Views &
Data in Your iPhone App

448460-1
Fall 2015
11/09/2015
Kyoung Shin Park
Multimedia Engineering
Dankook University

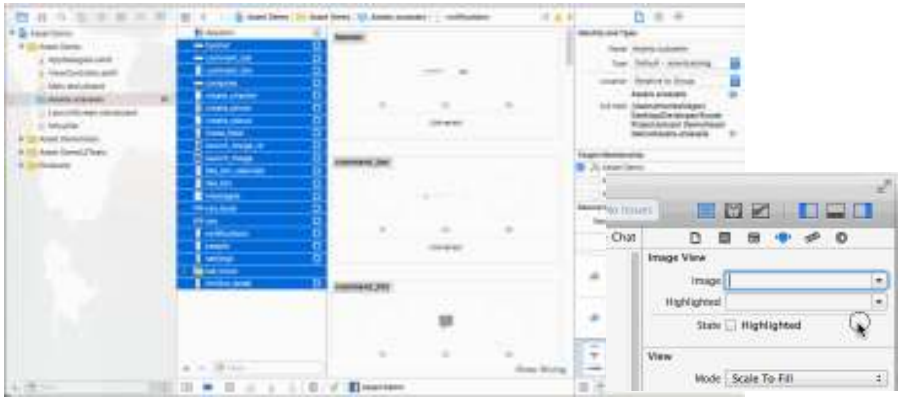
Overview

- 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

Adding Image Assets

- Select the asset catalog (Assets.xcassets)
- Add the image set by drag and drop
- Using the image set in Interface Builder



UIImageView

- UIImageView
  - Create it by dragging it out from Interface Builder

```
@IBOutlet weak var imageView: UIImageView!
override func viewDidLoad() {
    super.viewDidLoad()
    imageView.image = UIImage(named: "img.png")
}
```

  - Or using code programmatically

```
let imageView = UIImageView(frame: CGRectMake(10,10,300,500))
imageView.image = UIImage(named: "img.png")
self.view.addSubview(imageView)
```

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

UIImageView

- UIImageView also can animate a sequence of images

```
var images: [UIImage] = [ ]  // Array of UIImage
for i in 1..11 {
    images.append(UIImage(named: "frame₩(i)")!)
}
imageView.animationImages = images
imageView.animationDuration = 5
imageView.animationRepeatCount = 0
imageView.startAnimating()

imageView.isAnimating()
imageView.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

```
loadRequest:
loadHTMLString: baseURL:
loadData: MIMEType: textEncodingName: 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

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

  urlWithString:
  fileURLWithPath: 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

- UIWebView is used to load and display **the direct url web content** into your application

```
@IBOutlet weak var myWebView: UIWebView!

override func viewDidLoad() {
    super.viewDidLoad()
    // add NSURL & request URL into webview
    let url = NSURL(string: "http://dis.dankook.ac.kr/lectures/")
    let requestObject = NSURLRequest(URL: url!)
    myWebView.loadRequest(requestObject)
}
```


UIWebView

- UIWebView is used to load **the local html file** into webview

```
override func viewDidLoad() {
    super.viewDidLoad()
    // add the local html file ("index.html") into webview
    let localFilePath =
NSBundle.mainBundle().URLForResource("index", withExtension:
"html")
    let requestObject = NSURLRequest(URL: localFilePath!)
    myWebView.loadRequest(requestObject)
}
```


UIWebView

- UIWebView is used to **load HTML String** into webview

```
override func viewDidLoad() {
    super.viewDidLoad()
    // add String into webview
    var htmlString: String! = "<h2>Lecture Note for Design of
Portable Multimedia Device</h2>"
    myWebView.loadHTMLString(htmlString, baseURL: nil)
}
```

UIWebView

- UIWebView is used to **load the document files (e.g. pdf, .txt, .doc, etc)** into webview

```
override func viewDidLoad() {
    super.viewDidLoad()
    // add document & load data into webview
    let localFilePath =
NSBundle.mainBundle().pathForResource("test", ofType: "txt")
    let data =
NSFileManager.defaultManager().contentsAtPath(localFilePath!)
    myWebView.loadData(data!, MIMEType: "application/txt",
textEncodingName: "UTF-8", baseURL: nil)
}
```


UIWebView

- UIWebView navigation operations
  - myWebView.reload() // reload the current web page
  - myWebView.stopLoading() // stop the web content
  - myWebView.goBack() // load the previous web page
  - myWebView.goForward() // load the forward web page
- Controlling the display of the web view
  - myWebView.scalesPageToFit = YES // default NO
    - If yes, the web page allows users to perform zoom in and zoom out operations.
  - myWebView.dataDetectorTypes
    - Used to analyze the loaded webpages having any links, email or phone numbers and events (added to calendar).
    - Will automatically open appropriate application if user clicks on these "detected" data elements

UIWebViewDelegate

- Delegate for navigation management
  - webView: shouldStartLoadWithRequest: navigationType:
    - NavigationType specifies things like link clicked, reload form submitted, back/forward, or other
      - UIWebViewNavigationTypeLinkClicked
      - UIWebViewNavigationTypeFormSubmitted
- Delegate for load progress
  - webViewDidStartLoad:
  - webViewDidFinishLoad:
  - webView: didFailLoadWithError:

App Transport Security (ATS)

- *App Transport Security (ATS)* ***lets an app add a declaration to its Info.plist file*** *that specifies the domains with which it needs secure communication.*

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

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:

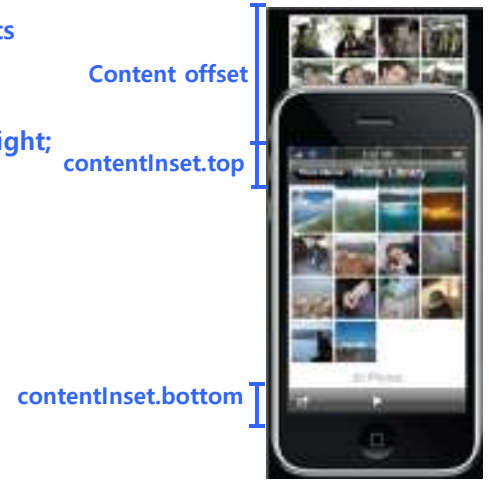
origin = (0, 0)

size = scrollView.contentSize

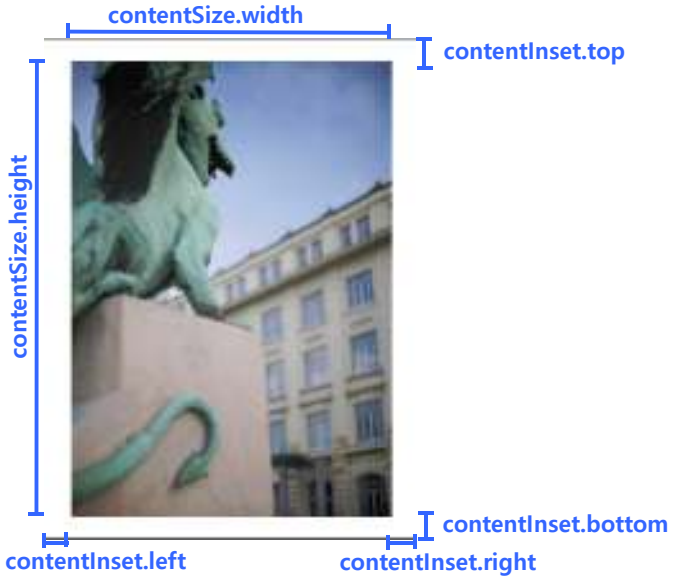
Content Offset

var contentSize: CGSize
var contentInset: UIEndgeInsets

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



Content Size & Inset



Content Offset

var contentOffset: CGSize

Content Offset

UIScrollView

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
let visibleRect: CGRect = aerial.convertRect(scrollView.bounds,
fromView: scrollView)
```

UIScrollView

- **Create a scroll view**
  - Just like any other UIView. Drag out in a storyboard or use UIScrollView(frame:)
  - Or select a UIView in your storyboard and choose ""Embed In -> Scroll View" from Editor menu
- **Add your "too big" UIView as subviews**
  - Frames may extend beyond scroll view bounds

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

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

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

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

```
// (optional) respond to interesting events
func scrollViewDidScroll(sender: UIScrollView)
...
// influence behavior
func scrollViewShouldScrollToTop(sender: UIScrollView) -> Bool
```

Implementing a Delegate

- Conform to the delegate protocol **UIScrollViewDelegate**
- Implement all required methods and any optional methods

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

Zooming with a Scroll View

- 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 also going to affect the scroll view's contentSize and contentOffset
- Set the minimum, maximum, initial zoom scales

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

- Implement delegate method for zooming

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

- Another delegate for notifying when zooming ends

```
func scrollViewDidEndZooming(sender: UIScrollView,
        withView zoomView: UIView, atScale scale: CGFloat)
```

Set Zoom Scale

```
func setZoomScale(scale: CGFloat, animated: Bool)
```

Zoom to Rect

func zoomToRect(rect: CGRect, animated: Bool)

UIScrollView Delegate

- Other delegate methods
  - func scrollViewDidScroll(sender: UIScrollView)
  - func scrollViewDidScrollToTop(sender: UIScrollView)
  - func scrollViewWillBeginZoom(sender: UIScrollView, withView zoomView: UIView)
  - func scrollViewDidEndScrollingAnimation(sender: UIScrollView)
  - func scrollViewWillBeginDecelerating(sender: UIScrollView)
- Property to Find out scrolling state
  - zooming: Bool
  - dragging: Bool
  - tracking: Bool
  - decelerating: Bool

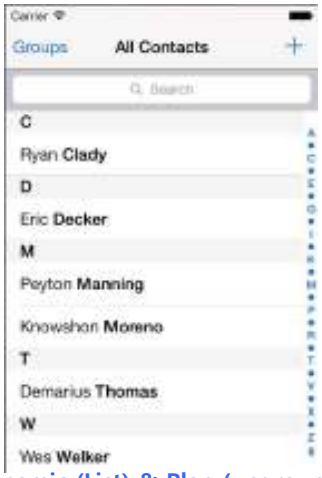
Table Views

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
  - **UITableViewStyle.Plain**
  - **UITableViewStyle.Grouped**

Table View Styles

UITableViewStyle.Plain



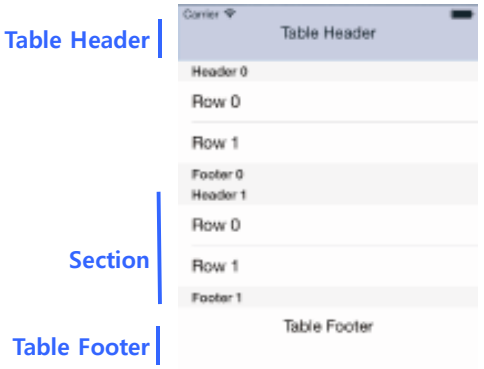
Dynamic (List) & Plan (ungrouped)

UITableViewStyle.Grouped

Static & Grouped

Table View Anatomy

UITableViewStyle.Plain



var tableHeaderView: UIView
var tableFooterView: UIView
DataSource's tableView(UITableView, titleForHeaderInSection: Int)
DataSource's tableView(UITableView, titleForFooterInSection: Int)
DataSource's tableView(UITableView, cellForRowAtIndexPath: NSIndexPath)

Table View Anatomy

UITableViewStyle.Grouped

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**
- **UITableViewDataSource**
  - Like a delegate, but purely data-oriented
  - 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

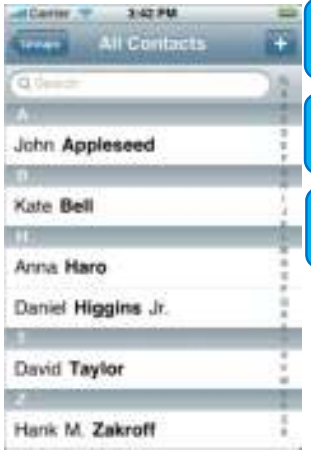
```
// optional method, defaults to 1 if not implemented
func numberOfSectionsInTableView(tableView: UTableView) ->
Int
// required method
func tableView(tableView: UITableView,
                numberOfRowsInSection section: Int) -> Int
```

- Provide cells for table view as needed

```
// required method
func tableView(tableView: UITableView,
            cellForRowAtIndexPath indexPath: NSIndexPath)
            -> UITableViewCell
```

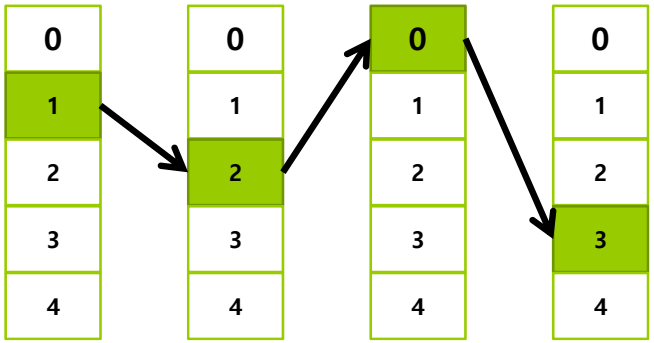
  - An **NSIndexPath** is a way to specifying a section and row
    - **indexPath.section** is the section, **indexPath.row** is the row

Datasource Message Flow



NSIndexPath

- Path to a specific node in a tree of nested arrays
- NSIndexPath enables you to get the represented **row** index and **section** index.



NSIndexPath and Table Views

- Cell location described with an index path
  - Section index + row index
- To construct an NSIndexPath from a given row index and section index

```
var path = NSIndexPath(forRow: index, inSection: 0)
self.tableView.reloadRowsAtIndexPaths([path],
        withRowAnimation: UITableViewRowAnimation.Automatic);
```

Single Section Table View

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

- Return the number of rows

```
func tableView(tableView: UITableView,
                numberOfRowsInSection section: Int) -> Int {
    return myArray.count; // no need to check section here
}
```

- Provide a cell when required

```
func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let data = myArray[indexPath.section][indexPath.row]
    let cell =
        self.tableView.dequeueReusableCellWithIdentifier("cell",
        forIndexPath: indexPath) as! UITableViewCell
    cell.textLabel?.text = data.title
    return cell;
}
```

Cell Reuse

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

```
self.tableView.registerClass(UITableViewCell.self,
                             forCellReuseIdentifier: "cell")

func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let data = myArray[indexPath.section][indexPath.row]
    let dequeued : AnyObject =
        tableView.dequeueReusableCellWithIdentifier("cell",
        forIndexPath: indexPath)
    let cell = dequeued as! UITableViewCell
    cell.textLabel?.text = data.title
    cell.detailTextLabel?.text = data.subtitle
    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**

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    self.tableView.reloadData()
}
```

- **func reloadData()** causes the UITableView to call **numberOfSectionsInTableView** and **numberOfRowsInSection** all over again and then **cellForRowAtIndexPath** on each visible row
- If only part of your Model changes, there are lighter-weight reloaders, **func reloadRowsAtIndexPaths(indexPaths: [NSIndexPath], withRowAnimation:UITableViewRowAnimation)**

Additional DataSource Methods

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

```
func tableView(tableView: UITableView,
    titleFor{Header,Footer}InSection section: Int) -> String
```

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

```
func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath)
```

  - Asking the dataSource to move rows in the underlying data

```
func tableView(tableView: UITableView,
    moveRowAtIndexPath sourceIndexPath: NSIndexPath,
    toIndexPath destinationIndexPath: NSIndexPath)
```

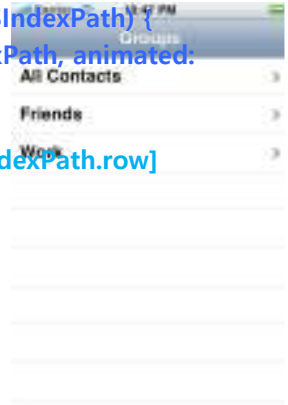
UITableViewDelegate

- The **delegate** controls **how** the UITableView is **displayed**
  - Not **the data** 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
  - Usually you will just segue when this happens, but if you want to track it directly
- 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

UITableView "Target/Action"

- UITableViewDelegate method sent when row is selected
  - This is sort of like "table view target/action" (only needed if you're not segueing, of course)

```
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    tableView.deselectRowAtIndexPath(indexPath, animated:
true)
    //something dependent on my data
    let data = myArray[indexPath.section][indexPath.row]
}
```



Responding to Selection

```
// for a navigation hierarchy
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    tableView.deselectRowAtIndexPath(indexPath, animated: true)
    print("You selected cell ₩(indexPath.row)")
    let data = myArray[indexPath.section][indexPath.row]
    //something dependent on my data
    let myViewController = ..
    navigationController?.pushViewController(myViewController,
                                             animated: true)
}
```

Table View Appearance & Behavior

- Delegate method sent when Detail Disclosure button is touched
  - You can just segue from that detail disclosure button if you prefer

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

- Delegate method to customize appearance of table view cell

```
func tableView(tableView: UITableView,
    willDisplayCell cell: UITableViewCell,
    forRowAtIndexPath index: NSIndexPath)
```

UITableViewController

- UITableViewController provides a convenient starting point for view controller with a table view
  - UITableViewController controller's **view** property is **UITableView**
  - UITableViewController automatically sets itself as the UITableView's delegate and dataSource.
  - Your UITableView Controller subclass will also have a property pointing to the UITableView

  var tableView: UITableView // self.view in UITableViewController
- 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
  - **textLabel: UILabel**
  - **detailTextLabel: UILabel**
  - **imageView: UIImageView**
- Designated initializer takes the style of the cell (no frame)

  UITableViewCell(style: UITableViewCellStyle,
  reuseIdentifier identifier: String!)
  - Cell Styles
    - UITableViewCellStyleDefault
    - UITableViewCellStyleSubtitle
    - UITableViewCellStyleValue1
    - UITableViewCellStyleValue2

UITableViewCell

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

  cell.imageView?.image = UIImage(named:"vitolidol.png")
- Accessory types
  - UITableViewCellAccessoryDisclosureIndicator
  - UITableViewCellAccessoryCheckmark
  - UITableViewCellAccessoryDetailDisclosureButton


```
func tableView(tableView: UITableView,
    accessoryButtonTappedForRowWithIndexPath
    indexPath: NSIndexPath) {
    let row = indexPath.row; // ...
}
```

Data in Your iPhone App

Data in Your iPhone Application

- Property Lists, NSUserDefaults and Settings
  - Quick & easy, but limited
- iOS'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
NSPropertyListSerialization.propertyListWithData:options:format:

// NSData to property list
```

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
writeToFile:atomically:
```

- 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
initWithContensOfFile:
```

Writing an Array to Disk

```
let path = NSBundle.mainBundle().pathForResource("Data", ofType: "plist")
let array= NSArray(contentsOfFile: path!)
data = array! as! [String]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<array>
    <string>Foo</string>
    <integer>10</integer>
    <real>1.0</real>
    <true/>
    <date>2010-02-02T09:26:18Z</date>
</array>
</plist>
```

Writing a Dictionary to Disk

```
let path = NSBundle.mainBundle().pathForResource("Data", ofType: "plist")
let dict = NSDictionary(contentsOfFile: path!)
data = dict!.objectForKey("item") as! [String]
```

```
<plist version="1.0">
<dict>
    <key>item</key>
    <array>
        <string>Apple</string>
        <string>Banana</string>
    </array>
</dict>
</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 exist create and enumerate directories; move, copy, delete files

```
let rootPath =
NSSearchPathForDirectoriesInDomains(NSSearchPathDirectory.DocumentDirectory, .UserDomainMask, true)[0]
var plistPath = rootPath + "/test.plist"
if !NSFileManager.defaultManager().fileExistsAtPath(plistPath) {
   let plistBundle = NSBundle.mainBundle().pathForResource("test", ofType: "plist") as String!
   do {
      try NSFileManager.defaultManager().copyItemAtPath(plistBundle, toPath: plistPath)
   } catch {
print("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)

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
init(coder aCoder: NSCoder) {
    super.init(aCoder) // must call super's version
     // encode code...
}
// read: decode an object from an archive
init(coder aDecoder: NSCoder) {
    super.init(aDecoder)
    // decode code...
    // note that order does not matter
}
```

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

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 (JSON)

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

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
- Writing a JSON string from Foundation objects

```
// create some data in your app
var jsonObject : [AnyObject] = [
   ["name": "tom", "age": 2],
   ["name": "jerry", "age": 5] ]
// convert into a JSON string before sending to the cloud
if NSJSONSerialization.isValidJSONObject(dict) {
    do {
        let data = try
NSJSONSerialization.dataWithJSONObject(jsonObject, options:
NSJSONWritingOptions.PrettyPrinted)
        if let string = NSString(data: data, encoding:
NSUTF8StringEncoding) {
            return string as String
        }
```

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

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

References

- Lecture 9(ScrollView) & 10(TableView) Slide from Developing iOS8 Apps with Swift (Winter 2015) @Stanford University
- http://makeapppie.com/2014/10/28/swift-swift-using-uiwebviews-in-swift/
- http://sourcefreeze.com/category/swift/
- http://ste.vn/2015/06/10/configuring-app-transport-security-ios-9-osx-10-11/