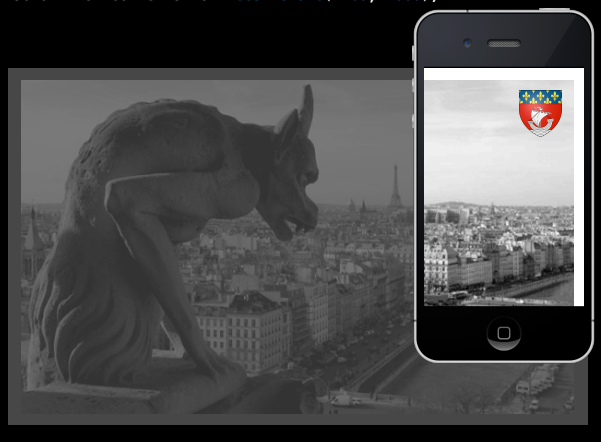
```
scrollView.contentSize = CGSizeMake(2200, 1000);
```


# UIScrollView

- Adding subviews to a UIScrollView

```
scrollView.contentSize = CGSizeMake(2200, 1000);
```

# UIScrollView

- **Adding subviews to a UIScrollView**

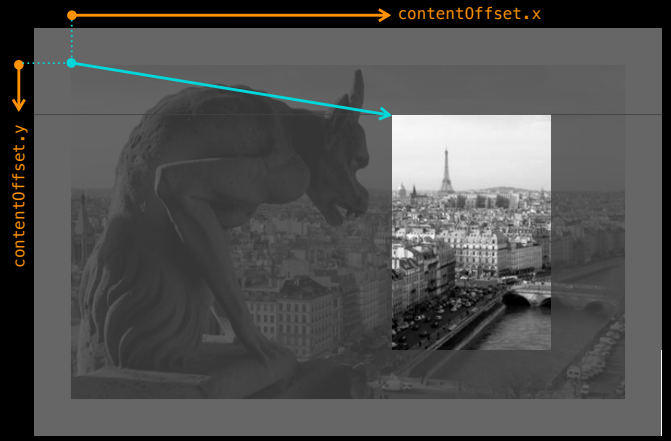
```
scrollView.contentSize = CGSizeMake(2200, 1000);
```

# UIScrollView


# UIScrollView

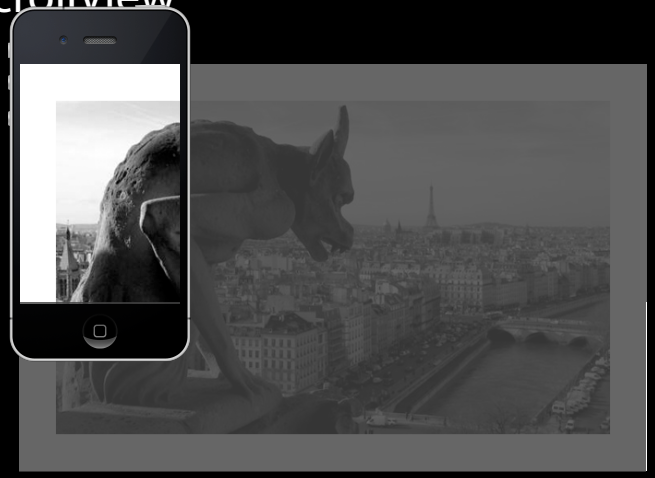
```
var contentOffset: CGSize
```

# UIScrollView




```
var contentOffset: CGSize
```

# UIScrollView



var contentOffset: CGSize

# UIScrollView


var contentOffset: CGSize

# UIScrollView

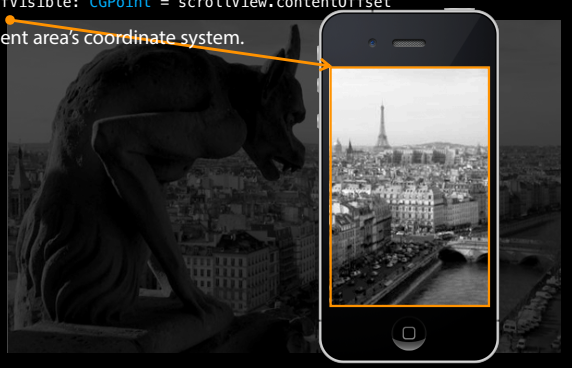
scrollView.bounds

# UIScrollView

- **Where in the content is the scroll view currently positioned?**

`let upperLeftOfVisible: CGPoint = scrollView.contentOffset`

  - in the content area's coordinate system.

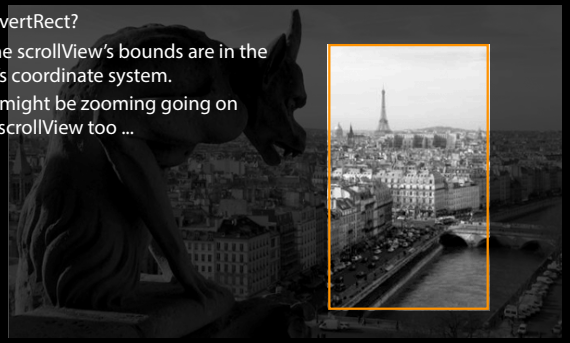


# UIScrollView

- **What area in a subview is currently visible?**

```
let visibleRect: CGRect = subview.convertRect(scrollView.bounds, fromView: scrollView)
```

- Why the convertRect?
  - Because the scrollView's bounds are in the scrollView's coordinate system.
  - And there might be zooming going on inside the scrollView too ...



# UIScrollView

- **How do you create a UIScrollView?**
  - Just like any other UIView. Drag out in a storyboard or use UIScrollView(frame: rect)
  - Or select a UIView in your storyboard and choose "Embed In > Scroll View" from Editor menu.
- **Or add your really large UIView using addSubview:**
  - Filling the entire screen (application:didFinishLaunchingWithOptions:)

```
if let image = UIImage(named: "bigimage.jpg")
{
    let imageView = UIImageView(image: image)
    scrollView.addSubview(imageView)
}
```

  - Add more subviews if you want.
  - All of the subviews' frames will be in the UIScrollView's content area's coordinate system (that is, (0,0) in the upper left & width/height of contentSize.width/height).
- **Don't forget to set the content size (the scroll view's scrollable area's size)**
  - Common bug is to do the above 3 lines of code (or embed in Xcode) and forget to say:

```
scrollView.contentSize = imageView.bounds.size
```

# Scrolling in a UIScrollView

- **Scrolling programmatically**

```
scrollView.scrollRectToVisible(CGRect, animated: Bool)
```

- **Other things you can control in a scroll view**
  - Whether scrolling is enabled: var scrollEnabled: Bool
  - Locking scroll direction to user's first "move": `var directionalLockEnabled: Bool`
  - The style of the scroll indicators `var indicatorStyle: UIScrollViewIndicatorStyle` (call `flashScrollIndicators` when your scroll view appears).
  - Whether the actual content is "inset" from the scroll view's content area (`contentInset` property).

# Zooming in a UIScrollView

- **Zooming**
  - All UIView's have a property (transform) which is an affine transform (translate, scale, rotate).
  - Scroll view just modifies this transform when you zoom.
  - Zooming is also going to affect the scroll view's contentSize and contentOffset.
- **Will not work without minimum/maximum zoom scale being set**

```
scrollView.minimumZoomScale = 0.5 // 0.5 means half its normal size
scrollView.maximumZoomScale = 2.0 // 2.0 means twice its normal size
```

- **Will not work without delegate method to specify view to zoom**

```
func viewForZoomingInScrollView(sender:UIScrollView) -> UIView
```

  - If your scroll view only has one subview, you return it here. More than one? Up to you.
- **Another delegate method will notify you when zooming ends**

```
func scrollViewDidEndZooming(UIScrollView, withView: UIView?, atScale: CGFloat)
```

  - If you redraw your view at the new scale, be sure to reset the affine transform back to identity.

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...

```
scrollView.zoomScale = 1.0
```

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...

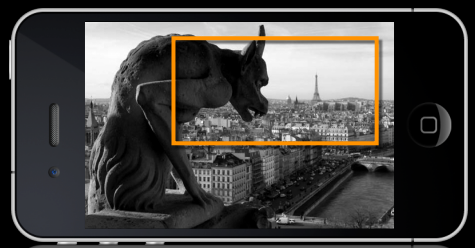
```
scrollView.zoomScale = 1.2
```

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...



```
scrollView.zoomToRect(CGRect, animated: Bool)
```

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...

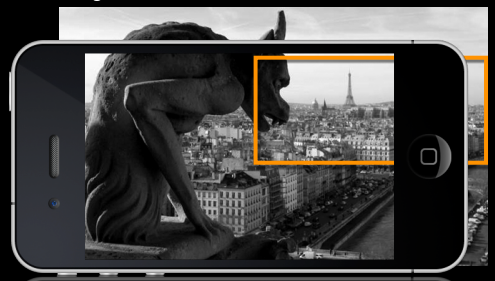
```
scrollView.zoomToRect(CGRect, animated: Bool)
```

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...



```
scrollView.zoomToRect(CGRect, animated: Bool)
```

# Zooming in a UIScrollView

- **Zooming programatically**

```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

  - Best explained with a few images...

```
scrollView.zoomToRect(CGRect, animated: Bool)
```

# UIScrollView Delegate Methods

- **Lots and lots of delegate methods!**
  - The scroll view will keep you up to date with what's going on.
- **Other delegate methods**

```
func scrollViewDidScroll(UIScrollView)
func scrollViewDidScrollToTop(UIScrollView)
func scrollViewWillBeginZooming(UIScrollView, withView: UIView?)
// if you redraw at the new scale, ensure affine transform is reset to identity
func scrollViewDidEndScrollingAnimation(UIScrollView)
func scrollViewWillBeginDecelerating(UIScrollView)
```

- **Methods in UIScrollView to find out scrolling state**

```
var zooming: Bool { get }
var dragging: Bool { get }
var tracking: Bool { get }
var tracking: Bool { get }
```

# TableView

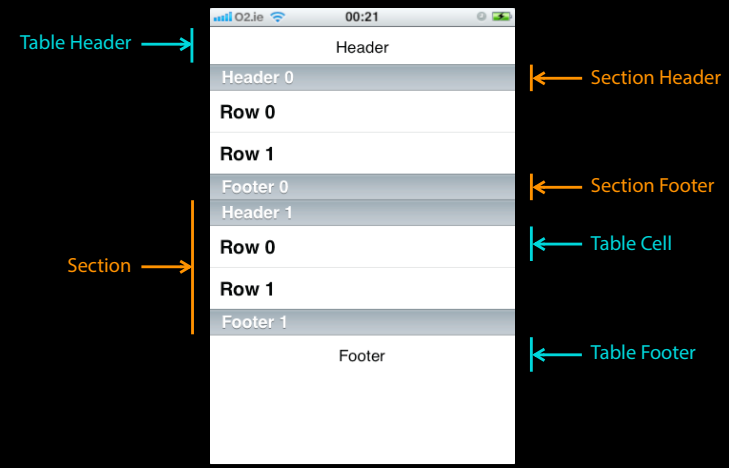
# UITableView

- **Very important class for displaying data in a table**
  - 1D table.
  - It's a subclass of UIScrollView.
  - Table can be static or dynamic (i.e. a 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.
- **Displaying multi-dimensional tables**
  - Usually done via a UINavigationController containing multiple MVC's where View is UITableView
- **Two styles**
  - `UITableViewStylePlain` or `UITableViewStyleGrouped`
  - Static or Dynamic
  - Divided into sections or not
  - Different formats for each row in the table (including completely customized)

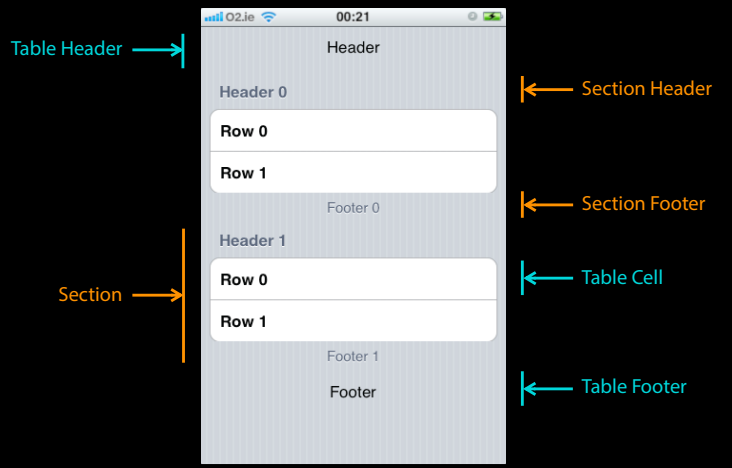
# UITableView


`UITableViewStylePlain` `UITableViewStyleGrouped`

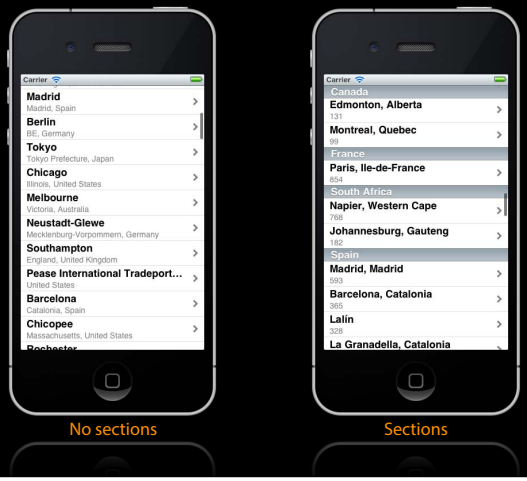
# UITableViewStylePlain




# UITableViewStyleGrouped

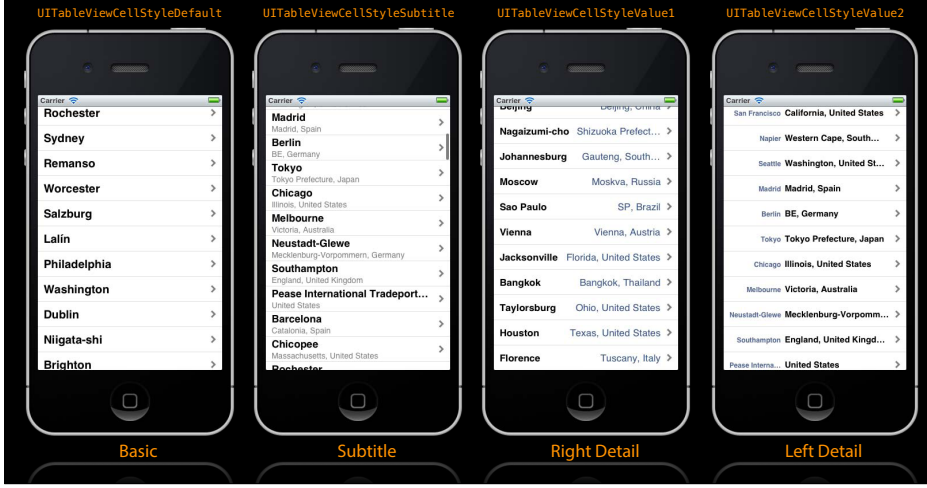


# UITableView and Sections




No sections Sections

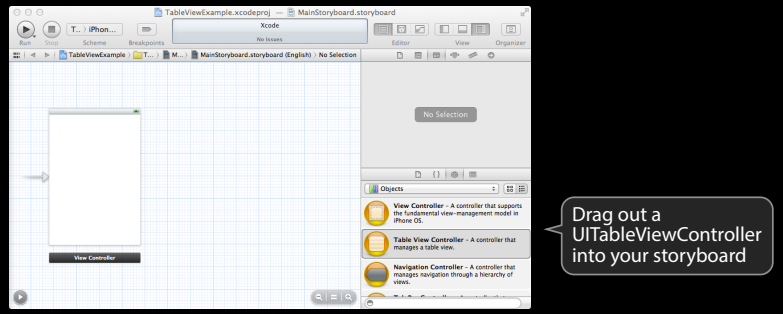
# UITableView and Cell Types




Basic Subtitle Right Detail Left Detail

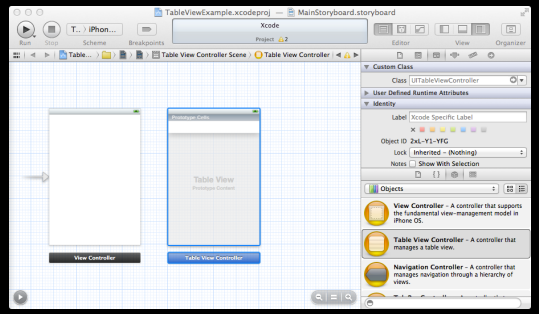
# Creating UITableViewController

- **UITableViewController**
  - iOS class used as the base class for MVC's that display UITableViews
  - Just drag one out in Xcode, create a subclass of it and you're on your way!



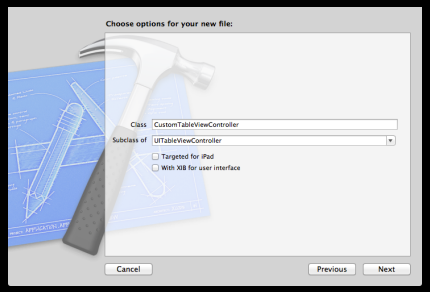

# Creating UITableViewController

- **UITableViewController**
  - iOS class used as the base class for MVC's that display UITableViews
  - Just drag one out in Xcode, create a subclass of it and you're on your way!



# Creating UITableViewController

- **UITableViewController**
  - iOS class used as the base class for MVC's that display UITableViews
  - Just drag one out in Xcode, create a subclass of it and you're on your way!

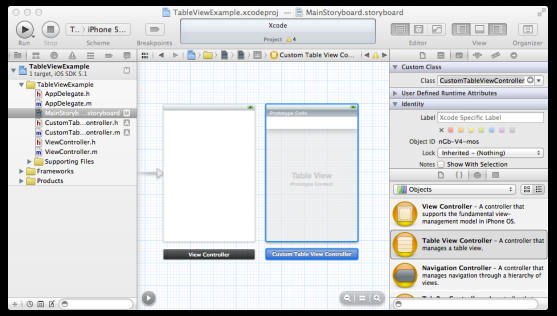



Choose New File ... from the File menu to create a custom subclass of UITableViewController

Be sure to set the superclass to UITableViewController

# Creating UITableViewController

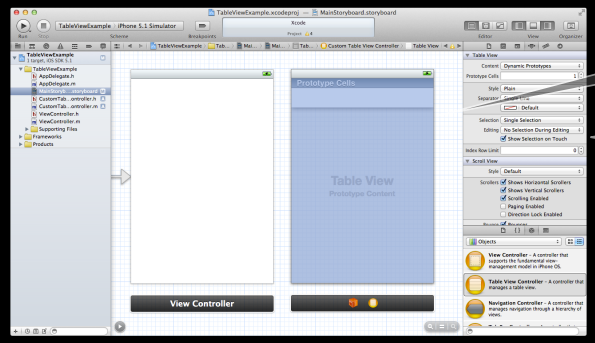
- **UITableViewController**
  - iOS class used as the base class for MVC's that display UITableViews
  - Just drag one out in Xcode, create a subclass of it and you're on your way!



And select your custom class in your storyboard

# Creating UITableViewController

- **UITableView**
  - You can customize both the look of the table view and its cells from Xcode.
  - Click on the table view (not the table view controller) to see its properties in the Inspector.

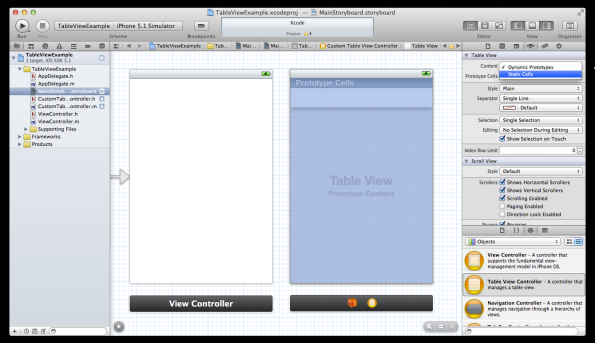



Click here to select the UITableView inside the UITableVC's View...

...then you should see table view properties in the Attributes Inspector.

# Creating UITableViewController

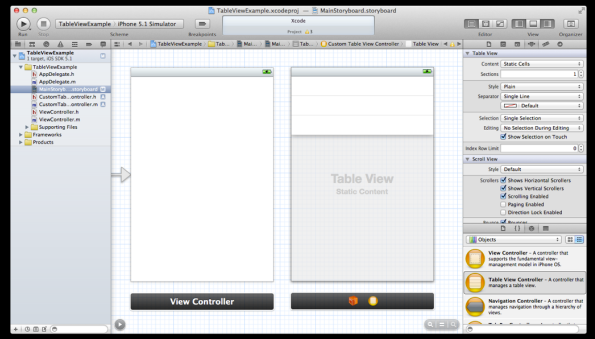
- **UITableView**
  - You can customize both the look of the table view and its cells from Xcode.
  - Click on the table view (not the table view controller) to see its properties in the Inspector.




more on dynamic prototypes in a minute, but for now, switch to a Static Table View.

# Creating UITableViewController

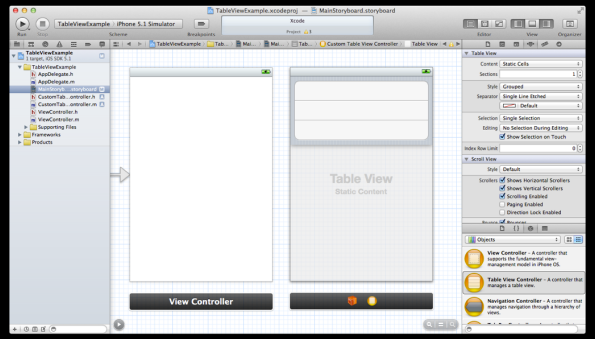
- **UITableView**
  - You can customize both the look of the table view and its cells from Xcode.
  - Click on the table view (not the table view controller) to see its properties in the Inspector.




This table view has the Plain style. Let's change it to Grouped.


# Creating UITableViewController

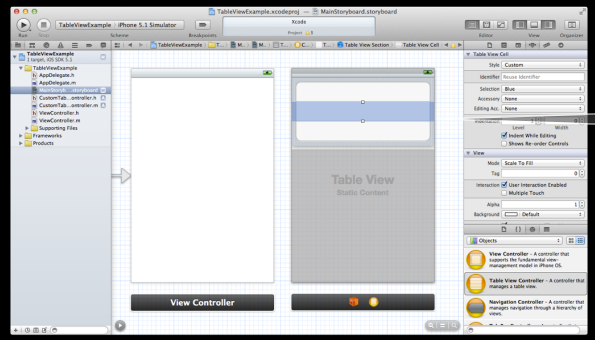
- **UITableView**
  - You can customize both the look of the table view and its cells from Xcode.
  - Click on the table view (not the table view controller) to see its properties in the Inspector.


This table view has the Grouped style.

# Creating UITableViewController

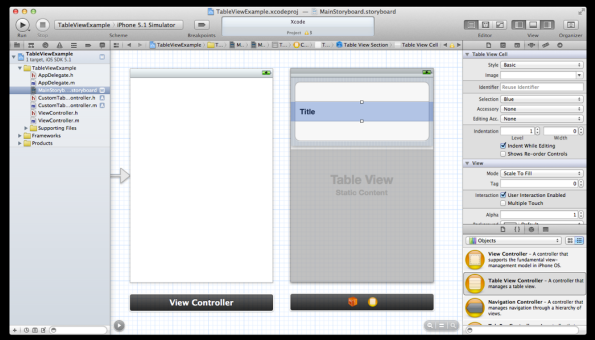
- **UITableViewCell**
  - And you can change the look of each cell as well.
  - Click on a cell that you want to change and set its attributes in the Inspector.




Click to select the cell whose properties we want to edit


# Creating UITableViewController

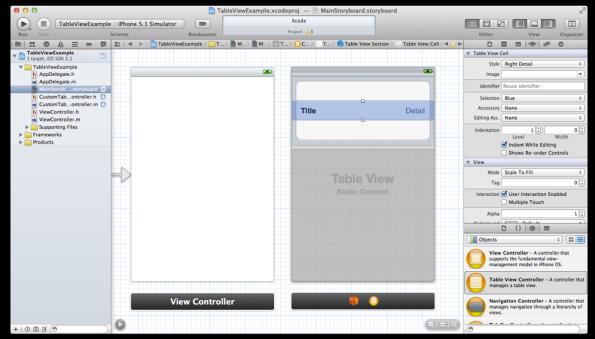
- **UITableViewCell**
  - And you can change the look of each cell as well.
  - Click on a cell that you want to change and set its attributes in the Inspector.


Change style to Basic

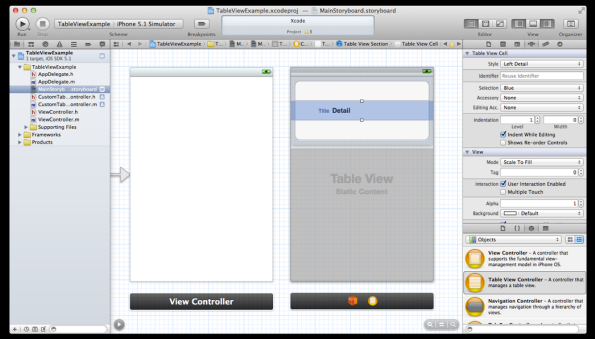
# Creating UITableViewController

- **UITableViewCell**
  - 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.



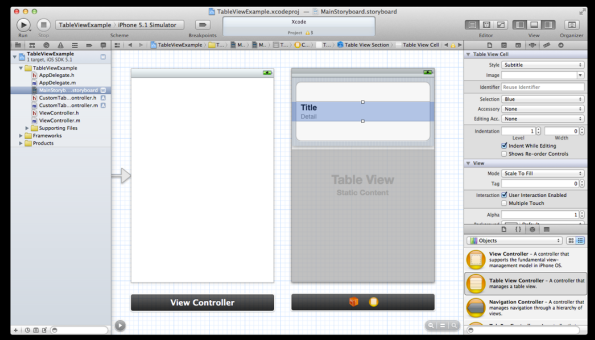

# Creating UITableViewController

- **UITableViewCell**
  - 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.



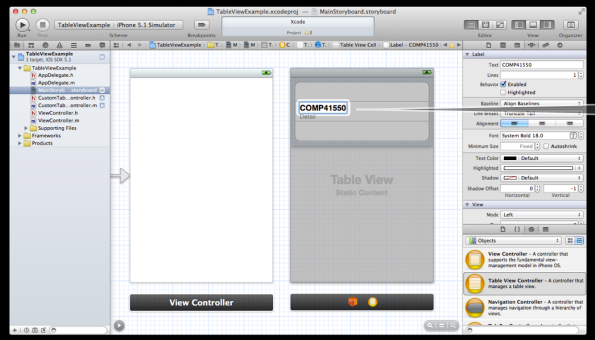

# Creating UITableViewController

- **UITableViewCell**
  - 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.



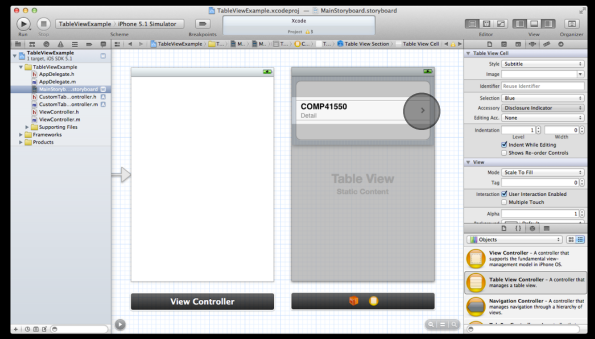

# Creating UITableViewController

- **UITableViewCell**
  - 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 UITableViewController

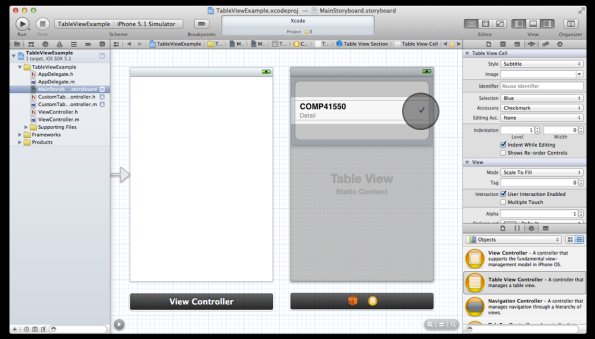
- **UITableViewCell**
  - And you can change the look of each cell as well.
  - Click on a cell that you want to change and set its attributes in the Inspector.




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

# Creating UITableViewController

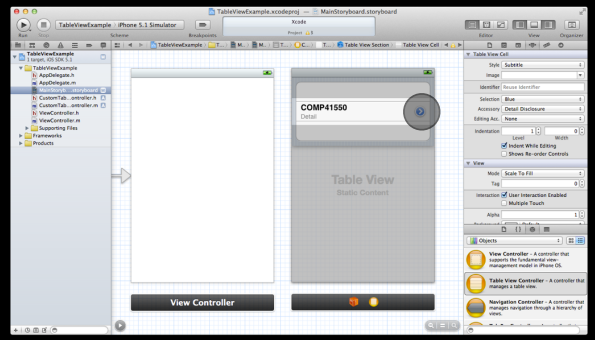
- **UITableViewCell**
  - And you can change the look of each cell as well.
  - Click on a cell that you want to change and set its attributes in the Inspector.




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

# Creating UITableViewController

- **UITableViewCell**
  - 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.

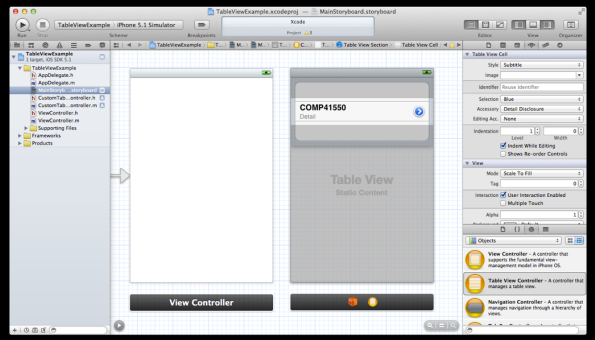



Show detail disclosure accessory.
This is an active control. Use it to show ancillary info. Clicking on the row should still do the "main thing" for this row.

# Creating UITableViewController

- **UITableViewCell**
  - When user taps on the blue detail disclosure, it will be sent to your UITableViewController:

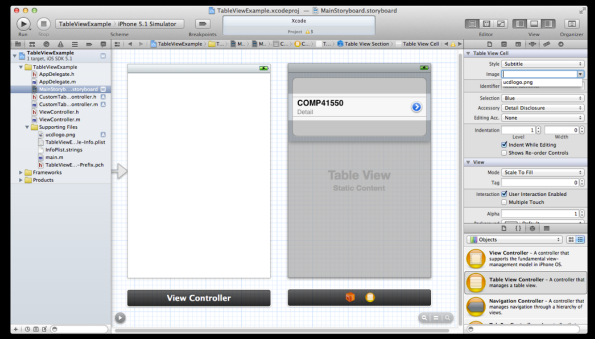
```
func tableView(UITableView, accessoryButtonTappedForRowWithIndexPath: NSIndexPath)
```




Show detail disclosure accessory.
This is an active control. Use it to show ancillary info. Clicking on the row should still do the "main thing" for this row.

# Creating UITableViewController

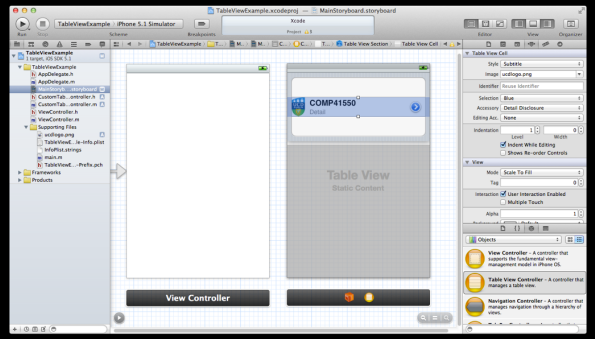
- **UITableViewCell**
  - Notice that some cell styles can have an image.
  - You can set this in the code as well (more in a moment on this).




Set Cell Image

# Creating UITableViewController

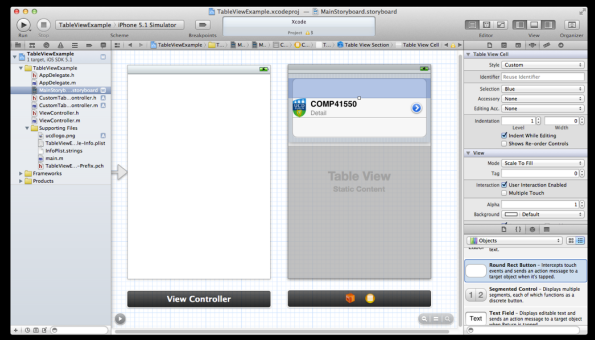
- **UITableViewCell**
  - Notice that some cell styles can have an image.
  - You can set this in the code as well (more in a moment on this).




Set Cell Image

# Creating UITableViewController

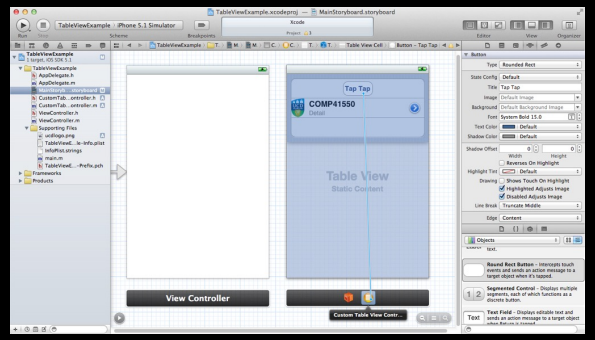
- **Custom UITableViewCell**
  - In the Custom style, you can drag out views and wire them up as outlets!




Drag a button

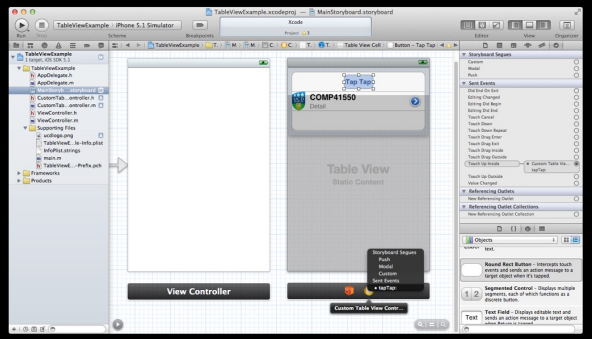
# Creating UITableViewController

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



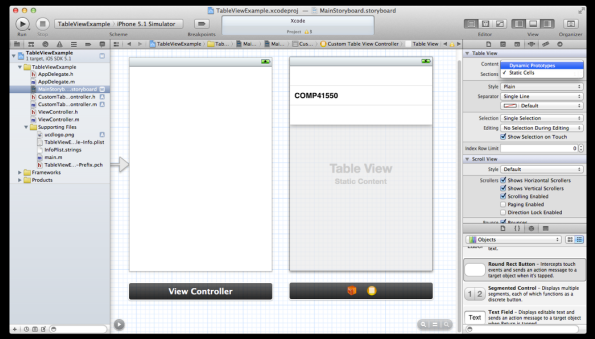
# Creating UITableViewController

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




# Creating UITableViewController

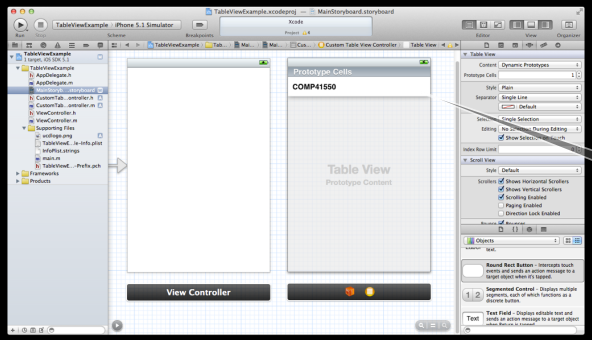
- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - If you switch to dynamic mode, then the cell you edit is a "prototype" for all cells in the list.




Switch to a Dynamic Table with Prototype Cells

# Creating UITableViewController

- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - If you switch to dynamic mode, then the cell you edit is a "prototype" for all cells in the list.

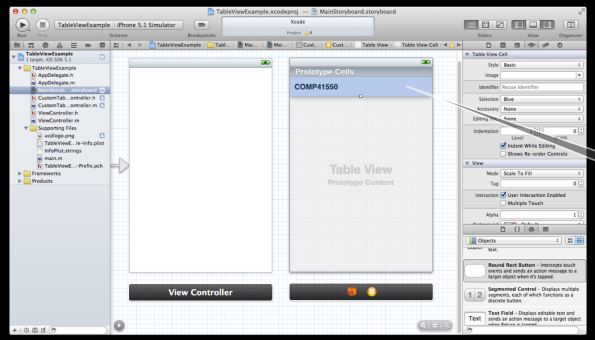



Now click on the Prototype to edit it.

All cells in this table will be like this Prototype (though we'll set the contents to be different in code)

# Creating UITableViewController

- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - If you switch to dynamic mode, then the cell you edit is a "prototype" for all cells in the list.



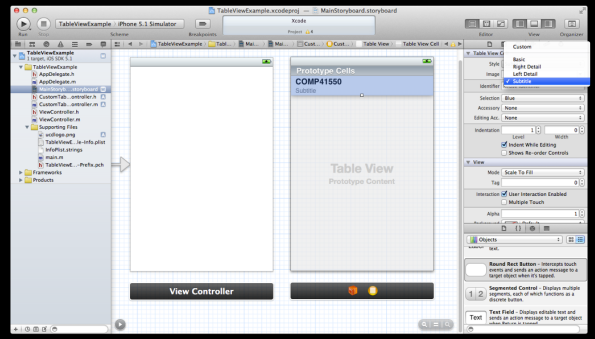

You should see Table View Cell properties appear in the Attributes Inspector

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 UITableViewController

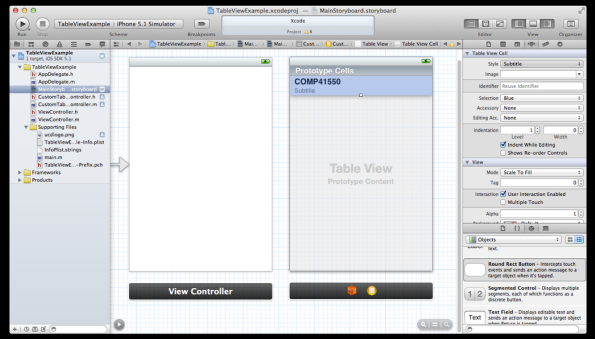
- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - If you switch to dynamic mode, then the cell you edit is a "prototype" for all cells in the list.




Let's change the Prototype's style to be Subtitle, for example

# Creating UITableViewController

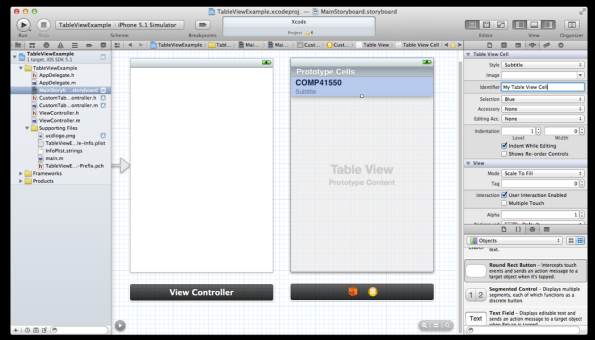
- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - If you switch to dynamic mode, then the cell you edit is a "prototype" for all cells in the list.




This 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 UITableViewController

- **Dynamic UITableViewCell**
  - All of the above examples were "static" cells (setup in the storyboard). Not a dynamic list.
  - 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.

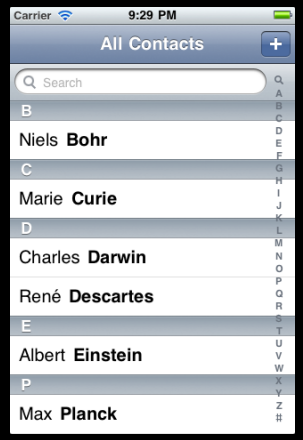
# UITableView

- **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 what is displayed inside the cells.
- **UITableViewController is automatically set as the delegate & dataSource.**
- **A UITableViewController subclass also have a property pointing to**
  - Delegated. Your MVC's Controller is almost always the dataSource for the UITableView.

```
var tableView: UITableView
```

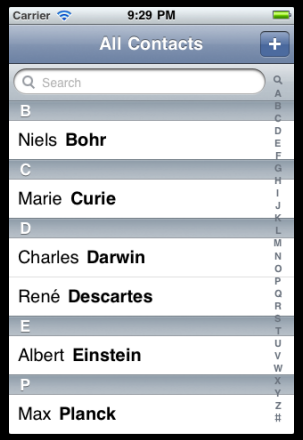
- **To be "dynamic," we need to be the UITableView's dataSource**
  - Three important methods in this protocol:
  1. How many sections in the table?
  2. How many rows in each section?
  3. Give me a UIView to use to draw each cell at a given row in a given section.

# Dynamic UITableView




DataSource

# Dynamic UITableView

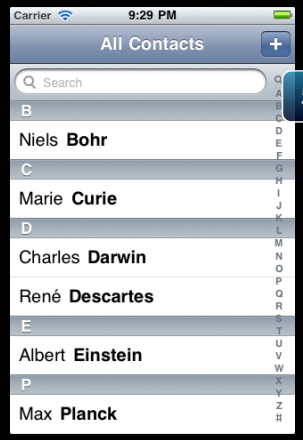



`numberOfSectionsInTableView()`

How Many Sections?

DataSource

# Dynamic UITableView

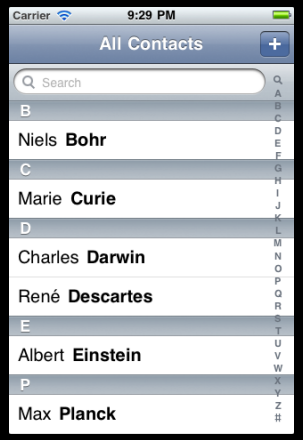



5

`numberOfSectionsInTableView()`

DataSource

# Dynamic UITableView

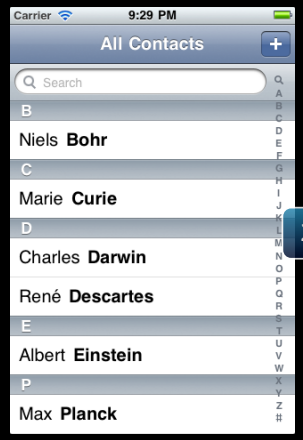



`tableView(UITableView, numberOfRowsInSection: Int)`

How many rows in section 2?

DataSource

# Dynamic UITableView

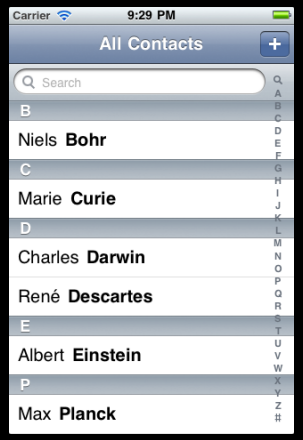



2

`numberOfSectionsInTableView()`

DataSource

# Dynamic UITableView

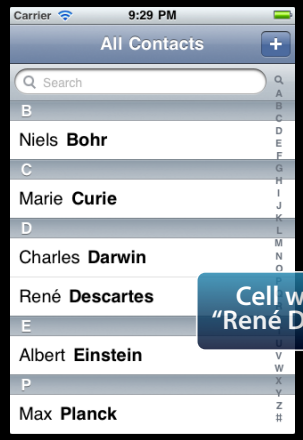



`tableView(UITableView, cellForRowAtIndexPath: NSIndexPath)`

What to display at section 2, row 1?

DataSource

# Dynamic UITableView




`tableView(UITableView, cellForRowAtIndexPath: NSIndexPath)`

DataSource

Cell with text "René Descartes"

# 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 (a UIView subclass).
  - Method to get that cell for a given row in a given section:

```
func tableView(tv: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{



}
```

NSIndexPath is just an object with two important properties for use with UITableView: row and section

# 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 (a UIView subclass).
  - Method to get that cell for a given row in a given section:

```
func tableView(tv: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    let data = dataModel[indexPath.section][indexPath.row]
    let dequeued: AnyObject = tv.dequeueReusableCellWithIdentifier("MyCell", forIndexPath: indexPath
}
```

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 (a UIView subclass).
  - Method to get that cell for a given row in a given section:

```
func tableView(tv: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    let data = dataModel[indexPath.section][indexPath.row]
    let dequeued: AnyObject = tv.dequeueReusableCellWithIdentifier("MyCell", forIndexPath: indexPath
}
```

- 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 this dequeue method will return nil (let's deal with that ...).

# 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 (a UIView subclass).
  - Method to get that cell for a given row in a given section:

```
func tableView(tv: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    let data = dataModel[indexPath.section][indexPath.row]
    let dequeued: AnyObject = tv.dequeueReusableCellWithIdentifier("MyCell", forIndexPath: indexPath
    let cell = dequeued as UITableViewCell
    cell.textLabel?.text = data.title
    cell.detailTextLabel?.text = data.subtitle
    return cell
}
```

Also set properties on cell

# 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 (a UIView subclass).
  - Method to get that cell for a given row in a given section:

```
func tableView(tv: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    let data = dataModel[indexPath.section][indexPath.row]
    let dequeued: AnyObject = tv.dequeueReusableCellWithIdentifier("MyCell", forIndexPath: indexPath
    let cell = dequeued as UITableViewCell
    cell.textLabel?.text = data.title
    cell.detailTextLabel?.text = data.subtitle
    return cell
}
```

There are obviously other things you can do in the cell besides setting its text e.g. detail text, image, checkmark...

# UITableViewDataSource

- **How does a dynamic table know how many rows there are?**
  - And how many sections, too, of course?

```
func numberOfSectionsInTableView(UITableView) -> Int
func tableView(UITableView, numberOfRowsInSection: Int) -> Int
```

- **Number of sections is 1 by default**
  - In other words, if you don't implement numberOfSectionsInTableView, it will be 1.
- **No default for number of rows in a section**
  - This is a REQUIRED method in this protocol (as is tableView(_:numberOfRowsInSection:)).
- **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**
  - They are mostly about getting the headers and footers for sections.
  - And about dealing with editing the table (moving/deleting/inserting rows)

# UITableViewDelegate

- **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).
- **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.

# TableView Target/Action

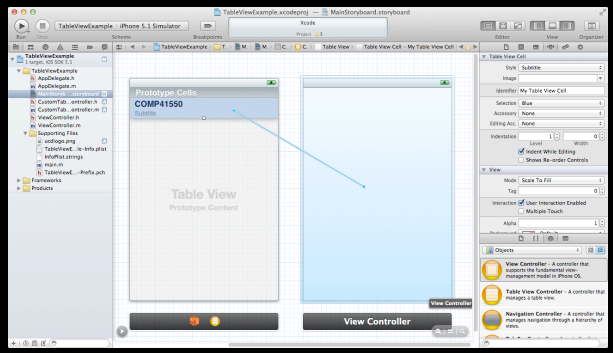
- **UITableViewDelegate method sent when row is selected**
  - This is sort of like "table view target/action"
  - You might use this to update a detail view in a split view if master is a table view

```
func tableView(UITableView, didSelectRowAtIndexPath: NSIndexPath) {
    // go do something based on information
    // about my data structure corresponding to indexPath.row in indexPath.section
}
```

- **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.

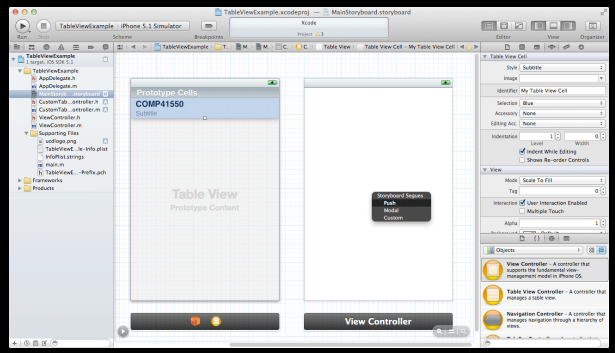
# TableView Segues

- **You can segue when a row is touched, just like from a button**
  - Segues will call prepareForSegue(_:sender:) with the chosen UITableViewCell as sender.
  - You can tailor whatever data the MVC needs to whichever cell was selected.



# TableView Segues

- **You can segue when a row is touched, just like from a button**
  - Segues will call prepareForSegue(_:sender:) with the chosen UITableViewCell as sender.
  - You can tailor whatever data the MVC needs to whichever cell was selected.




# TableView Segues

- **You can segue when a row is touched, just like from a button**
  - Segues will call prepareForSegue(_:sender:) with the chosen UITableViewCell as sender.
  - You can tailor whatever data the MVC needs to whichever cell was selected.
- **This works whether dynamic or static**

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    let indexPath = self.tableView.indexPathForCell(sender as UITableViewCell)
    // prepare segue.destinationController to display based on information
    // about my data structure corresponding to indexPath.row in indexPath.section
}
```

# UITableView

- **What if your Model changes?**

```
func reloadData()
```

  - Causes the table view to call numberOfSectionsInTableView() and numberOfRowsInSection() all over again and then cellForRowAtIndexPath() on each visible cell.
  - Relatively heavyweight 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 ...

```
func reloadRowsAtIndexPaths(indexPaths: [NSIndexPath], withRowAnimation: 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.

# Demo

*Spirograph Artwork*

# Demo

- **Replace FavoritesVC by Table View Controller**

# Multithreading

# Closures

- **Capturing**
  - Closures "capture" variables in the surrounding context
  - That means that it keeps those variables around as long as the closure stays around
  - You can even make assignments to the variables or modify what they point to
  - This can lead to some very elegant code…
- **Interesting use of a closure …**
  - It might be that sometimes using a closure is a better tool than delegation.
  - For example, consider the following code ...

```
class Grapher {
    var plot: ((x: Double) -> Double?)?
}

let grapher = Grapher()
let calcModel = CalcModel()
calcModel.session = expressionToGraph
grapher.plot = { (x: Double) -> Double? in
    calcModel.valueForVar["X"] = x
    return calcModel.evalStack() // gets captured and reused each time plot is called
}
```

# Closures

- **Caveat**
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - This will display the correct output 1 2 3 4 5 6 7 8 9 10

# Closures

- Caveat
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - This will display the correct output 1 2 3 4 5 6 7 8 9 10

# Closures

- Caveat
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - It captured that x for as long as this closure is around

# Closures

- Caveat
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - It makes sure self stays around so we can call show

# Closures

- Caveat
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - But what's not so cool is that self points to this closure (via its action property).
  - Neither can now ever leave the heap (they point to each other).

# Closures

- **Caveat**
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - To fix this, we need to tell the closure not to keep that self in memory

# Closures

- **Caveat**
  - We have to be a little bit careful about capturing because of memory management
  - Specifically, we don't want to create a memory cycle
  - Closures capture pointers (i.e. it keeps what they point to in memory)
  - If a captured pointer points (directly or indirectly) back at the closure, that's a problem
  - Because now there will always be a pointer to the closure and to the captured thing
  - Neither will ever be able to leave the heap
- **Example:**

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { print("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { [unowned self] in x = x + 1; self.show(x) }
    }
    func doMyAction10times() { for i in 1...10 { action() } }
}
```

  - Now that reference to self inside the closure will not keep self in memory
  - That self will be kept in memory as long as someone else has a pointer to it

# Multithreading

- **Queues**
  - Multithreading is mostly about "queues" in iOS
  - Functions (usually closures) are lined up in a queue
  - Then those functions are pulled off the queue and executed on an associated thread
- **Main Queue**
  - There is a very special queue called the "main queue"
  - All UI activity MUST occur on this queue and this queue only
  - And, conversely, non-UI activity that is at all time consuming must NOT occur on that queue
  - We want our UI to be responsive
  - Functions are pulled off and worked on in the main queue only when it is "quiet"
- **Other Queues**
  - Mostly iOS will create these for us as needed
  - We'll give a quick overview of how to create your own (but usually not necessary)

# Multithreading

- **Executing a function on another queue**

```
let queue: dispatch_queue_t = <the Q you want>
dispatch_async(queue) {
    // do work here
}
```

- **The main queue (a serial queue)**

```
let mainQ: dispatch_queue_t =dispatch_get_main_queue()
let mainQ: NSOperationQueue = NSOperationQueue.mainQueue() // for OO APIs
```

  - All UI stuff must be done on this queue!
  - And all time-consuming (or, worse, potentially blocking) stuff must be done off this queue!
  - Common code to write …

```
dispatch_async(not_main_Q) {
    // do something takes a while
    dispatch_async(dispatch_get_main_queue()) {
        // call UI functions with results
    }
}
```

# Multithreading

- **Other (concurrent) queues (i.e. not the main queue)**
  - Most non-main-queue work will happen on a concurrent queue with a certain quality of service

```
QOS_CLASS_USER_INTERACTIVE  // quick and high priority
QOS_CLASS_USER_INITIATED    // high priority, might take time
QOS_CLASS_UTILITY           // long running
QOS_CLASS_BACKGROUND        // prefetching, etc...

let qos = Int(<one of the above>.rawValue) // historical reasons
let queue = dispatch_get_global_queue(qos, 0)
```

  - You will probably use these queues to do any work that you don't want to block the main queue
- **You can create your own serial queue if you need serialization**

```
let serialQ = dispatch_queue_create("name", DISPATCH_QUEUE_SERIAL)
```

  - Maybe you are downloading a bunch of things things from a certain website but you don't want to deluge that website, so you queue the requests up serially
  - Or maybe the things you are doing depend on each other in a serial fashion

# Multithreading

- **Scheduling work in the future**

```
let delayInSeconds = 25.0
let delay = Int64(delayInSeconds * Double(NSEC_PER_MSEC)) // historical
let dispatchTime = dispatch_time(DISPATCH_TIME_NOW, delay) // adds delay to NOW
dispatch_after(dispatchTime, dispatch_get_main_queue()) {
    // do something on the main queue 25 seconds from now
}
```

- **We are only seeing the tip of the iceberg**
  - There is a lot more to GCD
  - You can do locking, protect critical sections, readers and writers, synchronous dispatch, etc.
  - Check out the documentation if you are interested

# Multithreading

- **Multithreaded iOS API**
  - Quite a few places in iOS will do what they do off the main queue
  - They might even afford you the opportunity to do something off the main queue
  - You may pass in a function (a closure, usually) that sometimes executes off the main thread
- **Remember that if you want to do UI work there, you must dispatch back to the main queue!**

# Multithreading

- **Example of a multithreaded iOS API**
  - This API lets you fetch something from an http URL to a local file
  - Obviously it can't do that on the main thread!

```
let session = NSURLSession(NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: "http://url") {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        // Can we update UI with the result of the download?
    }
    task.resume()
}
```

  - The answer to the above comment is "no".
  - That's because the block will be run off the main queue.
  - How do we deal with this?
  - One way is to use a variant of this API that lets you specify the queue to run on.
  - Another way is …

# Multithreading

- **How to do UI stuff safely**
  - You can simply dispatch back to the main queue …

```
let session = NSURLSession(NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: "http://url") {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        dispatch_async(dispatch_get_main_queue()) {
           // Update UI with the result of the download
        }
    }
    task.resume()
}
```

  - OK because the UI code you want to do has been dispatched back to the main queue
  - But remember that that code might run MINUTES after the request is fired off
  - The user might have long ago given up on whatever was being fetched