Views & Data in Your iPhone App

448460-1
Fall 2011
11/03/2011
Kyoung Shin Park
Multimedia Engineering
Dankook University

Overview

- Views
  - Image View & Web View
    - UIViews for displaying images or web content
  - Scroll View
    - **contentSize** property, understanding that its **bounds** is its visible area
  - Table View
    - UITableView styles (plain and grouped)
    - UITableViewDataSource (number of sectiuons, row, and loading up a cell to display)
    - UITableViewCell (how it is reused, properties that can be set on it)
    - UITableViewDelegate (other customization for the table via a delegate)
    - UITableViewController
- Data in your iPhone application
  - Saving & loading local data
  - Accessing remote data over the Internet

Image View & Web View

UIView's frame

- Who's responsible for setting a UIView's frame?
  - The object that puts the UIView in a view hierarchy
- What about when you use alloc/initWithFrame:?
  - If you're putting it into a view hierarchy right away, pick the appropriate frame
  - If you are not, then it doesn't really matter what frame you choose
  - This is because the code that eventually does put you in a view hierarchy will have to set it
- When creating your view with initWithFrame in loadView?
  - A good default to pick is [[UIScreen mainScreen] applicationFrame]
  - This rectangle represents the part of the screen available to applications
  - This is what view controllers (e.g. Navigation, TabBar, Split) do
  - So you don't have to set their view's frame in application:didFinishLaunchingWithOptions

UIImageView

- What can you do with a UIImage?
  - Use it in your drawRect:
  - Ask a UIImageView to draw it
- UIImageView
  - Create it by dragging it out from Interface Builder
  - Or using code

```
UIImageView *imageView = [[UIImageView alloc] initWithImage:
                                              (UIImage *)image];
```

- You can change the image in the UIImageView at any time
  - Note that setting the image after initialization does not modify the UIImageView's frame

```
@property (retain) UIImage *image;
```

UIImageView

- UIImageView also has a highlighted option

```
@property BOOL highlighted;
@property (retain) UIImage *highlightedImage;
UIImageView * imageView = [[UIImageView alloc]
                                    initWithImage: (UIImage *) image
                                    highlightedImage: (UIImage *)];
```

- And it can animate a sequence of images

```
@property (retain) NSArray *animationImages; // of UIImage
@property (retain) NSArray *highlightedAnimationImages;
@property NSTimeInterval animationDuration;
@property NSInteger animationRepeatCount;
@property BOOL isAnimating;
-(void) startAnimating;
-(void) stopAnimating;
```

UIWebView

- Web content can be displayed with UIWebView
  - Also can be used to display PDF's and other complex document
- Content can be
  - **Local HTML string**
  - **Local raw data + MIME type**
  - **Remote URL**
- Leverage WebKit
  - An open source HTML rendering framework (started by Apple)
  - Also use delegate to control/observe user's clicking
  - Can prevent clicking through certain links or respond to user's navigation
  - Supports JavaScript
    - But limited to 5 seconds & 10 MB of memory allocation

UIWebView

- Three ways to load up HTML

```
-(void) loadRequest: (NSURLRequest *)request;
-(void) loadHTMLString: (NSString *)string
        baseURL: (NSURL *) baseURL;
-(void) loadData: (NSData *)data MIMEType: (NSString *)MIMEtype
        textEncodingName: (NSString *)encodingName
        baseURL: (NSURL *)baseURL
```

  - **Base URL** is the "environment" to load resources out of
    - It's the base URL for relative URL's in the data or HTML string
  - **MIME type** says how to interpret the passed-in data
    - Multimedia Internet Mail Extension
    - Standard way to denote file types (like PDF)
    - Think "email attachments" (that's where the name MIME comes from)

UIWebView

- NSURLRequest
  - Encapculates a URL to load and caching policy for fetched data

```
+(NSURLRequest *) requestWithURL: (NSURL *)url;
+(NSURLRequest *) requestWithURL: (NSURL *)url
                     cachePolicy: (NSURLRequestCachePolicy)policy
                 timeoutInterval: (NSTimeInterval)timeoutInterval;
```

- NSURL (like an NSString, but enforced to be well-formed)
  - Example: file://... Or http://...

```
+(NSURL *) urlWithString: (NSString *)urlString;
+(NSURL *) fileURLWithPath: (NSString *)path
               isDirectory: (BOOL)isDirectory;
```

- NSURLRequestCachePolicy
  - Ignore local cache; ignore caches on internet; use expired caches
  - Use cache only (don't go out onto the internet); use cache only if validated

UIWebView

- Putting "browser" user-interface around a web view

```
@property (readonly) BOOL loading;
@property (readonly) BOOL canGoBack;
@property (readonly) BOOL canGoForward;
-(void)reload;
-(void)stopLoading;
-(void)goBack;
-(void)goForward;
```

- Controlling the display of the web view

```
@property BOOL scalesPageToFit; // default is NO
@property UIDataDetectorTypes dataDetectorTypes;
```

  - UIDataDetectorTypePhoneNumber/Link/Address/CalendarEvent
  - Will automatically open appropriate application if user clicks on these "detected" data elements

UIWebViewDelegate

- Delegate for navigation management

```
-(BOOL)webView: (UIWebView *)sender
    shouldStartLoadWithRequest: (NSURLRequest *)request
    navigationType: (UIWebViewNavigationType)navigationType;
```

  - NavigationType specifies things like link clicked, reload form submitted, back/forward, or other
    - UIWebViewNavigationTypeLinkClicked
    - UIWebViewNavigationTypeFormSubmitted
- Delegate for load progress

```
-(void)webViewDidStartLoad: (UIWebView *)sender;
-(void)webViewDidFinishLoad: (UIWebView *)sender;
-(void)webView: (UIWebView *)sender
    didFailLoadWithError: (NSError *)error;
```

Scroll View

UIScrollView

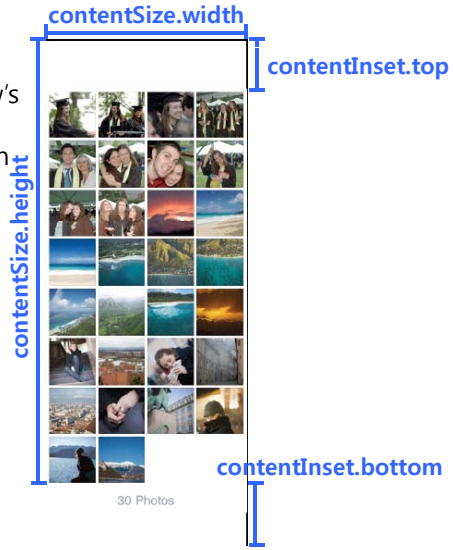
- For displaying more content than can fit on the screen
- Handles gestures for panning and zooming
  - Pans & zooms around a predefined size containing its subviews
- Two important subclasses: **UITableView** & **UITextView**

Content Size & Inset

@property CGSize contentSize;

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.

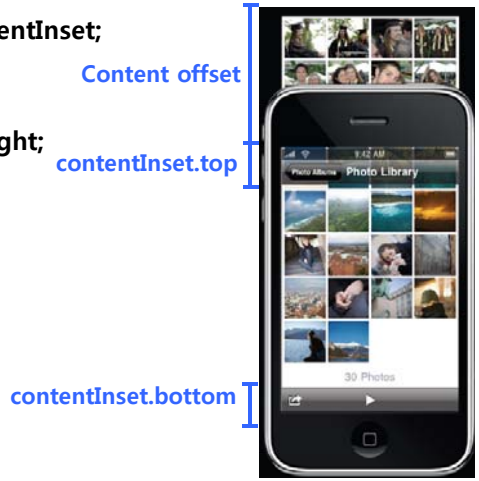
Usually the UIScrollView only has one subview which fills the entire space, i.e., that subview's frame is usually:
origin = (0, 0)
size = scrollView.contentSize



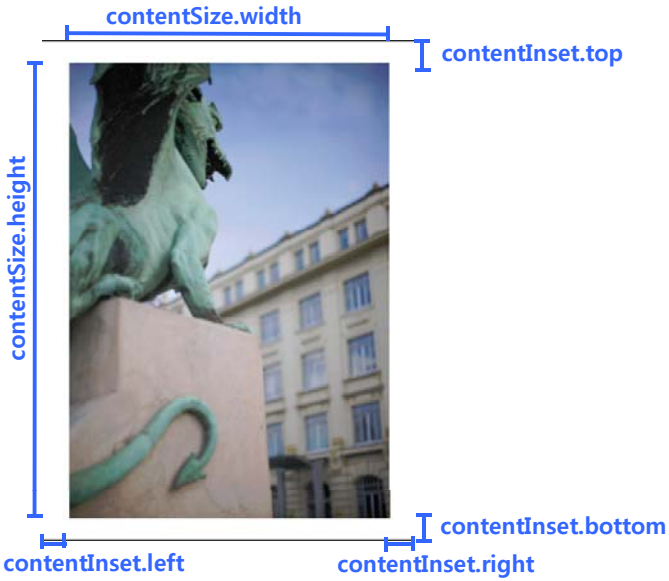
Content Offset

@property CGSize contentSize;
@property UIEndgeInsets contentInset;

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



Content Size & Inset



Content Offset

@property CGSize contentOffset;

Content Offset

UIScrollView

- Create a scroll view

```
CGRect frame = [[UIScreen mainScreen] applicationFrame];
UIScrollView * scrollView = [[UIScrollView alloc]
                                        initWithFrame:frame];
[window addSubview:scrollView];
```

- Add your "too big" UIView as subviews
  - Frames may extend beyond scroll view bounds

```
UIImage *image = [UIImage imageNamed:@"bigimage.jpg"];
UIImageView *imageView = [[UIImageView alloc]
                                        initWithImage:image];
[scrollView addSubview:imageView];
```

- Set the content size (the scroll view's scrollable area's size)

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

UIScrollView

- Bouncing
  - When user scrolls/zooms "too far" it bounds back
- Constraining scrolling

```
@property BOOL scrollEnabled;
@property BOOL directionalLockEnabled;
```

- Display of scroll indicators

```
@property UIScrollViewIndicatorStyle indicatorStyle;
@property BOOL showsHorizontal ScrollIndicator;
-(void)flashScrollIndicators;
```

- Programmatic scrolling

```
-(void)scrollRectToVisible: (CGRect)aRect
        animated: (BOOL)animated;
```

Extending Scroll View Behavior

- Applications often want to know about scroll events
  - When the scroll offset is changed
  - When dragging begins & ends
  - When deceleration begins & ends

Extending with a Subclass

- Create a subclass
- Override methods to customize behavior
- Issues with this approach
  - Application logic and behavior is now part of a View class
  - Tedious to write a one-off subclass for every scroll view instance
  - **Your code becomes tightly coupled with superclass**

Extending with Delegation

- Delegate is a separate object
- Clearly defined points of responsibility
  - Change behavior
  - Customize appearance
- Loosely coupled with the object being extended

UIScrollView Delegate

```
@protocol UIScrollViewDelegate<NSObject>

@optional
// respond to interesting events
-(void)scrollViewDidScroll: (UIScrollView *)sender;
...
// influence behavior
-(BOOL)scrollViewShouldScrollToTop: (UIScrollView *)sender;
@end
```

Implementing a Delegate

- Conform to the delegate protocol

```
@interface MyController: NSObject<UIScrollViewDelegate>
```

- Implement all required methods and any optional methods

```
-(void)scrollViewDidScroll: (UIScrollView *)sender
{
    // do something in response to the new scroll position
    if (sender.contentOffset ... ) {
    }
}
```

Zooming with a Scroll View

- Zooming
  - All UIViews have a property (transform) which is an affine transform (translate, scale, rotate)
  - Scroll view just modifies this transform when you zoom
- Set the minimum, maximum, initial zoom scales

```
scrollView.maximumZoomScale = 2.0;
scrollView.minimumZoomScale = scrollView.size.width /
                              myImage.size.width;
```

- Implement delegate method for zooming

```
-(UIView *)viewForZoomingInScrollView: (UIScrollView *)sender {
    return someViewThatWillBeScaled; }
```

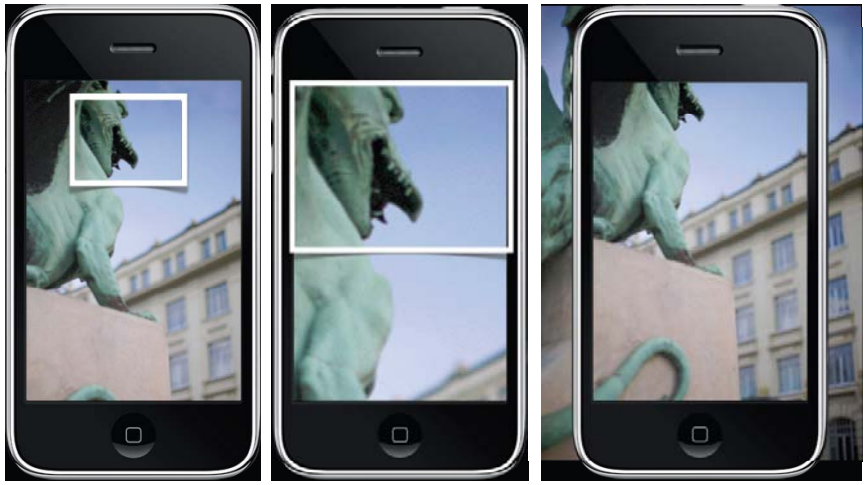
- Another delegate for notifying when zooming ends

```
-(void)scrollViewDidEndZooming: (UIScrollView *)sender
        withView: (UIView *)zoomView atScale: (CGFloat)scale;
```

Set Zoom Scale

-(void) setZoomScale: (float)scale animated: (BOOL)

Zoom to Rect



-(void) zoomToRect: (CGRect)rect animated: (BOOL)

UIScrollView Delegate

- Other delegate methods
  - **-(void)scrollViewDidScroll: (UIScrollView *)sender;**
  - **-(void)scrollViewDidScrollToTop: (UIScrollView *)sender;**
  - **-(void)scrollViewWillBeginZoom: (UIScrollView *)sender withView: (UIView *)zoomView;**
  - **-(void)scrollViewDidEndScrollingAnimation: (UIScrollView *) sender**
  - **-(void)scrollViewWillBeginDecelerating: (UIScrollView *) sender;**
- Property to Find out srolling state
  - **@property (readonly) BOOL zooming;**
  - **@property (readonly) BOOL dragging;**
  - **@property (readonly) BOOL tracking;**
  - **@property (readonly) BOOL decelerating;**

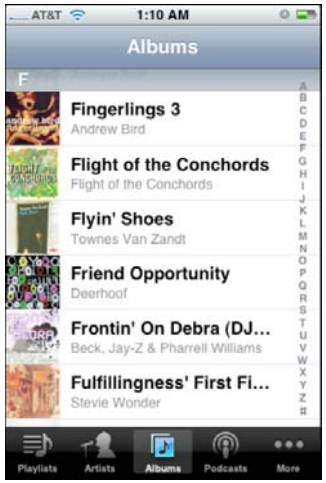
Table Views

UITableView

- Very important class for displaying the lists of content
  - It's a subclass of UIScrollView
  - Lots of customization via **dataSource** and **delegate**
- Can only display one column of data at a time
  - Often table views are pushed from each other's to display a hierarchical data set
  - The column can be divided into sections for user-interface cleanliness or easy access to large lists
- Designated initializer takes the style you want
  - UITableViewStylePlain
  - UITableViewStyleGrouped

Table View Styles

UITableViewStylePlain



UITableViewStyleGrouped

Table View Anatomy

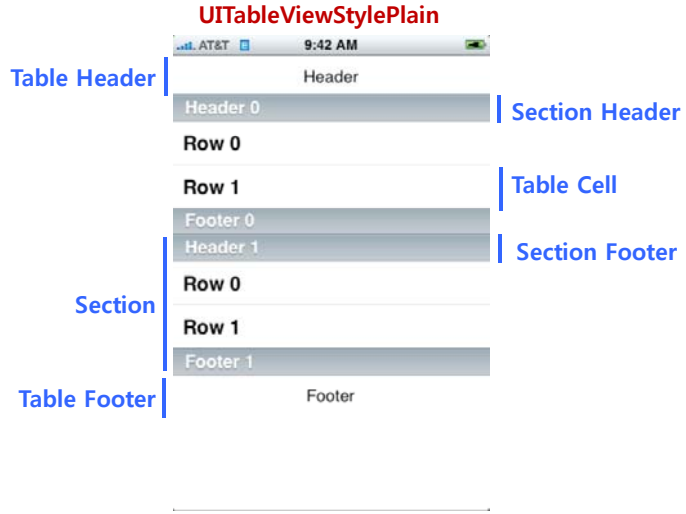
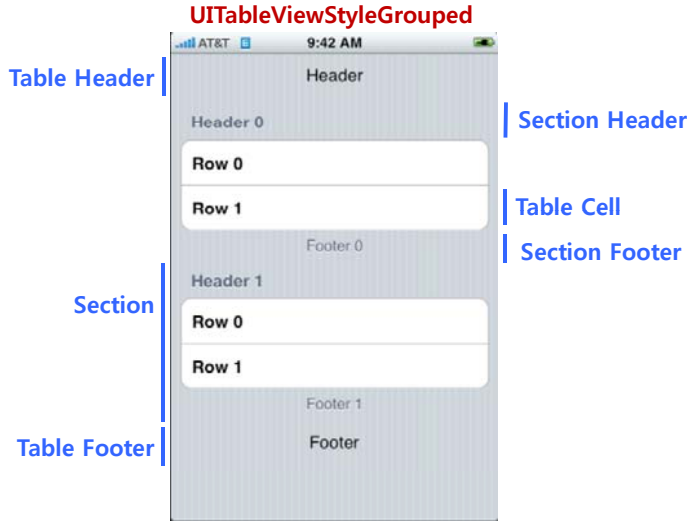


Table View Anatomy



Displaying Data in a Table View : A Naïve Solution

- Table views display a list of data, so use an array
  [myTableView setList:myListOfStuff];
- Issues with this approach
  - All data is loaded upfront
  - All data stays in memory

Displaying Data in a Table View : A More Flexible Solution

- Another object provides data to the table view: delegated
  - **We do NOT want to load all the data at once**
  - **UITableView only asks for data just as it's needed for display**
- **DataSource**
  - Like a delegate, but purely data-oriented

  @property (assign) id <UITableViewDataSource> dataSource;
  - But the table view needs to know the size of the data up front
    - So that it can set the contentSize of itself
  - So the first thing it will ask its dataSource is "how many row?"
    - 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

- Provide number of sections and rows

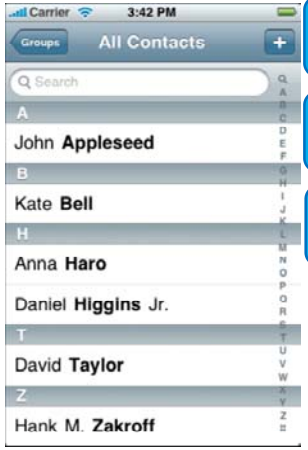
```
// optional method, defaults to 1 if not implemented
-(NSInteger)numberOfSectionsInTableView: (UTableView *)table;
// required method
-(NSInteger)tableView: (UITableView *)tableview
        numberOfRowsInSection: (NSInteger)section;
```

- Provide cells for table view as needed

```
// required method
-(UITableViewCell *)tableView: (UITableView *)tableView
        cellForRowAtIndexPath: (NSIndexPath *)indexPath;
```

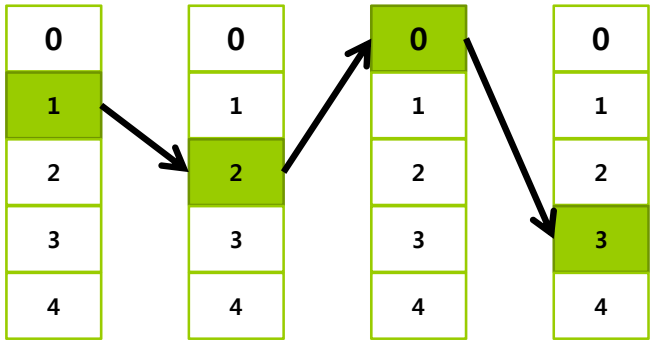
  - An **NSIndexPath** is a way to specifying a section and row
    - **indexPath.section** is the section, **indexPath.row** is the row

Datasource Message Flow



NSIndexPath

- Generic class in Foundation
- Path to a specific node in a tree of nested arrays



NSIndexPath and Table Views

- Cell location described with an index path
  - Section index + row index
- Category on NSIndexPath with helper methods

```
@interface NSIndexPath (UITableView)
+(NSIndexPath *)indexPathForRow: (NSUInteger)row
                      inSection: (NSUInteger)section;
@property (nonatomic, readonly) NSUInteger section;
@property (nonatomic, readonly) NSUInteger row;
@end
```

Single Section Table View

E.g., Data is stored in an NSArray of NSString

- Return the number of rows

```
-(NSInteger)tableView: (UITableView *)sender
                numberOfRowsInSection: (NSInterger)section {
    return myArray.count; // no need to check section here
}
```

- Provide a cell when required

```
-(UITableViewCell *)tableView: (UITableView *)sender
          cellForRowAtIndexPath: (NSIndexPath *) indexPath
{
    UITableViewCell *cell = ...;
    cell.textLabel.text = [myArray objectAtIndex:indexPath.row];
    return [cell autorelease];
}
```

Cell Reuse

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

```
-(UITableViewCell *)
        dequeueReusableCellWithIndentifier: (NSString *) id;
-(UITableViewCell *)tableView: (UITableView *)sender
        cellForRowAtIndexPath: (NSIndexPath *)indexPath {
    UITableViewCell *cell = [sender
        dequeueReusableCellWithIndentifier:@"MyIdentifier"];
    if (cell == null) {
        cell = [[[UITableViewCell alloc] initWithStyle: ...
            reuseIdentifier:@"MyIdentifier"] autorelease];
    }
    cell.textLabel.text = [myArray objectAtIndex:indexPath.row];
    return cell;
}
```

Triggering Updates

- When is the datasource asked for its data?
  - When a row becomes visible
  - When an update is explicitly requested by calling –reloadData

```
-(void)viewWillAppear: (BOOL)animated {
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

Section and Row Reloading

```
-(void)insertSections: (NSIndexSet *)sections
        withRowAnimation: (UITableViewRowAnimation)animation;
-(void)deleteSections: (NSIndexSet *)sections
        withRowAnimation: (UITableViewRowAnimation)animation;
-(void)reloadSections: (NSIndexSet *)sections
        withRowAnimation: (UITableViewRowAnimation)animation;

-(void)insertRowsAtIndexPaths: (NSArray *)indexPaths
        withRowAnimation: (UITableViewRowAnimation)animation;
-(void)deleteRowsAtIndexPaths: (NSArray *)indexPaths
        withRowAnimation: (UITableViewRowAnimation)animation;
-(void)reloadRowsAtIndexPaths: (NSArray *)indexPaths
        withRowAnimation: (UITableViewRowAnimation)animation;
```

Additional DataSource Methods

- Titles for section headers and footers
  - This is the dataSource providing data for what's in the header (similar footer methods)

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

- Allow editing and reordering cells
  - Asking the dataSource to actually edit the underlying data

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

  - Asking the dataSource to move rows in the underlying data

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

UITableViewDelegate

- The **delegate** controls **how** the UITableView is displayed
  - Not **what** it displays (that's the **dataSource**'s job)
  - It customizes the table view's appearance and behavior
- The delegate also lets you observe what the table view is doing
  - Especially responding to when the user selects a row
  - It keeps application logic separate from view
- 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.

Table View Appearance & Behavior

- Customize appearance of table view cell

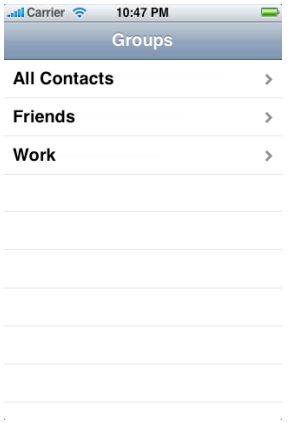
```
-(void)tableView: (UITableView *)sender
    willDisplayCell: (UITableViewCell *)cell
    forRowAtIndexPath: (NSIndexPath *)indexPath;
```

- Validate and respond to selection changes

```
-(NSIndexPath *)tableView: (UITableView *)sender
    willSelectRowAtIndexPath: (NSIndexPath *)indexPath;
-(void)tableView: (UITableView *)sender
    didSelectRowAtIndexPath: (NSIndexPath *)indexPath {
  MyViewController *vc = [[MyViewController alloc] init];
  vc.someProperty = .... //something dependent on my data
  [self.navigationController pushViewController:vc animated:YES];
  [vc release];
}
```

Row Selection in Table Views

- In iPhone applications, rows rarely stay selected
- Selecting a row usually triggers an event



Responding to Selection

```
// for a navigation hierarchy
-(void)tableView: (UITableView *)sender
     didSelectRowAtIndexPath: (NSIndexPath *)indexPath {
    // get the row and the object it represents
    NSUInteger row = indexPath.row;
    id objectToDisplay = [myObjects objetAtIndex:row];
    // create a new view controller and pass it along
    MyViewController *vc = ...;
    vc.object = objectToDisplay;
    [self.navigationController pushViewController:vc animated:YES];
}
```

Altering or Disabling Selection

```
-(NSIndexPath *)tableView: (UITableView *)sender
     willSelectRowAtIndexPath: (NSIndexPath *)indexPath {
    // don't allow selecting certain row?
    if (indexPath.row == ...) {
        return nil;
    } else {
        return indexPath;
    }
}
```

UITableViewController

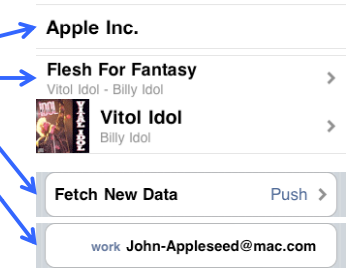
- Convenient staring point for view controller with a table view
  - Table view is automatically created
  - Controller is table view's delegate and datasource
- Takes care of some default behaviors
  - Calls **–reloadData** the first time it appears
  - Deselects rows when user navigates back
  - Flashes scroll indicators

UITableViewCell

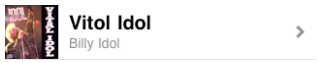
- You set it up to display the data in a given row
  - **@property (readonly) UILabel *textLabel;**
  - **@property (readonly) UILabel *detailTextLabel;**
  - **@property (readonly) UIImageView *imageView;**
- Designated initializer takes the style of the cell (no frame)

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

  - Cell Styles
    - UITableViewCellStyleDefault
    - UITableViewCellStyleSubtitle
    - UITableViewCellStyleValue1
    - UITableViewCellStyleValue2



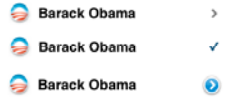
UITableViewCell



- Basic properties
  - UITableViewCell has an image view and one or two text labels

```
cell.imageView.image = [UIImage imagedNamed:@"vitolidol.png"];
```

- Accessory types
  - UITableViewCellAccessoryDisclosureIndicator
  - UITableViewCellAccessoryCheckmark
  - UITableViewCellAccessoryDetailDisclosureButton




```
-(UITableViewCellAccessoryType)tableView: (UITableView *)sender
      accessoryTypeForRowWithIndexPath: (NSIndexPath *)indexPath;
-(void)tableView: (UITableView *)sender
     accessoryButtonTappedForRowWithIndexPath: (NSIndexPath *)
     indexPath {
    NSUInteger row = indexPath.row; // ...
}
```

Customizing the Content View

- For cases where a simple image + text cell doesn't suffice
- UITableViewCell has a content view property
  - Add additional views to the content view

```
-(UITableViewCell *) tableView: (UITableView *) tableView
    cellForRowAtIndexPath: (NSIndexPath *) indexPath {
    UITableViewCell *cell = ....;
    CGRect frame = cell.contentView.bounds;
    UILabel *myLabel = [[UILabel alloc] initWithFrame:frame];
    myLabel.text = ...;
    [cell.contentView addSubview:myLabel];
    [myLabel release];
}
```

Data in Your iPhone App

Data in Your iPhone Application

- Property Lists, NSUserDefaults and Settings
  - Quick & easy, but limited
- iPhone's File System
- Archiving Objects
  - More flexible, but require writing a lot of code
- SQLite
  - Elegant solution for many types of problems
- XML and JSON
  - Low-overhead options for talking to "the cloud"
  - Apple Push Notification Service pushes JSON from your server to devices

Property Lists

- Convenient way to **store a small amount of data**
  - NSArray, NSDictionary, NSString, NSNumber, NSDate, NSData
- **NSUserDefaults** class uses property lists under the hood
  - Also three formats for storing in files or reading from internet via a URL
    - XML
    - Binary
    - "Old-Style" ASCII (deprecated)

When Not to User Property Lists

- More than a few hundred KB of data
  - Loading a property list is all-or-nothing
- Complex object graphs
- Custom object types
- Multiple writers (e.g. not ACID)

NSPropertyListSerialization

- Allows finer-grained control
  - File format
  - More descriptive errors
  - Mutability

```
// property list to NSData
+(NSData *) dataFromPropertyList: (id)plist
        format: (NSPropertyListFormat)format // XML or binary
        errorDescription: (NSString **)errorString; // 0 if unused
// NSData to property list
+(id) propertyListFromData: (NSData *)data
        mutabilityOption: (NSPropertyListMutabilityOptions)opt
        format: (NSPropertyListFormat *)format
        errorDescription: (NSString **)errorString;
```

Reading & Writing Property Lists

- NSArray and NSDictionary convenience methods
- To write a property list to a file
  - Use NSPropertyListSerialization to get an NSData, then use this NSData method

```
// writing
+(BOOL) writeToFile: (NSString *)aPath atomically: (BOOL)flag;
+(BOOL) writeToURL: (NSURL *)aURL atomically: (BOOL)flag;
```

- To read a property list NSData from a URL/File
  - Get the NSData from a URL/File, then use NSPropertyListSerialization to turn the NSData to a property list

```
// reading
-(id) initWithContensOfFile: (NSString *)aPath;
-(id) initWithContentsOfURL: (NSURL *)aURL;
```

Writing an Array to Disk

```
NSArray *array = [NSArray arrayWithObjects:@"Foo",
                  [NSNumber numberWithBool:YES],
                  [NSDate dateWithTimeIntervalSinceNow:60],
                  nil];
[array writeToFile:@"MyArray.plist" atomically:YES];
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>Foo</string>
    <true/>
    <date>2010-02-02T09:26:18Z</date>
</array>
</plist>
```

Writing a Dictionary to Disk

```
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
                  @"Bob", @"Name",
                  [NSNumber numberWithInt:9], @"Lecture",
                  nil];
[dict writeToFile:@"MyDict.plist" atomically:YES];
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>Foo</string>
    <true/>
    <date>2010-02-02T09:26:18Z</date>
</array>
</plist>
```

Keeping Applications Separate

- Security
- Privacy
- Cleanup after deleting an application

NSFileManager

- NSFileManager
  - Provides utility operations (reading and writing is done via NSData, et al)
  - Check to see if files existl create and enumerate directories; move, copy, delete files
  - Just alloc/init an instance and start performing operations.

```
NSFileManager *manager = [[NSFileManager alloc] init];
- (BOOL)createDirectoryAtPath:(NSString *)path
      withIntermediateDirectories:(BOOL)createIntermediates
      attributes:(NSDictionary *)attributes // permissions, etc.
      error:(NSError **)error;
- (BOOL)isReadableFileAtPath:(NSString *)path;
- (NSArray *)contentsOfDirectoryAtPath:(NSString *)path
      error:(NSError **)error;
```

Home Directory Layout

- **Each application has its own set of directories**
- <Application Home>
  - MyApp.app
    - MyApp
    - MainWindow.nib
    - SomeImage.png
  - Documents
  - Library
    - Caches
    - Preferences
- Applications only read and write within their home directory
- Backed up by iTunes during sync (mostly)

File Paths in Your iPhone Application

```
// basic directories
NSString *homePath = NSHomeDirectory();
NSString *tmpPath = NSTemporaryDirectory();

// documents directory
NSArray *paths = NSSearchPathForDirectoriesInDomains
                    (NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsPath = [paths objectAtIndex:0];

// <Application Home>/Documents/foo.plist
NSString *fooPath = [documentsPath
                        stringByAppendingPathComponent:@"foo.plist"];
```

Including Writable Files with Your App

- Many applications want to include some starter data
- But application bundles are code signed
  - You can't modify the contents of your app bundle
- To include a writable data file with your app
  - Build it as part of your app bundle
  - **On first launch, copy it to your Documents directory**

Archiving Objects

- Next logical step from property lists
  - Include arbitrary classes (not just graphs with NSArray, NSDictionary, etc)
  - Complex object graphs
- Used by Interface Builder for NIBs

Making Objects Archivable

- Conform to the <NSCoding> protocol

```
// save: encode an object for an archive
-(void) encodeWithCoder: (NSCoder *)coder {
    [super encodeWithCoder:coder]; // must call super's version
    [coder encodeObject:name forKey:@"Name"];
    [coder encodeInteger:numberOfSides forKey:@"Side"];
}
// read: decode an object from an archive
-(id) initWithCoder: (NSCoder *)coder {
    self = [super initWithCoder:coder];
    name = [[coder decodeObjectForKey:@"Name"] retain]; // retain!
    // note that order does not matter
    numberOfSides = [coder decodeIntegerForKey:@"Side"];
}
```

Archiving & Unarchiving Objects Graphs

- Creating an archive

```
NSArray *polygons = ...;
NSString *path = ...;
BOOL result = [NSKeyedArchiver archiveRootObject:polygons
                                          toFile:path];
```

- Decoding an archive

```
NSArray *polygons = nil;
NSString *path = ...;
polygons = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
```

SQLite

- Complete SQL database in an ordinary file
- Simple, compact, fast, reliable
- No server
- Free/Open Source Software
- Great for embedded devices
  - Included on the iPhone platform

When Not to Use SQLite

- Multi-gigabyte databases
- High concurrency (multiple writers)
- Client-server applications
- "Appropriate Uses for SQLite"
  - http://www.sqlite.org/whentouse.html

SQLite C API Basics

- Open the database

```
int sqlite3_open(const char *filename, sqlite3 **db);
```

- Execute a SQL statement

```
int sqlite3_exec(sqlite3 *db, const char *sql,
                 int (*callback)(void*, int, char**, char**),
                 void *context, char **error);
// your callback
int callback_func(void *context, int count,
                  char **values, char **columns);
```

- Close the database

```
Int sqlite3_close(sqlite3 *db);
```

Core Data

- Object-graph management and persistence framework
  - Makes it easy to save & load model objects
    - Properties
    - Relationships
  - Higher-level abstraction than SQLite or property lists
- Available on the Mac OS X desktop
- Available on iPhone OS 3.0 or higher

Two classes you should know about…

- **NSPredicate**
  - Used to define logical conditions used to constrain a search either for a fetch or for in-memory filtering

  `-[NSPredicate predicateWithFormat:]`
  - Simple comparions:
    - Grade == "7"
    - User.firstName like "Tome"
    - "first contains [c]" chris"
- **NSEntityDescription**
  - Used for inserting a new object into a Core Data Managed Object context

  `-[NSEntityDescription insertNewObjectForEntityForName: inManagedObjectContext:]`

Your Application & The Cloud

- Store & access remote data
- May be under your control or someone else's
- Many Web 2.0 apps/sites provide developer API

Integrating with Web Services

- Many are exposed via RESTful interfaces with XML or JSON
  - REpresentational State Transfer
    - Stateless interactions
    - Well defined client/server roles & interfaces
    - E.g. HTTP
  - High level overview of parsing these types of data

Options for Parsing XML

- Libxml2
  - Tree-based: easy to parse, entire tree in memory
  - Event-driven: less memory, more complex to manage state
  - Text reader: fast, easy to write, efficient
- NSXMLParser
  - Event-driven API: simpler but less powerful than libxml2

JavaScript Object Notification

- More lightweight than XML
- Looks a lot like a property list
  - Arrays, dictionaries, strings, numbers
- Open source json-framework wrapper for Objective-C

What does a JSON object look like?

```
{
   "instructor" : "Kyoung Shin Park",
   "students" : 10,
   "itunes-u" : false,
   "midterm-exam" : null,
   "assignments" : ["Assignment1", "Assignment2", "Assignment3"]
}
```

Using json-framework

- Reading a JSON string into Foundation objects

```
@import <JSON/JSON.h>
// get a JSON string from the cloud
NSString *jsonString = ...;
// parsing will result in Foundation objects
// top level may be an NSDictionary or an NSArray
id object = [jsonString JSONValue];
```

- Writing a JSON string from Foundation objects

```
// create some data in your app
NSDictionary *dict = ...;
// convert into a JSON string before sending to the cloud
jsonString = [dict JSONRepresentation];
```

NSUserDefaults

- Convenient way to store settings and lightweight state
  - Arrays, dictionaries, strings, numbers, dates, raw data
  - Settings bundles can be created so that user defaults can be set from Settings app
  - Internally stored as property lists

Reading & Writing User Defaults

- Key-value store
- Base methods accept and return objects for values

```
+(NSUserDefaults *)standardUserDefaults;
-(id)objectForKey: (NSString *)defaultName;
-(void)setObject: (id)value forKey: (NSString *)defaultName;
-(void)removeObjectForKey: (NSSTring *)defaultName;
-(BOOL)synchronize;
```

- Many convenience methods that 'box' and 'unbox' the obje and perform type checking

```
-(NSString *)stringForKey: (NSString *)defaultName;
-(NSArray *)arrayForKey: (NSString *)defaultName;
-(NSInterger)integerForKey: (NSString *)defaultName;
-(void)setInteger: (NSInteger)value forKey: (NSString *)defaultNam
-(void)setFloat: (float)value forKey: (NSString *)defaultName;
```

-[NSUserDefaults synchronize]

- Call [[NSUserDefaults standardUserDefaults] synchronize] to write changes to disk
- Also loads external changes from disk (useful on Mac OS X)

References

- Lecture 8 & 9 & 12 Slide from iPhone Application Development (Winter 2010) @Stanford University