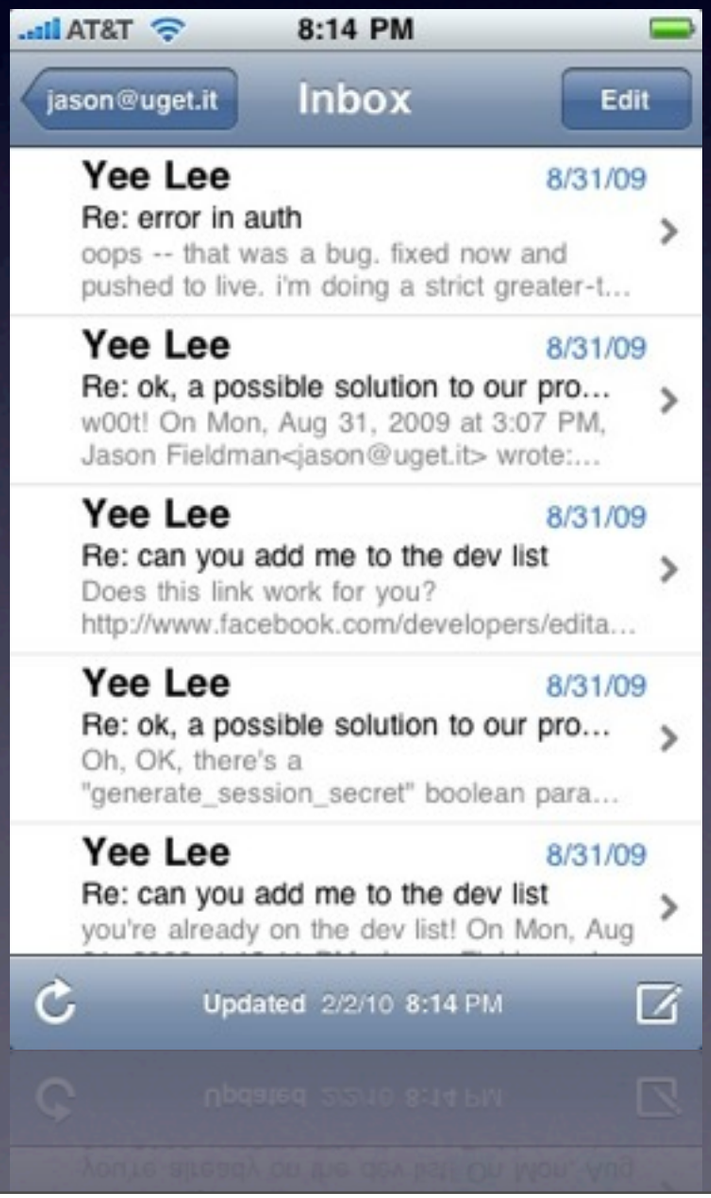
CS 47

Beginning iPhone Application Development

Week 4: Tables

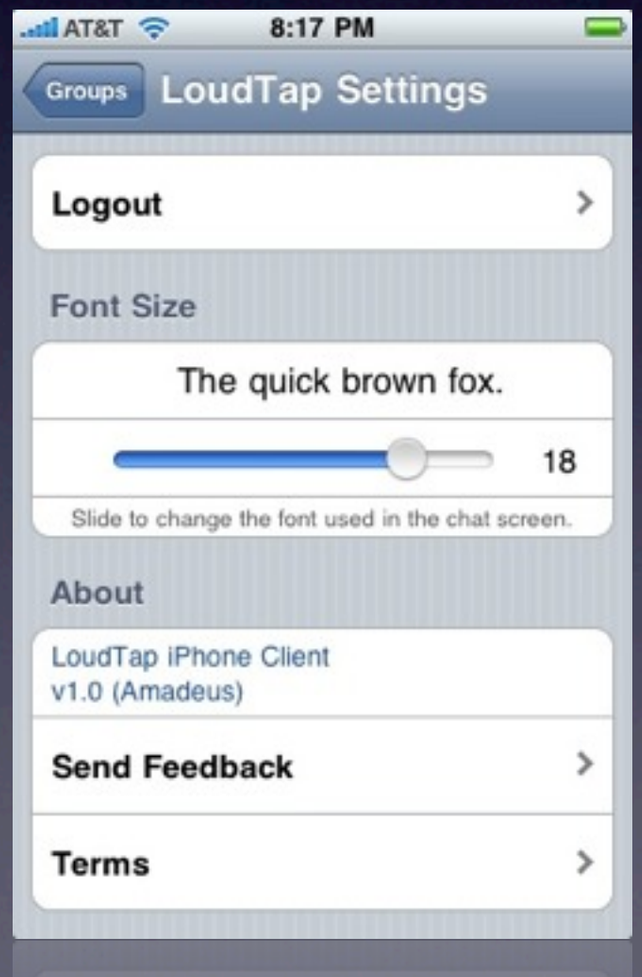
Why?

- You want to render a list of data as a table



Why?

- Appearance convention for certain types of screens (login, settings, data entry; modular)



Basic Vocabulary

Row

Section

Header

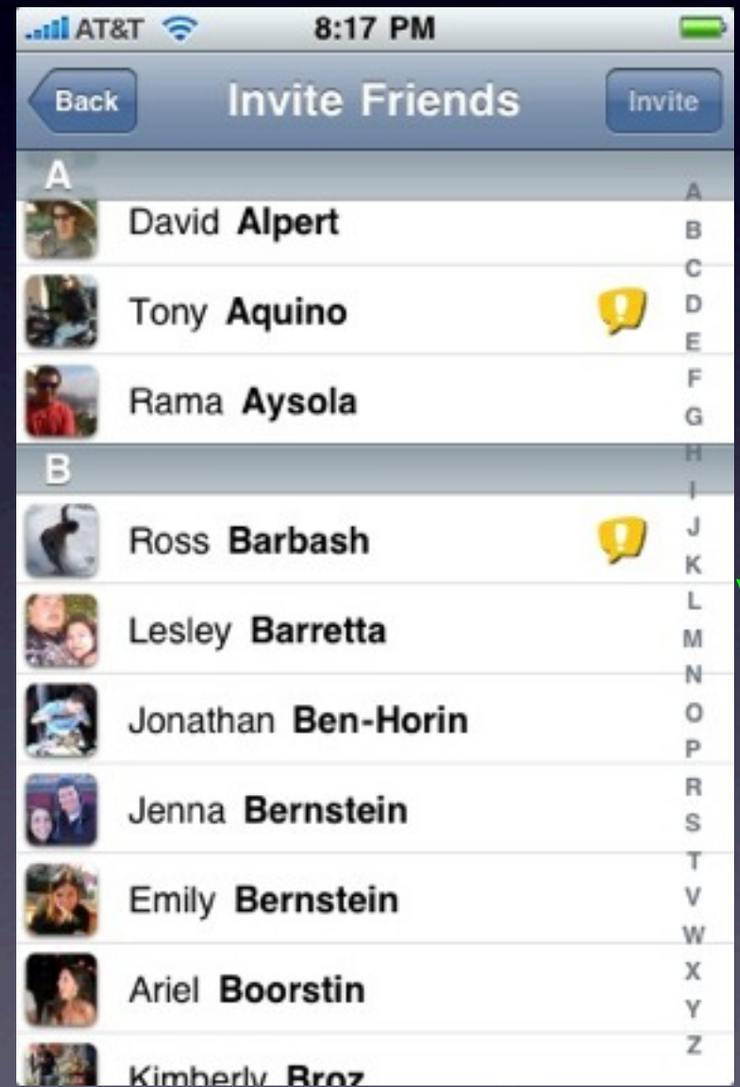
UITableViewStyleGrouped

Header

Row

Section

Index

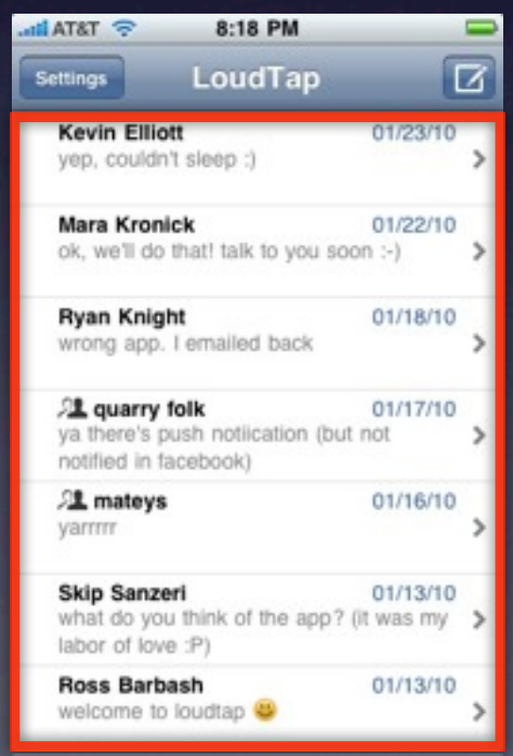


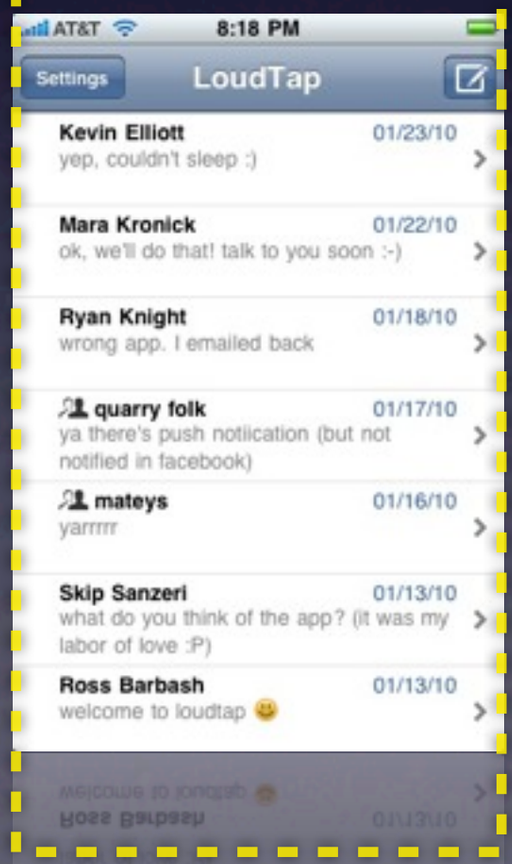
UITableViewStylePlain

UIScrollView

- Different from all other UIView subclasses
  - Manages a scrollable contentView

frame





contentView

contentSize

contentOffset

Problem

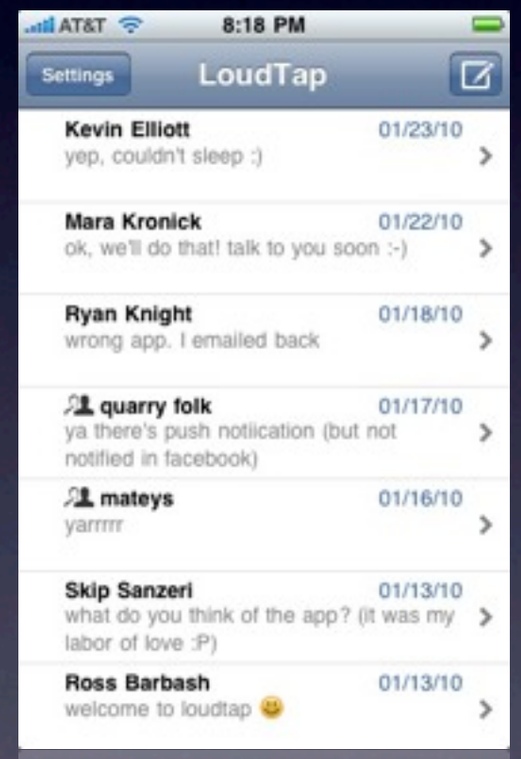
- What if we have a list with thousands of entries?
- Could we render that entire list in the contentView of a UIScrollView?
  - Render time
  - Memory limitations

Solution

- We need a way to scroll through a list of thousands of objects, with quick rendering time and low memory requirements
- UITableView : UIScrollView : UIView

How Does it Work?

- Look at this table: Even if we have a thousand rows, we're only displaying seven at a time
- The table does not care about data that is off-screen
- When the table scrolls, old rows are dropped and new rows are added



MVC

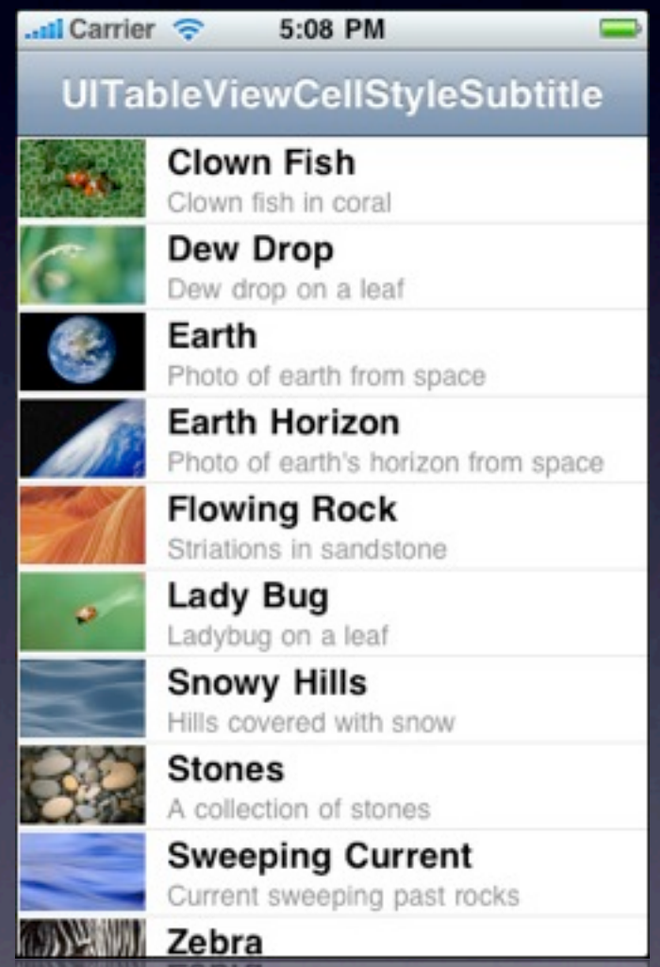
- The entire data set is not managed by the table (only the active cells)
- And it shouldn't be - the table is a view and only responsible for displaying information
- You need a separate model that contains the actual table data (e.g. a database on disk)

Cells

- So how does the UITableView manage rows?
- UITableViewCell : UIView
- Each visible row of the table is represented by a unique instance of UITableViewCell

Cells

- UITableViewCell has some helper attributes to render basic types of data



- initWithStyle:reuseIdentifier: sets the "style" of the cell; various styles available
- Can access the basic image, text and description fields

Cells

- But normally you will want to customize your own UITableViewCell objects
- Subclass UITableViewCell to add your own subviews and accessors

```
GroupListCellView *groupCell = ...;
Group             *myGroup   = ...;

groupCell.group = myGroup;
```

Cells

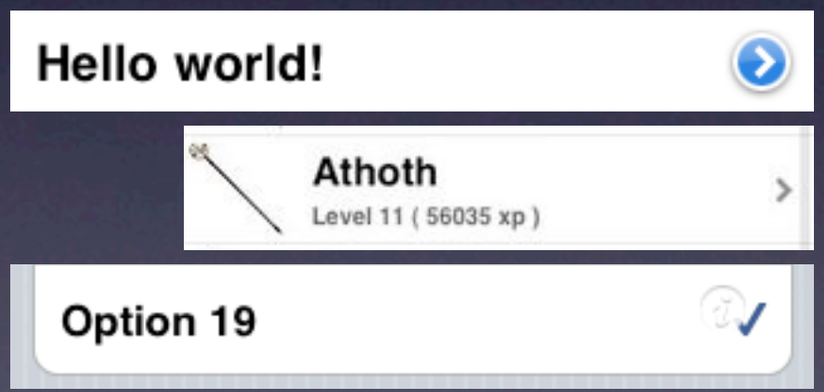
- Even though UITableViewCell is a UIView, you should not add views to it directly (e.g. do not call `addSubview:` directly on the cell

- Use the contentView property of the UITableViewCell

```
UITableViewCell *cell   = ...;
UIButton        *button = ...;
[cell.contentView addSubview:button];
```

Accessories

- Cells can also manage accessories
- Use the accessoryType accessor to set standard accessories




```
cell.accessoryType = ...
```

UITableViewCellAccessoryDetailDisclosureButton

UITableViewCellAccessoryDisclosureIndicator

UITableViewCellAccessoryCheckmark

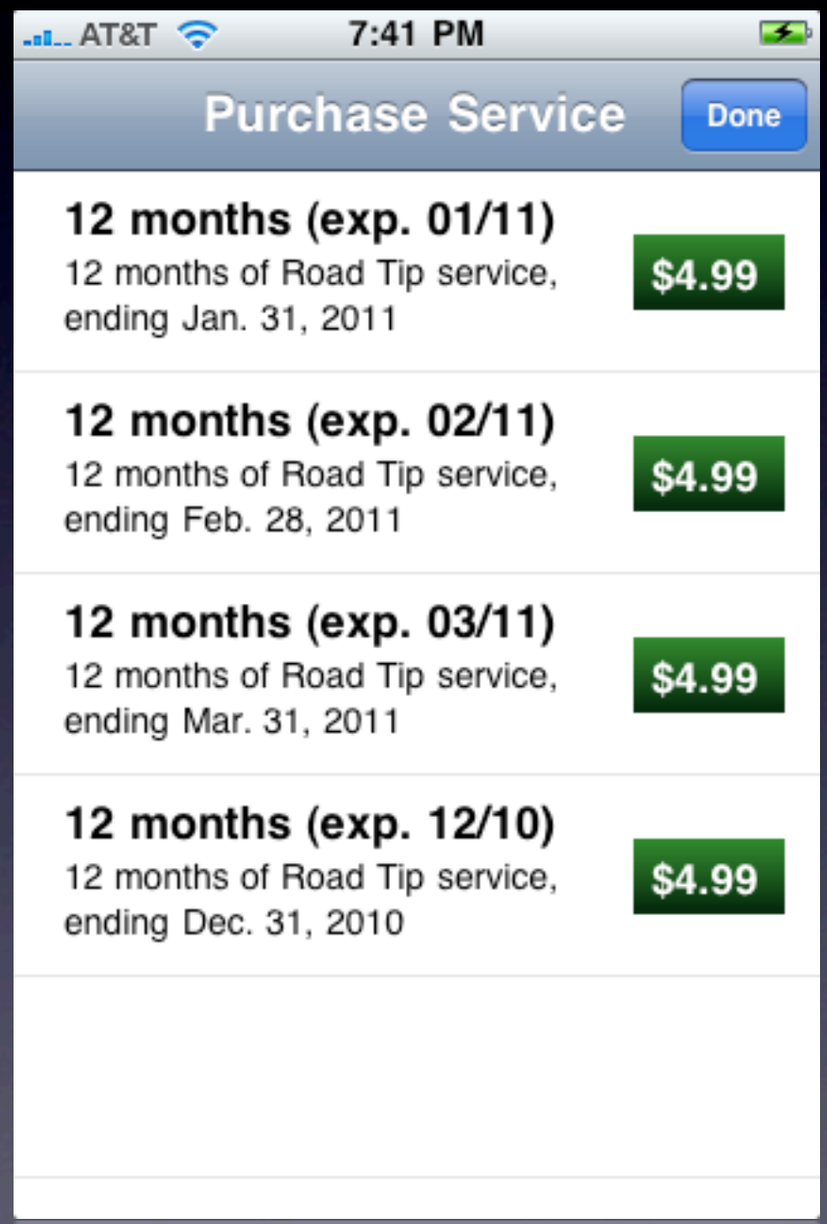
Accessories

- You can provide any custom view as a managed accessory by setting the accessoryView property

```
UISwitch *switch = [[[UISwitch alloc] init] autorelease];
[switch addTarget:target action:@selector(toggled:)
                forControlEvents:UIControlEventValueChanged];
cell.accessoryView = switch;
```

Cell Interaction

- You can put any number of interactive elements inside a cell
- Let's say those price tags are buttons.. what happens when you touch one?



Reusing Cells

- So we have a small amount of cells visible at a time. As we scroll, cells leave one side and enter from the other
- It would be really inefficient to dealloc and reinitialize a new cell every time this happened

Reusing Cells

- Instead, there is the concept of the "reuseIdentifier"

- UITableViewCells that perform the same function should be initialized with the same reuseIdentifier

Reusing Cells

- The UITableView keeps a pool of UITableViewCell objects that have left the screen

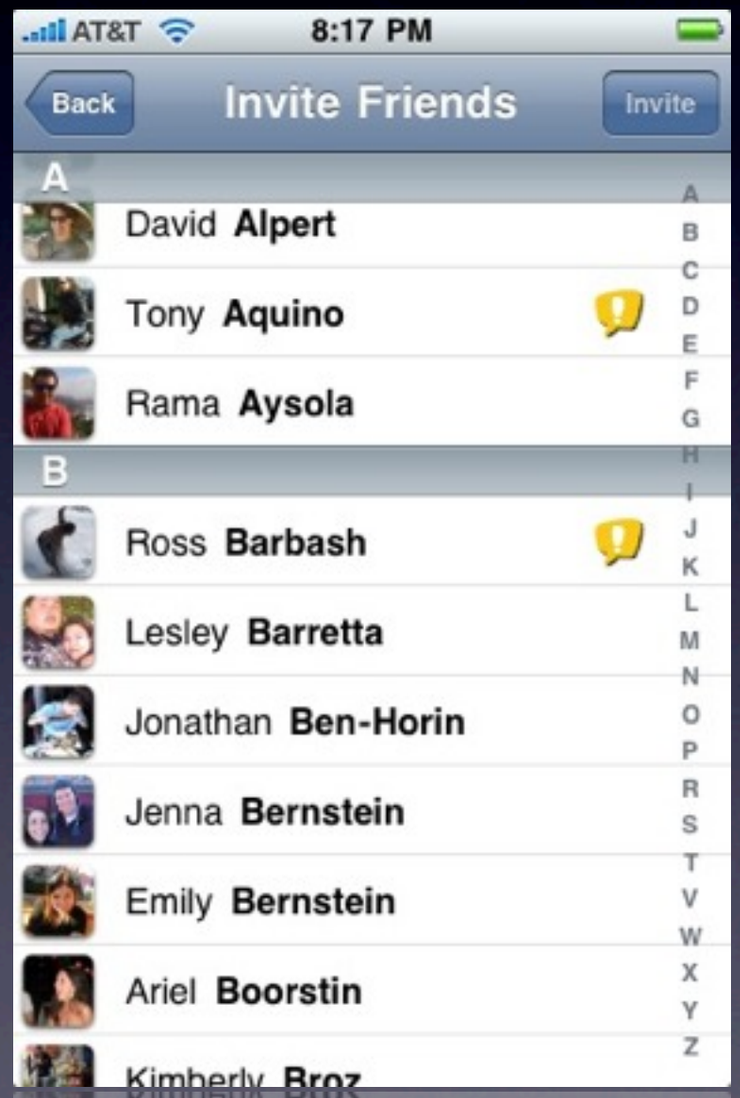
- You can query this pool with the reuseIdentifier key

```
UITableView *tableView = ...;
NSString    *ident     = @”Some String”;
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:ident];

/* cell now points to an old cell, that had the same reuseIdentifier..
   or nil if none existed */
```

Reusing Cells

- So if we have a thousand rows that all use the same cell style to display, the table only needs to keep a few instances of that type of UITableViewCell object
- Constantly recycling them with new data



(Only ~10 cell objects are instantiated)

Reusing Cells

- If you create a UITableViewCell without a reuseIdentifier (nil), then it will be released by the table as soon as it leaves the screen
- Why would we want to do this?

Static vs. Dynamic

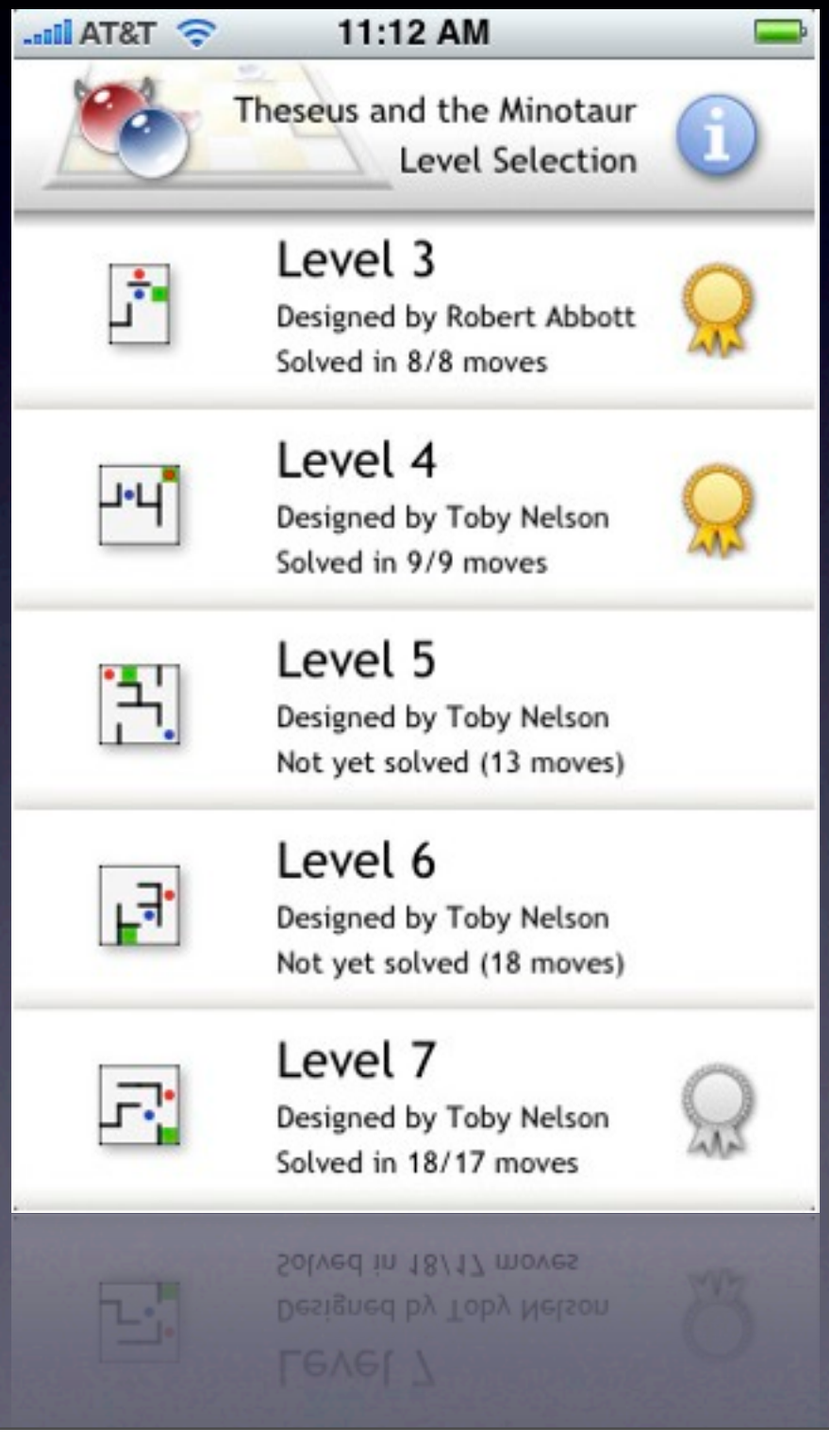
- If we intend to pre-allocate all of the table cells
- What? Pre-allocating all of the UITableViewCells? Isn't that what we're trying to avoid?
- Under what conditions would we want to pre-allocate all of our table cells?

Static vs. Dynamic

- Reason to pre-create UITableViewCells:
- Table is small with lots of unique row types (login, settings, etc)

Static vs. Dynamic

- If cells take a relatively long time to render new info, and you have a small/fixed set of rows



Static vs. Dynamic

- What if you have thousands of rows that all take a long time to render when the data changes?
- You shouldn't pre-allocate thousands of cells, but you'll need some kind of pre-render caching mechanism

Static vs. Dynamic

- Remember, the UITableView only retains the non-reusable UITableViewCells that are on the screen

- When you pre-allocate the entire set of UITableViewCells, you need to keep them in a data structure that will retain them (like an NSArray or NSDictionary)

Populating the Table

- Ok, so how do we actually link the table with its cells? (Think MVC)
- Protocol-delegate interface

- Everything handled by the UITableViewDelegate and UITableViewDataSource
- Generally, this will be implemented by your view controller

Assign the Delegate

```
@interface MyViewController : UIViewController <UITableViewDelegate,
                                                UITableViewDataSource> {
  UITableView myTableView;
}
@end

@implementation MyViewController
- (id) init {
  ...
  myTableView = [[[UITableView alloc] initWithFrame:CGRectMake(0, 0, 320, 400)
                                              style:UITableViewStylePlain]
                                              autorelease];
  myTableView.delegate   = self; /* Attaching myself as the delegate and */
  myTableView.dataSource = self; /* data source for the table            */
  [self.view addSubview:myTableView]; /* Adding it to my view */
  ...
}
...
@end
```

Reloading Data

- A UITableView reloads data from its delegate and data source under two conditions:
  - When it is first created
  - Passing the reloadData message

    ```
    [myTable reloadData];
    ```

    - reloadData is not immediate!

Sections and Rows

- We need to tell the table how large it will be (how many sections and rows)
- The delegate implements these methods:

```
- (NSInteger) numberOfSectionsInTableView:(UITableView *)tableView {
  return numSections;
}


- (NSInteger) tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {

  return [sections objectAtIndex:section].numRows;
}
```

Row Height

- You can make all rows have the same height

```
myTable.rowHeight = 50;
```

- Or you can give rows variable height through the delegate

```
- (CGFloat) tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath {
  return (( indexPath.row + 1 ) * ( 10 + indexPath.section ));
}
```

- Using rowHeight is waaay faster

Linking In Cell Views

- How do we identify which cells go where?
- UITableViewDataSource delegate implements

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

  return /* Returns UITableViewCell */;

}
```

Linking in Cell Views

- Pre-allocated method

```
- (id) init { ...
  myCellArray = [[NSMutableArray alloc] init];
  for (...) {
    tableCell = [[[UITableViewCell alloc] initWith...] autorelease];
    tableCell.textLabel.text = [myNameArray objectAtIndex:..];
    [myCellArray addObject:tableCell];
  }
... }

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

  return [myCellArray objectAtIndex:indexPath.row];

}
```

Linking in Cell Views

- Dynamic method (reusable cells)

```
#define kReuseIdent @”MyCellType”

- (UITableViewCell*) tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  UITableViewCell *cell =
          [tableView dequeueReusableCellWithIdentifier:kReuseIdent];

  if (!cell) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                    reuseIdentifier:kReuseIdent] autorelease];
  }

  cell.textLabel.text = [myNameArray objectAtIndex:indexPath.row];
  return cell;
}
```

Detecting Touches

- How do we know if a user selected our row?

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

  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  /* Do something with the value of this cell? */
  MyViewController *myVC = [[[MyViewController alloc] init] autorelease];
  myVC.someData = [someArray objectAtIndex:indexPath.row];
  [self.navigationController pushViewController:myVC animated:YES];

  /* Let's also deselect the row */
  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Detecting Accessories

- The `UITableViewCellAccessoryDetailDisclosureButton` accessory has a special delegate callback if pressed

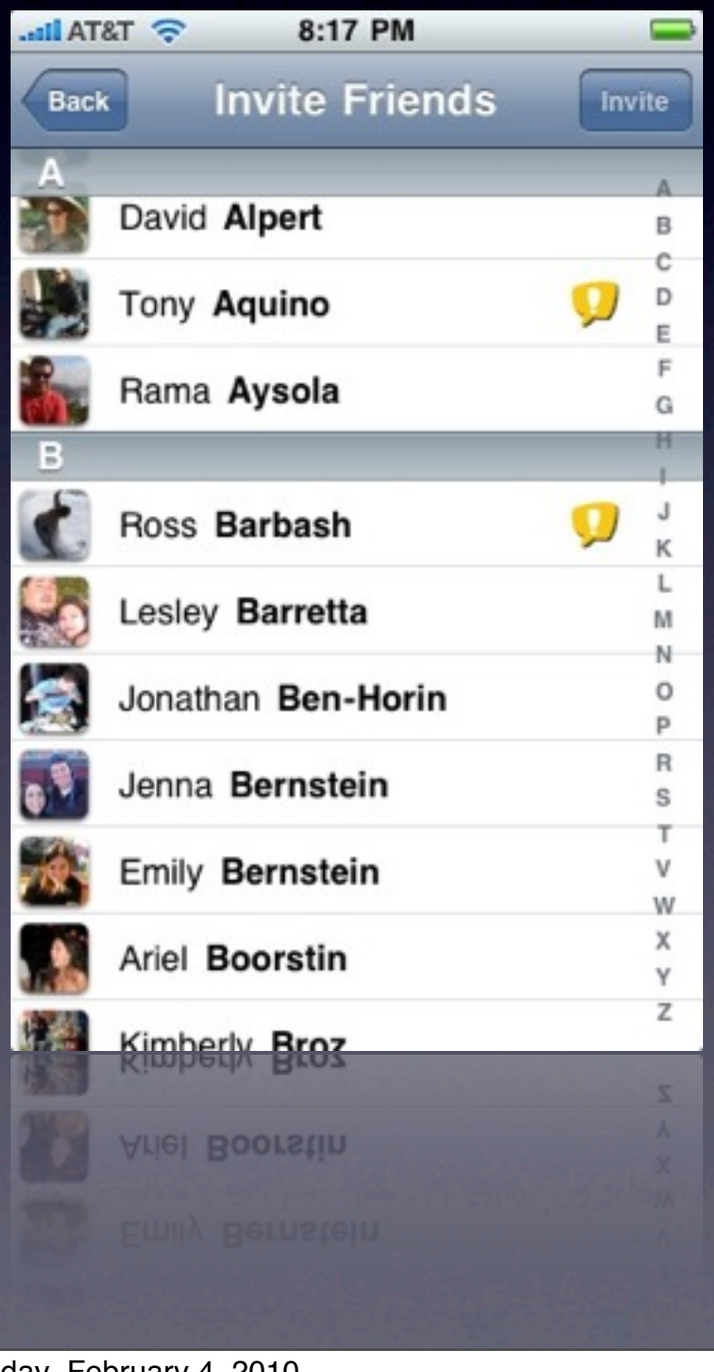
```
- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath {

  /* Do something special with the disclosure button */

}
```

- Use this to make different actions for a row-touch vs. disclosure-touch

Table Index



- For long, multi-section tables
- Allows quick-jump to remote sections

```
- sectionIndexTitlesForTableView:
- tableView:sectionForSectionIndexTitle:atIndex:
```

Editing the Table

There is an entire suite of delegate functions to help dynamically edit a table (add item, delete item, move items around)

- tableView:willBeginEditingRowAtIndexPath:
- tableView:didEndEditingRowAtIndexPath:
- tableView:editingStyleForRowAtIndexPath:
- tableView:titleForDeleteConfirmationButtonForRowAtIndexPath:
- tableView:shouldIndentWhileEditingRowAtIndexPath:

- tableView:canEditRowAtIndexPath:
- tableView:commitEditingStyle:forRowAtIndexPath:
- tableView:canMoveRowAtIndexPath:
- tableView:moveRowAtIndexPath:toIndexPath:

UITableViewController

- There is a special type of UIViewController subclass: UITableViewController
- This automatically creates its base view (self.view) as a UITableView, and assigns the delegates to itself
- Problem: since the base view is a table, you cannot add any stationary floating views on top of it