Stanford CS193p

Developing Applications for iPhone 4, iPod Touch, & iPad

Fall 2010

Today

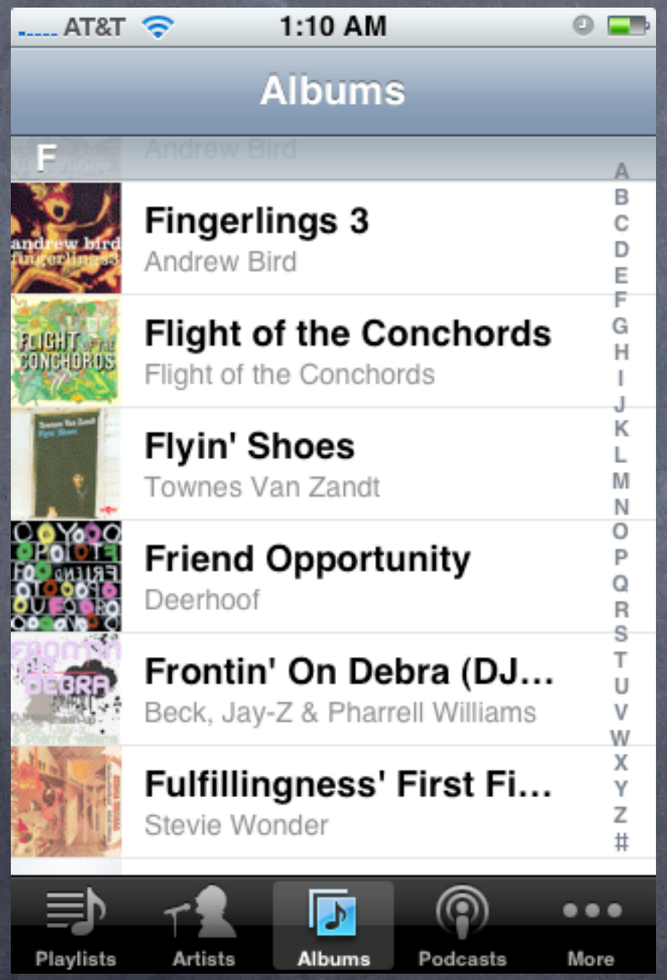
- `UITableView`
- `UITableViewDataSource`
- `UITableViewCell`
- `UITableViewDelegate`
- `UITableViewController`
- Demo
  Vocabulous

UITableView

- Very important class for displaying data in lists
  - It's a subclass of `UIScrollView`.
  - Lots and lots of customization via a `dataSource` protocol and a `delegate` protocol.
  - Very efficient even with very large sets of data.
- Can only display one column of data at a time
  - Often table views are pushed from each other to display a hierarchical data set.
  - The column can be divided into sections for user-interface cleanliness or easy access to large lists.
- Two styles
  - `UITableViewStylePlain`
  - `UITableViewStyleGrouped`
  - Designated initializer takes the style you want …
  - `- (id)initWithFrame:(CGRect)aRect style:(UITableViewStyle)style;`
  - You cannot change the style.

UITableView

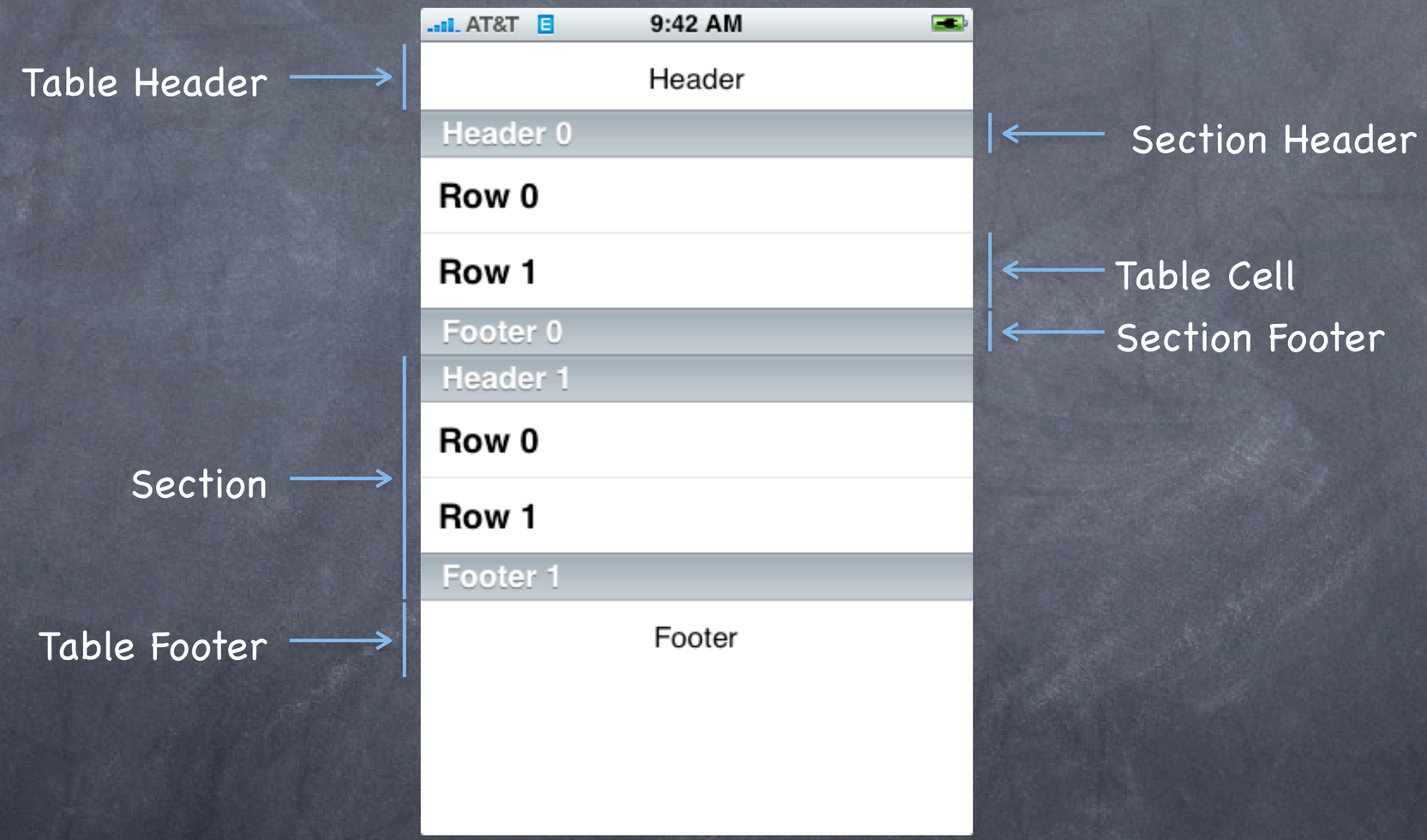
UITableViewStylePlain



UITableViewStyleGrouped

UITableView

Plain Style



UITableView

Grouped Style

Table Header

Section

Table Footer

Section Header

Table Cell

Section Footer

UITableView

- How do you get data into the table?
  Delegated. Your MVC's Controller is almost always the `dataSource` for the `UITableView`.
  `@property (assign) id <UITableViewDataSource> dataSource;`
- We don't want to load all the data at once
  That would be very inefficient.
  So, just like your `GraphView`, the `UITableView` only asks for data as it needs it.
  This means that if a row in the column is not on screen, the data is not asked for for that row.
- But the table view needs to know the "size" of the data up front
  So that it can set the `contentSize` of itself (it's a `UIScrollView`, remember).
- So the first thing it will ask its `dataSource` is "how many rows?"
  Actually, it will ask how many sections, then ask how many rows in each section.
- Then it will start asking the `dataSource` to provide the data
  But only as each row comes on screen.

UITableViewDataSource

- How many sections in the table?

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)sender;
```

The default (if you don't implement this method) is 1.

- How many rows in a given section?

```
- (NSInteger)tableView:(UITableView *)sender numberOfRowsInSection:(NSInteger)section;
```

This method is REQUIRED. It cannot default to anything sensible.

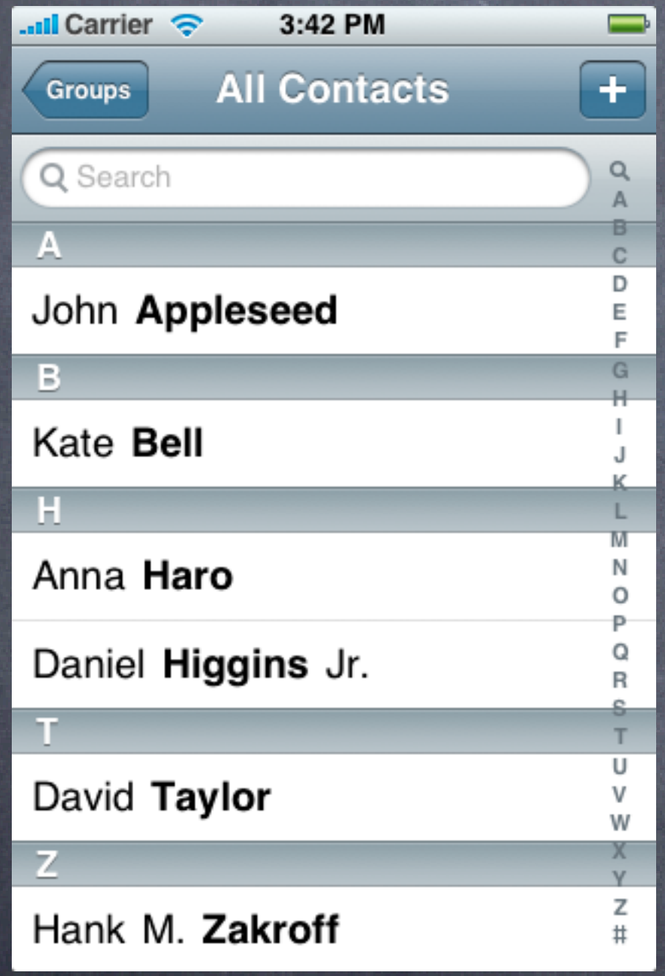
- Provide data for a given row in a given section

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

An `NSIndexPath` is a way of specifying a section and row in one argument.

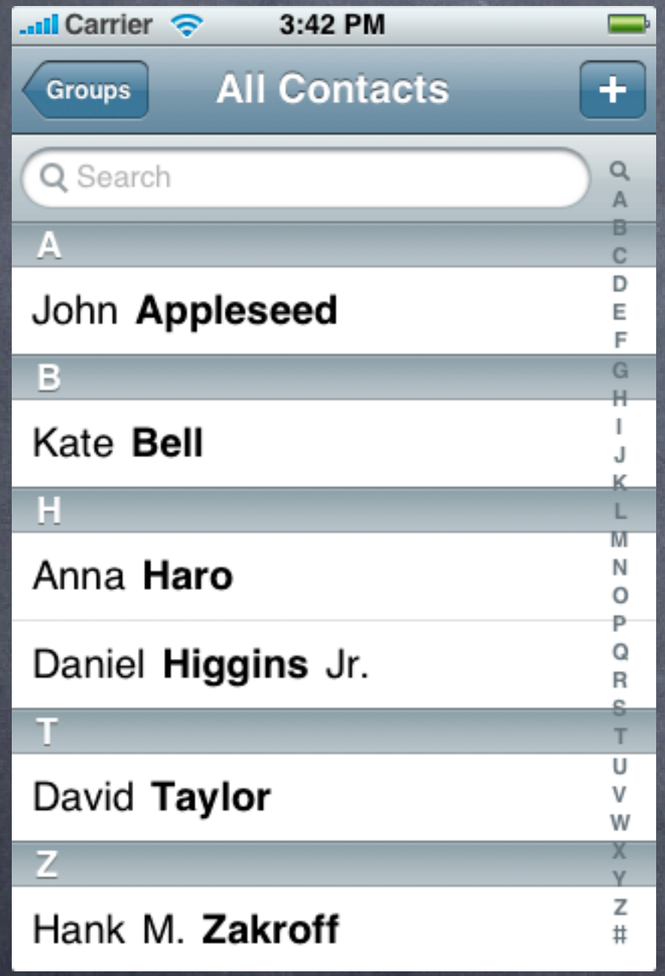
`indexPath.section` is the section, `indexPath.row` is the row.

UITableView



Datasource

UITableView

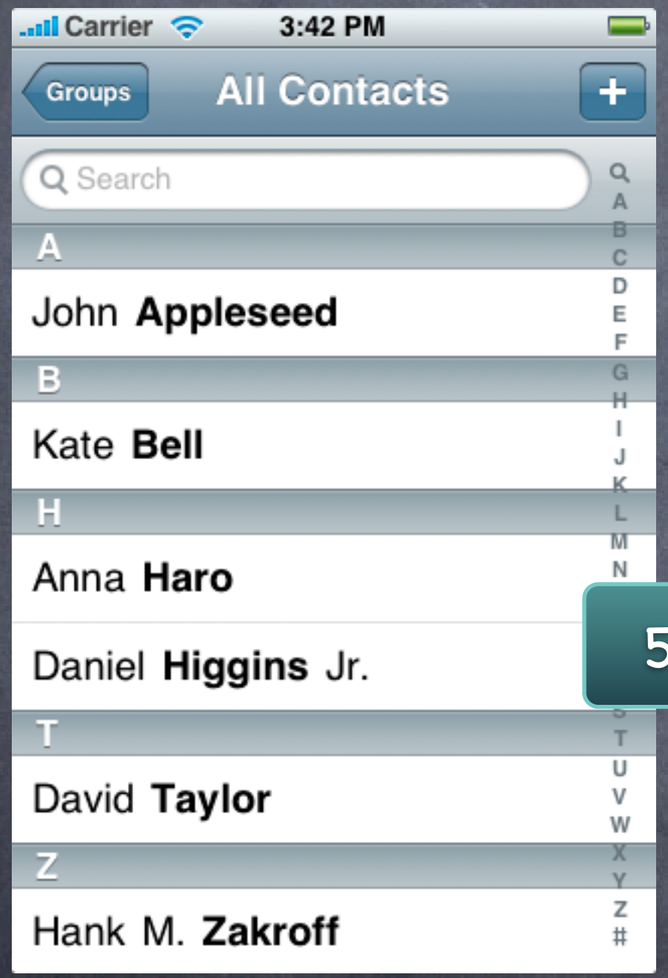


numberOfSectionsInTableView:

How many sections?

Datasource

UITableView

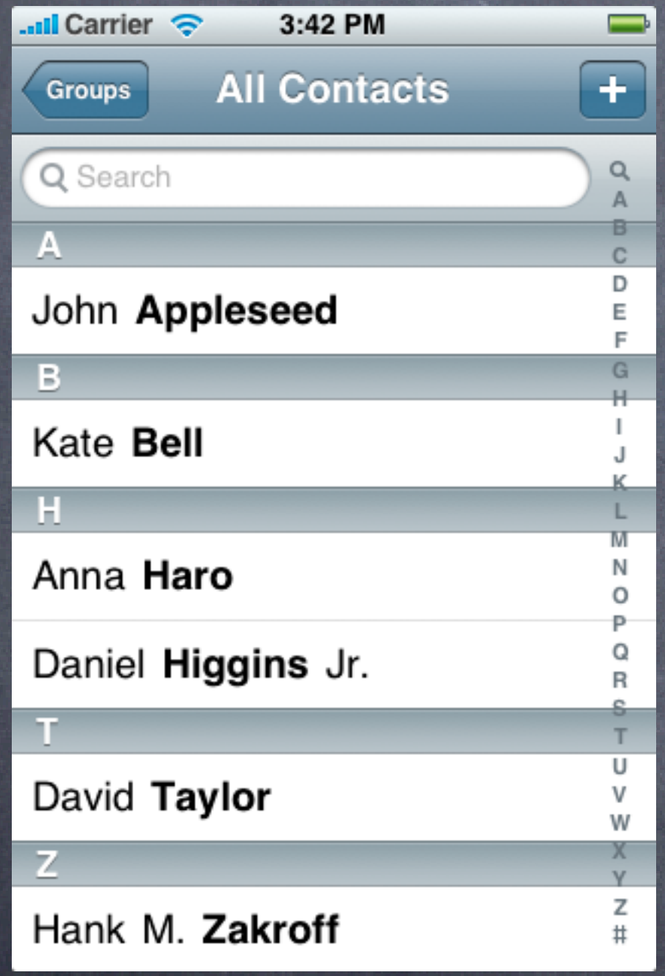


numberOfSectionsInTableView:

5

Datasource

UITableView

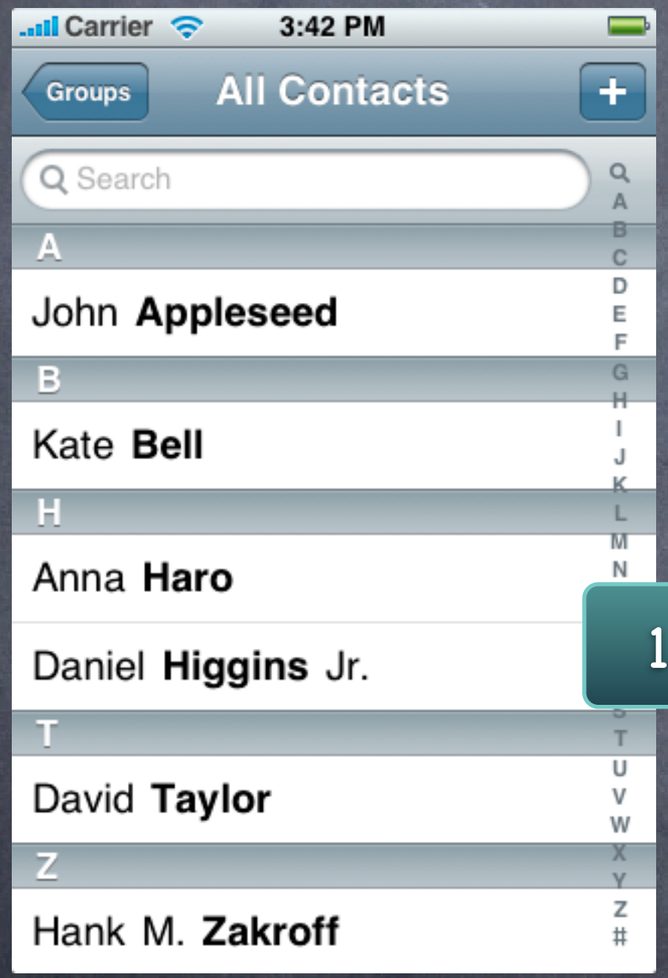


`tableView:numberOfRowsInSection:`

How many rows in section 0?

Datasource

UITableView

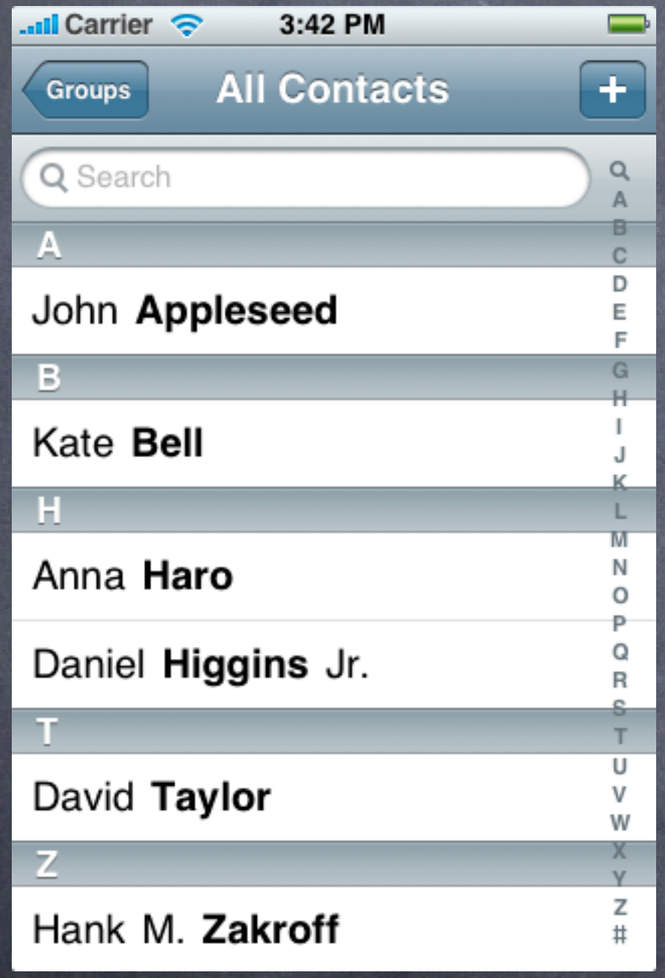


tableView:numberOfRowsInSection:

Datasource

1

UITableView

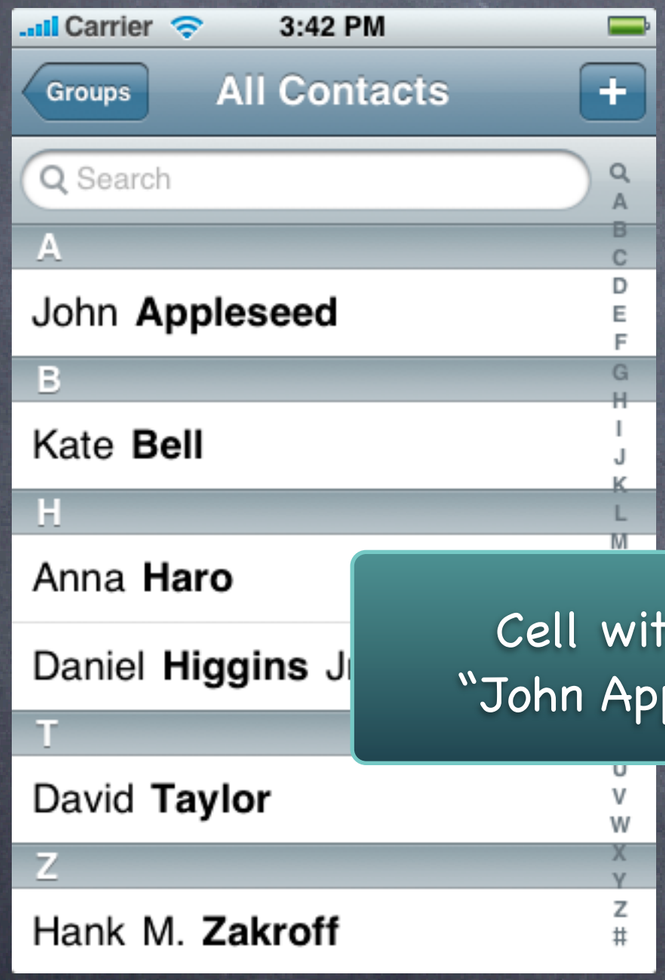


`tableView:cellForRowAtIndexPath:`

What to display at section 0, row 0?

Datasource

UITableView



tableView:cellForRowAtIndexPath:

Cell with text "John Appleseed"

Datasource

UITableViewDataSource

- Example: Data is stored in an NSArray of NSString
- Just one long list of strings in this example

  So no need to implement numberOfSectionsInTableView:.
- Number of rows is just count of objects in my array

```
- (NSInteger)tableView:(UITableView *)sender numberOfRowsInSection:(NSInteger)section {
    return myArray.count;   // no need to check section here since there's only one
}
```

- How to respond when asked to supply data for a row

```
- (UITableViewCell *)tableView:(UITableView *)sender
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = ...;  // more to come on this
    cell.textLabel.text = [myArray objectAtIndex:indexPath.row];  // section always 1
    return cell;
}
```

UITableViewCell

- So what's this `UITableViewCell` thing?
  It's a `UIView` subclass for drawing a row in the table view.

- You set it up to display the data in a given row
  Important methods:

```
@property (readonly) UILabel *textLabel;
@property (readonly) UILabel *detailTextLabel;
@property (readonly) UIImageView *imageView;
```

  Note that these are all `readonly`. They are lazily created when you call them.
  The `detailTextLabel` is like a little subtitle or other secondary text (depending on style).
  Obviously you wouldn't want to set the `imageView`'s `image` property to a huge image.

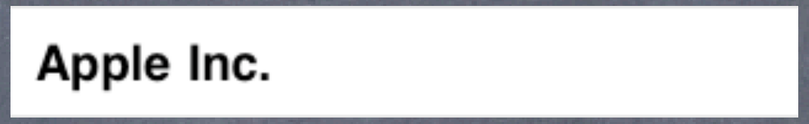
- Designated initializer takes the style of the cell (no frame)

```
- initWithStyle:(UITableViewCellStyle)style reuseIdentifier:(NSString *)reuseId;
```

  We'll talk about the reuse identifier in a moment.
  First, let's talk about style and the above properties.

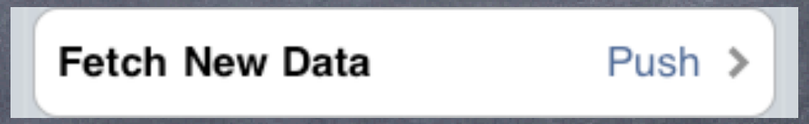
UITableViewCell

UITableViewCellStyleDefault

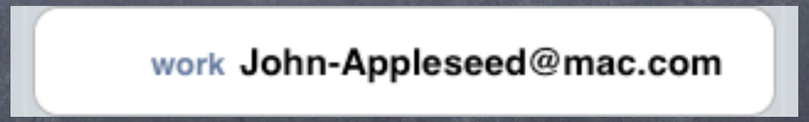


UITableViewCellStyleSubtitle

UITableViewCellStyleValue1



UITableViewCellStyleValue2

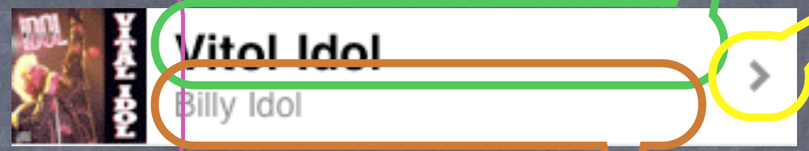


UITableViewCell

@property (readonly) UILabel *textLabel;

@property (readonly) UIImageView *imageView;

@property
UITableViewAccessoryType
accessoryType;

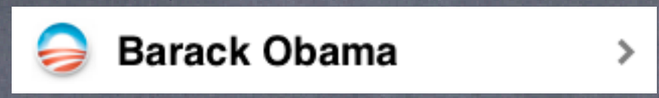


@property (readonly) UILabel *detailTextLabel;

```
- (UITableViewCell *)tableView:(UITableView *)sender
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = ...; // reuse or create
    cell.textLabel.text = [myArray objectAtIndex:indexPath.row];
    cell.imageView.image = [myImages objectAtIndex:indexPath.row];
    cell.detailTextLabel.text = [mySubtitles objectAtIndex:indexPath.row];
    cell.accessoryType = <figure out what kind of accessory type we want>;
    return cell;
}
```

UITableViewCell

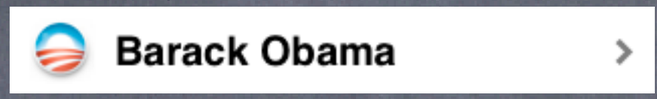
UITableViewCellAccessoryDisclosureIndicator



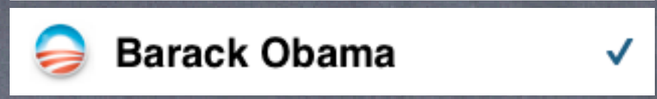

```
- (UITableViewAccessoryType)tableView:(UITableView *)sender
     accessoryTypeForRowWithIndexPath:(NSIndexPath *)indexPath
{
    if ([[myArray objectAtIndex:indexPath.row] someMethod]) {
        return UITableViewAccessoryTypeDisclosureIndicator;
    }
}
```

UITableViewCell

UITableViewCellAccessoryDisclosureIndicator



UITableViewCellAccessoryCheckmark



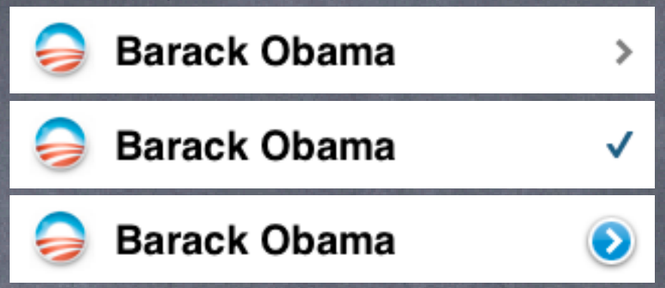

```
- (UITableViewAccessoryType)tableView:(UITableView *)sender
     accessoryTypeForRowWithIndexPath:(NSIndexPath *)indexPath
{
    if ([[myArray objectAtIndex:indexPath.row] someMethod]) {
        return UITableViewAccessoryCheckmark;
    }
}
```

UITableViewCell

UITableViewCellAccessoryDisclosureIndicator

UITableViewCellAccessoryCheckmark

UITableViewCellAccessoryDetailDisclosureButton




```
- (UITableViewAccessoryType)tableView:(UITableView *)sender
     accessoryTypeForRowWithIndexPath:(NSIndexPath *)indexPath
{
    if ([[myArray objectAtIndex:indexPath.row] someMethod]) {
        return UITableViewCellAccessoryDetailDisclosureButton;
    }
}

- (void)tableView:(UITableView *)sender
          accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
    // only get here if a blue button is tapped
}
```

UITableViewCell

- So what's this "reuse identifier" business?

  It's all about performance.

  We don't want to create 10,000 UITableViewCells for a table with 10,000 rows.

  We only want to create about 7 or 8 (which is how many are on screen at the same time).

  So we "reuse" them.

- Back to our NSArray of NSStrings example

```
- (UITableViewCell *)tableView:(UITableView *)sender
        cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [sender dequeueReusableCellWithIdentifier:@"MyTV"];
    if (!cell) cell = [[[UITVC alloc] initWithStyle:... reuseIdentifier:@"MyTV"] autorelease];
    cell.textLabel.text = [myArray objectAtIndex:indexPath.row];
    return cell;
}
```

* UITVC is shorthand for UITableViewCell

UITableViewCell

- What else can you do to display a row besides text/image?
  Turns out … a lot!
- The accessory area can be a view (instead of the indicators)
  `@property (retain) UIView *accessoryView;`
  You set this property to the view you want to appear in the accessory area.
- You can display a custom view in a table view cell
  `@property (readonly) UIView *contentView;`
  Get the value of this property and add your custom view as a subview.
  Match your custom view's `frame` to the `contentView`'s & make sure your "stretchiness" is set right.
- You can subclass `UITableViewCell` and/or build its UI in IB too
  Doing so is beyond the scope of this course.
  But there is a lot of good Apple documentation on the subject.

More UITableView

- What else can the dataSource control?

  Content of the headers and footers.
  Editability of the information in the table (inserting, rearranging, deleting rows).

- Controlling the display of headers and footers

```
- (NSString *)tableView:(UITableView *)sender
titleForHeaderInSection:(NSInteger)section;
```

  This is the `dataSource` providing data for what's in the header (similar footer method of course).

Editing UITableView

- Asking the dataSource to actually edit the underlying data

```
- (void)tableView:(UITableView *)sender
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
     forRowAtIndexPath:(NSIndexPath *)indexPath;
```

```
UITableViewCellEditingStyleDeletion    // delete the data at the specified indexPath
UITableViewCellEditingStyleInsertion   // insert a new row at the specified indexPath
```

You don't have to do the insert/delete, but if you do, you MUST update the UITableView (for example, using the method deleteRowsAtIndexPaths:withRowAnimation:).

- Controlling whether a row can be deleted or added

```
- (BOOL)tableView:(UITableView *)sender canEditRowAtIndexPath:(NSIndexPath *)indexPath;
```

Default is YES. Thus, if you don't implement, cells will be editable according to their editingStyle property whose value is retrieved from the UITableView's delegate (default is Deletion).

- User activates delete/insert button on a row in one of two ways

By swiping, or by clicking a button which sets the UITableView @property BOOL editing.
Pre-canned button: self.navigationController.rightBarButtonItem = self.editButtonItem.

Reordering UITableView

- Asking the dataSource to move rows in the underlying data

```
- (void)tableView:(UITableView *)sender
    moveRowAtIndexPath:(NSIndexPath *)sourcePath
           toIndexPath:(NSIndexPath *)destinationPath;
```

  If you do not implement this, then cells cannot be reordered.
  If you implement it, you MUST move the row when asked (different than similar editing method).

- Movability controlled by dataSource

```
- (BOOL)tableView:(UITableView *)sender canMoveRowAtIndexPath:(NSIndexPath *)indexPath;
```

  Default is NO, so if you do not implement this, then cells cannot be reordered.

- Must show reordering controls in cells for this to work

```
@property BOOL showsReorderControl;  // this is a settable property in UITableViewCell
```

  The table view must be in editing mode for the reordering control to appear.
  Mode often activated via `self.navigationController.rightBarButtonItem = self.editButtonItem`.
  Can you be reordering and editing (deleting or inserting) at the same time?
  Yes, because swiping across a cell changes the reorder control to delete/insert control.

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

UITableViewDelegate

- Notification that a row is about to appear on screen

```
- (void)tableView:(UITableView *)sender
  willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath;
```

This is a last-second chance to muck with the cell before it is displayed.

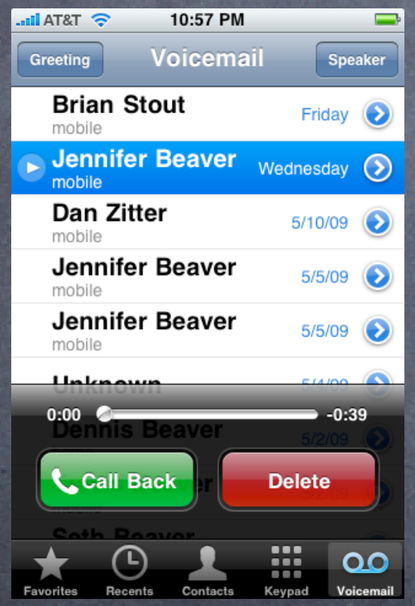
- Row selection

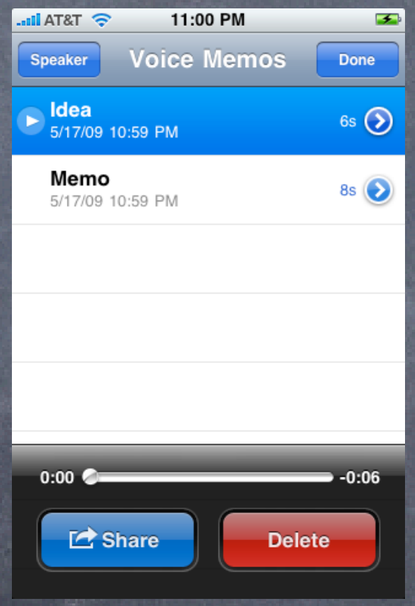
Here's where you put your code to handle selection of a row …

```
- (void)tableView:(UITableView *)sender
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    SomeViewController *vcToPush = [[SomeViewController alloc] init];
    vcToPush.someProperty = <something dependent on my data at indexPath>;
    [self.navigationController pushViewController:vcToPush animated:YES];
    [vcToPush release];
}
```

UITableViewDelegate

- Usually selecting a cell pushes another view controller
  - So it is rare for a selection to "stick" in a `UITableView`.
  - But it is not unheard of, especially with disclosure detail buttons.





UITableViewDelegate

- Controlling the height of cells

```
- (CGFloat)tableView:(UITableView *)sender
  heightForRowAtIndexPath:(NSIndexPath *)indexPath;
```

- Last chance to modify cell before it is displayed

Some properties (like the background color of the `UITableViewCell`) must be set here.

```
- (void)tableView:(UITableView *)sender
  willDisplayCell:(UITableViewCell *)cell
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    cell.backgroundColor = [UIColor greenColor];
}
```

UITableViewDelegate

- Controlling the height of headers and footers

```
- (CGFloat)tableView:(UITableView *)sender
    heightForHeaderInSection:(NSInteger)section;
```

- You can even provide your own custom header/footer views

```
- (UIView *)tableView:(UITableView *)sender
    viewForHeaderInSection:(NSInteger)section;
```

- Editing style for cells (as previously discussed)

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)sender
    editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath;
```

- Preventing selection of cells

```
- (NSIndexPath *)tableView:(UITableView *)sender
    willSelectRowAtIndexPath:(NSIndexPath *)indexPath  {
    if (<we don't want row at indexPath to be selected>) return nil;
}
```

UITableViewController

- Convenience class

  It's a `UIViewController` which implements `dataSource` and `delegate` protocols of `UITableView`

- Implements `loadView` for you

  Sets the `view` property to be a `UITableView` that it creates for you.
  You can still have a `.xib` file if you want though (same as any other `UIViewController`).
  Just create a `.xib` file with the same name as your `UITableViewController` subclass.
  If you go `.xib`, make sure you connect an outlet from the `tableView` ivar to the `UITableView`.

- Does some other stuff for you as well ...

  Properly selects and deselects rows at right time if pushing in navigation controller.
  Reloads the data just before the view appears.
  Handles the Edit button (`self.editButtonItem`) to make rows editable.
  Flashes scroll indicators when the view comes on screen.
  Makes sure there's room for a keyboard if there is an editable view inside a cell in the table.

Coming Up

- Demo
  - "Vocabulous"
  - `UITableView` which displays an `NSDictionary` of `NSArrays` of `NSStrings`
  - Pushes an MVC which uses a `UIWebView`
- Homework
  - No more Calculator!
  - "Places"
  - Lets the user browse popular photo spots from Flickr and look at photos from those spots.
  - `UITabBarController`
  - `UITableViewController`
  - `UIImageView`
  - `UIScrollView`
- Next Week
  - Persistence and Multithreading