ITP 342 Mobile App Development

Data Persistence

Persistent Storage

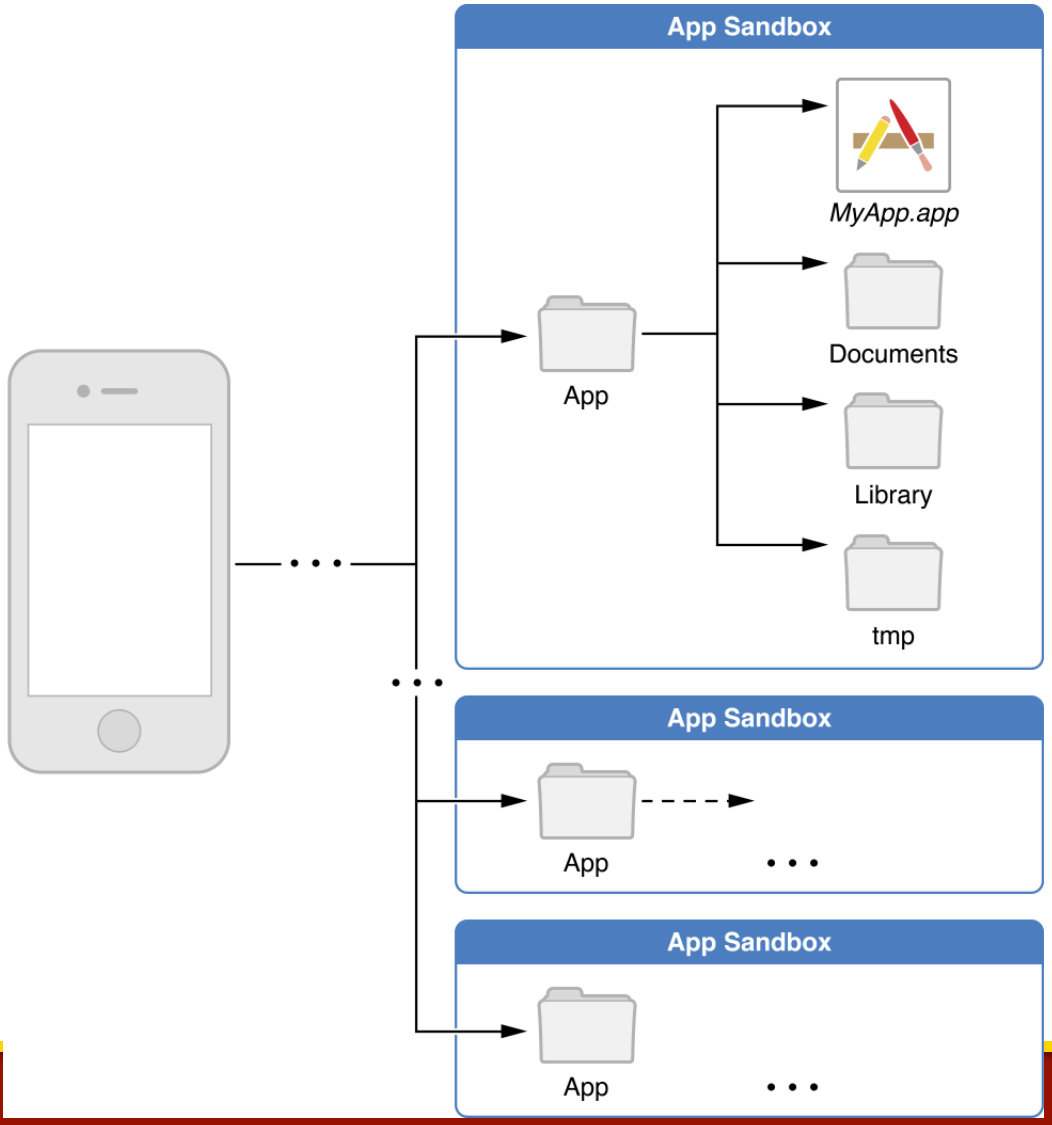
- Want our app to save its data to persistent storage
  - Any form of nonvolatile storage that survives a restart of the device
- Want a user to make changes & have the changes stored and there when the user launches the app again
- A number of different mechanisms to persist data to the iOS's file system
  - User Defaults
  - Property lists
  - Object archives (or archiving)
  - SQLite3 (iOS's embedded relational database)
  - Core Data (Apple's provided persistence tool)

The App Sandbox

- For security reasons, iOS places each app (including its preferences and data) in a sandbox at install time
- A sandbox is a set of fine-grained controls that limit the app's access to files, preferences, network resources, hardware, and so on
- As part of the sandboxing process, the system installs each app in its own sandbox directory, which acts as the home for the app and its data

Sandbox Directories in iOS

- Subdirectories
  - Documents
  - Library
  - tmp



App Sandbox in Simulator

- Open a **Finder** window
- Press **command-shift-G** to use "Go to"
- Enter **~/Library/Developer/CoreSimulator/Devices**
  - ~ represents the user you are logged in as
  - ~ can be substituted for *harddrive*/Users/*username*
  - The Library folder is hidden
- Select one of the folders (8#-4#-4#-4#-12#) representing a device
- Select **data/Containers/Data/Application** folder
- Select one of the folders (8#-4#-4#-4#-12#) representing an app
- You will see folders for Documents, Library, and tmp
- This listing represents the simulator, but the file structure is similar to what's on the actual device

Application Folder

- The Application folder is where iOS stores its applications for that device
- Drill down and you will see a bunch of folders with names that are long strings of characters
  - These names are globally unique identifiers (GUIDs) and are generated automatically by Xcode
  - Each folder represents an app
- Click on one of them (any of them)

Your App's Folder

- Drill down into one of the application subfolders
  - **Documents**
    - Your applications stores its data in Documents, with the exception of NSUserDefaults-based preference settings
  - **Library**
    - NSUserDefaults-based preference settings are stored in the Library/Preferences folder
  - **tmp**
    - Place to store temporary files that will not be backed up by iTunes, but your app has to be responsibility for deleting the files

Settings App

- iPhone and other iOS devices have a dedicated application called Settings
- Goal:
  - Add settings for your applications to the Settings application
  - How to access those settings from your application

Settings Bundle

- A settings bundle is a groups of files built in to an application that tells the Settings application which preferences the application wishes to collect from the user
- The Settings app acts as a common user interface for the iOS User Defaults mechanism
  - User Defaults is the part of the system that stores and retrieves preferences
  - In iOS, User Defaults is implemented by the **NSUserDefaults** class
    - It's the same class used to store and read preferences on the Mac
    - Your apps will use NSUserDefaults to read and store preference data using a key value (like NSDictionary)
    - NSUserDefaults data is persisted to the file system

NSUserDefaults

- Simple dictionary-like API
- Save basic data to disk
  - NSString
  - NSNumber
  - NSDate
  - NSArray
  - NSDictionary (hashtables)

Saving to NSUserDefaults

```
// QuotesModel.m
#import "QuotesModel.h"

// Declare constants outside of @interface and @implementation
NSString *const kQuotesArrayKey = @"QuotesArray";

@implementation QuotesModel

- (void) save {
  NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

  [defaults setObject:self.quotes forKey:kQuotesArrayKey];

  [defaults synchronize];
}
```

Consts

- `const` is a pointer to the object in memory
  - It is more efficient than `#define`
    - `#define` is a pre-processor macro
- Why use constants at all?
  - You could type a constant's value in each and every place it's being used, but there is the human factor.
    - You could easily mistype the string and the compiler would not complain about your grammar.
  - This way is more convenient since Xcode tries to do his best to autocomplete stuff for us and these constants are no exception.

Consts

- Placement
  - Declare constants outside of `@interface` and `@implementation`
- Naming conventions
  - UpperCamelCase
  - May start with a two-letter upper-case prefix

```
NSString *const UDKeyQuotesArray = @"QuotesArray";
```

  - Other Objective-C code used a k prefix

```
NSString *const kUserDefaultsQuotesArrayKey = @"QuotesArrayKey";
```

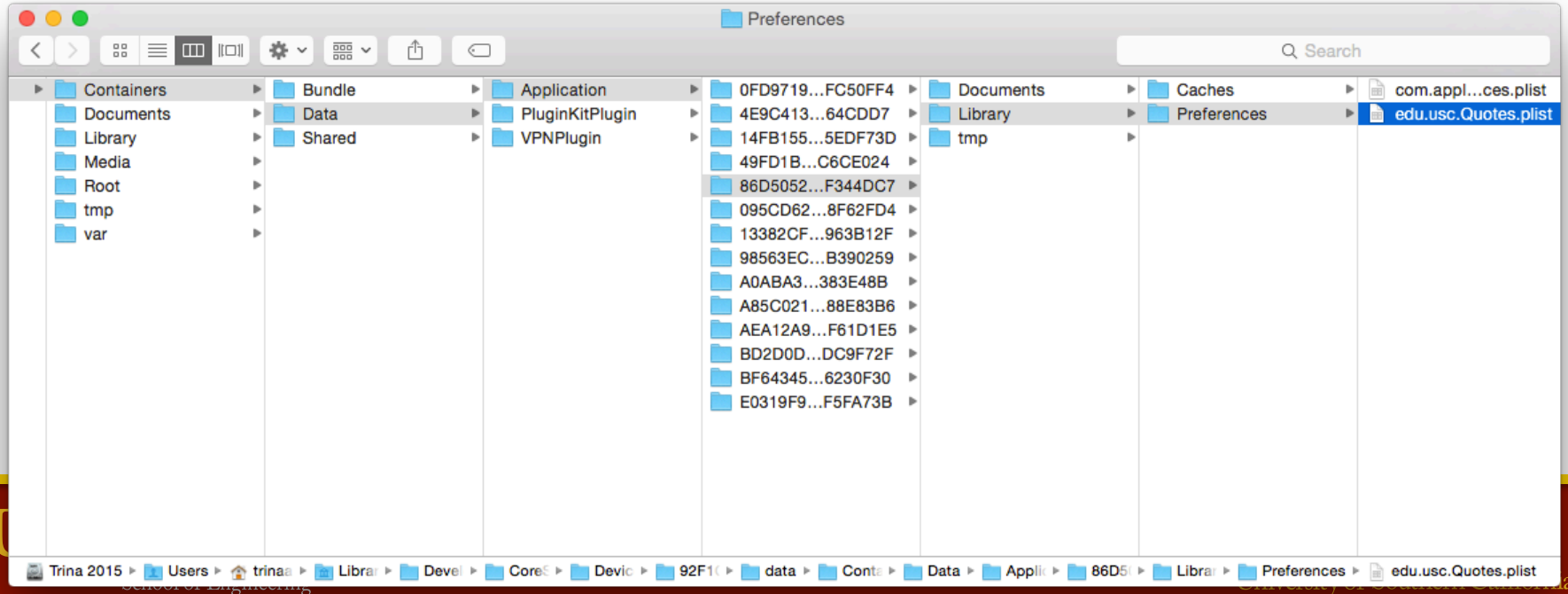
Persisting

- When do we need to save to the User Defaults (i.e. call our new `save` method)?

1. Add a new quote
2. Remove a quote

Find NSUserDefaults file

- Open a **Finder** window
- Press **command-shift-G** to use "Go to"
- Copy & paste the app path
- Click on Library → Preferences



Restoring

- Model still needs to read from user defaults
- User Defaults collections are always immutable
  - We need to create our own mutable copy
- Need to update the `quotes` property
  - Do in the init method or
  - Do a lazy load (override the get method)
    - Get it the first time it is needed

Overriding the Get Method

```
// QuotesModel.m

- (NSMutableArray *) quotes {
  if (!_quotes) {
    NSArray *storedQuotes = [[NSUserDefaults standardUserDefaults]
      objectForKey: kQuotesArrayKey];

    if (storedQuotes) {
      _quotes = [storedQuotes mutableCopy];
    } else {
      NSDictionary *quote1 = [[NSDictionary alloc]
        initWithObjectsAndKeys: @"Quote 1", kQuoteKey,
        @"Author 1", kAuthorKey, nil];
      NSDictionary *quote2 = [[NSDictionary alloc]
        initWithObjectsAndKeys: @"Quote 2", kQuoteKey,
        @"Author 2", kAuthorKey, nil];
      _quotes = [[NSMutableArray alloc]
        initWithObjects: quote1, quote2 nil];
    }
  }
  return _quotes;
}
```

Update the init Method

```
// QuotesModel.m

- (instancetype) init {
  self = [super init];
  if (self) {
    NSArray *storedQuotes = [[NSUserDefaults standardUserDefaults]
      objectForKey: kQuotesArrayKey];

    if (storedQuotes) {
      _quotes = [storedQuotes mutableCopy];
    } else {
      NSDictionary *quote1 = [[NSDictionary alloc]
        initWithObjectsAndKeys: @"Quote 1", kQuoteKey,
        @"Author 1", kAuthorKey, nil];
      NSDictionary *quote2 = [[NSDictionary alloc]
        initWithObjectsAndKeys: @"Quote 2", kQuoteKey,
        @"Author 2", kAuthorKey, nil];
      _quotes = [[NSMutableArray alloc]
        initWithObjects: quote1, quote2 nil];
    }
    // set up other instance variables
  }
  return self;
}
```

NSUserDefault Limitations

- No high performance
  - Meant for small amounts of data
- No complex structure or relationship support
- Not searchable

Save Files in Documents

- Instead of using User Defaults, let's put data into the Documents folder of our app (**Lab5**)
- You can write/read various types of files
  - Property Lists (plists)
    - Standard way to store text and settings since the early days of Cocoa
    - Data can be either in XML or binary format
  - Text Files
  - Archiving Objects
    - Save data objects as binary data through a process of encoding/decoding objects
    - The encoding operation supports scalar types and objects that support the NSCoding protocol

Getting the Documents Folder

- Since our app is in a folder with a seemingly random name, how do we retrieve the full path to the Documents folder so we can read and write files?
  - Use the C function **`NSSearchPathForDirectoriesInDomain()`**
    - It's a Foundation function, also used by Cocoa for Mac OS X

Documents Folder

- The contant **`NSDocumentDirectory`** says we are looking for the path to the Documents directory
- The constant **`NSUserDomainMask`** indicates that we want to restrict our search to our app's sandbox
- It gives us an array of matching paths, but we know we are only going to get one which is at index 0

```
NSArray *paths = NSSearchPathForDirectoriesInDomains
  (NSDocumentDirectory, NSUserDomainMask, YES);

NSString *documentsDirectory = [paths objectAtIndex:0];

NSString *filepath = [documentsDirectory
  stringByAppendingPathComponent:@"theFile.plist"];
```

Getting the tmp Folder

- Use the Foundation function called `NSTemporaryDirectory()`

```
NSString *tempPath = NSTemporaryDirectory();

NSString *tempFile = [tempPath
  stringByAppendingPathComponent:@"tempFile.txt"];
```

Single File Persistence

- Using a single file is the easiest approach
- With many apps, it is a perfectly acceptable one
- You start off by creating a root object, usually an NSArray or NSDictionary
  - It could also be based on a custom class
- Whenever you need to save, your code rewrites the entire contents of the root object to a single file
- When your app launches, it reads the entire contents of that file into memory; when it quits, it writes out the entire contents
- The downside of using a single file is that you need to load of all of your app's data into memory
  - And you must write all of it to the file system for event the smallest changes

Multiple File Persistence

- It allows the app to load only data that the user has requested (another form of lazy loading)
- When the user makes a change, only the files that changed need to be saved
- This method also gives you the opportunity to free up memory when you receive a low-memory notification
- The downside is that it adds a fair amount of complexity

Using Property Lists

- Property lists are convenient and can be edited manually using Xcode or the Property List Editor app
- Both **NSDictionary** and **NSArray** instances can be written to and created from property lists, as long as the dictionary or array contains only specific serializable object
- **Serialized object** - one that has been converted into a stream of bytes so it can be stored in a file or transferred over a network

Property List Serialization

- Only certain objects can be placed into a collection class, such as an NSDictionary or NSArray, and then stored to a property list using the collection class's `writeToFile:atomically` method
- The following Objective-C classes can:
  - NSArray, NSMutableArray
  - NSDictionary, NSMutableDictionary
  - NSData, NSMutableData
  - NSString, NSMutableString
  - NSNumber
  - NSDate

Write To File

- The atomically parameter tells the method to write the data to an auxiliary file, not to the specified location
- Once it has successfully written the file, it will then copy the auxiliary file to the location specified by the first parameter
- This is a safer way to write a file:
  - If the app crashes during the save, the existing file (if there was one) will not be corrupted
- It adds a bit of overhead, but in most situations, it's worth the cost

```
[myArray writeToFile:filePath atomically:YES];
```

```
[myDictionary writeToFile:filePath atomically:YES];
```

Persistent Model for Quotes

- We could change the code in our **QuotesModel** to use a plist file
- Call it something like **quotes.plist**

Saving Quotes

- Create constants
  - Not required, but easier to work with

```
// QuotesModel.h

// Keys for dictionary
static NSString *const kQuoteKey = @"quote";
static NSString *const kAuthorKey = @"author";
```

```
// QuotesModel.m

// Filename for data – quotes plist
static NSString *const kQuotesPList = @"Quotes.plist";
```

Saving Quotes

- Create a property of type `NSString` for the file path
- The model has a property of type NSMutableArray to hold the quotes
  - Use that property to save to a file

```
// QuotesModel.m

// Add property in appropriate section
@property (strong, nonatomic) NSString *filepath;

// Add method in appropriate section
- (void) save {
    [self.quotes writeToFile:self.filepath atomically:YES];
}
```

Saving Quotes

- Initialize filepath & quotes in the `init` method
- Do not use the lazy load methods for quotes

Saving Quotes

```
// QuotesModel.m

- (id) init {
  self = [super init];
  if (self) {
    NSArray *paths = NSSearchPathForDirectoriesInDomains (
      NSDocumentDirectory, NSUserDomainMask, YES);

    NSString *documentsDirectory = [paths objectAtIndex:0];

    _filepath = [documentsDirectory stringByAppendingPathComponent:
      kQuotesPlist];

    _quotes = [NSMutableArray arrayWithContentsOfFile:_filepath];

    if (!_quotes) {  // no file
      // create quotes array
    }
  }
  return self;
}
```

Saving Quotes

- When do we need to save to the plist file (i.e. call our new `save` method)?

1. Add a new quote
   - Update the `insertQuote:` methods
2. Remove a quote
   - Update the `removeQuoteAtIndex:` method

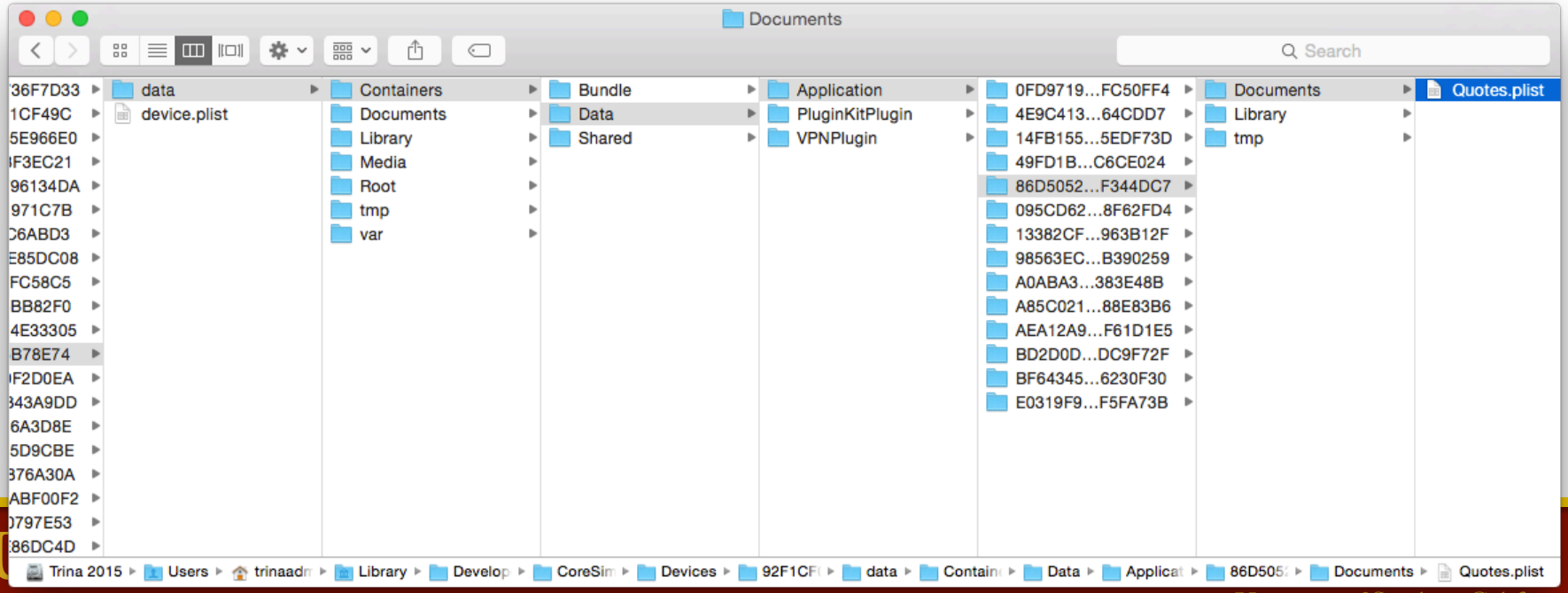
iOS Simulator

- We want to find the crazy folder to make sure the plist file is create.
- Print out the directory path in the console.
- Add the following code to the viewDidLoad method in the QuotesViewController.

```
#if TARGET_IPHONE_SIMULATOR
  NSLog(@"Documents Directory: %@", [[[NSFileManager defaultManager]
    URLsForDirectory:NSDocumentDirectory
    inDomains:NSUserDomainMask] lastObject]);
#endif
```

Find .plist file

- Open a **Finder** window
- Press **command-shift-G** to use "Go to"
- Copy & paste the Documents path
- Verify that your Quotes.plist file is there



Embedded SQL

- SQLite3 is very efficient at storing and retrieving large amounts of data
- It's also capable of doing complex aggregations of our data, with much faster results than you would get doing the same thing using objects
- SQLite3 uses the Structured Query Language (SQL)
  - It's the standard for interacting with relational databases
  - http://www.sqlite.org/cintro.html
  - http://www.sqlite.org/lang.html

Creating or Opening the DB

- Use the sqlite3_open() function to open an existing database
  - If none exists at the specified location, it will create a new one
  - If result is equal to the constant SQLITE_OK, then the database was successfully opened
  - The path to the database file must be passed in as a C string, not as an NSString
  - SQLite3 was written in portable C, not Objective-C

```
sqlite3 *database;
int result = sqlite3_open("/path/to/database/file", &database);
```

Core Data

- Converts objects to and from a persistent format
  - SQLite on iOS
- Provides modeling tools to capture the classes
- Allows you to edit objects and persist later
  - Makes Undo easy
  - Keeps local changes local
- Will have a separate presentation on this