Advanced iOS

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 22 — 11/03/2011

Goals of the Lecture

- Present a few additional topics and concepts related to iOS programming
  - persistence
    - serialization
  - advanced view controllers
    - navigation, image picker
  - But First: Objective-C 2.0 Categories and Protocols

Objective-C Categories (I)

- Have you ever been in a situation where you're using a class provided by a library and you say
  - "I wish this class had a method that did …"
- In most languages, if you want to add a method to an existing class, say java.lang.String, you would need to create a subclass: "class MyString extends String"
  - Warning: Abandon All Hope, Ye Who Enter Here!
  - This approach is fraught with peril
- In Objective-C, you don't have to subclass: just use a category

Objective-C Categories (II)

- Objective-C Categories let you re-open a class definition and add a new method!
  - The original class will then act as if it had that method all along!
- Your new method is often implemented using just the publicly available methods of the original class and so you don't require any special knowledge of the original class to add the new method

Objective-C Categories (IV)

- To create a category, you use the following syntax

```
@interface ExistingClass (NameOfCategory)
      <method signatures>
@end

@implementation ExistingClass (NameOfCategory)
      <method defs>
@end
```

Objective-C Categories (V)

- We've seen this before with class extensions; a class extension uses the same syntax but doesn't provide a name for the category (we'll see examples of class extensions in today's example app)

- Here's an example of extending the built-in NSArray class

```
@interface NSArray (NestedArrays)
- (NSInteger) countOfNestedArray:(NSUInteger)pos;
@end
```

Objective-C Categories (VI)

- Example of extending built-in NSArray class

```
@implementation NSArray (NestedArrays)

- (NSInteger) countOfNestedArray:(NSUInteger)pos {
    NSArray *subArray = [self objectAtIndex:section];
    return [subArray count];
}
@end
```

Objective-C Categories (VII)

- Now, you simply include the category in new code and NSArray will act as if it always had the method **countOfNestedArray:** (!!!)

```
#import "NSArray-NestedArrays.h"

NSArray *foo = <code to get an array>

NSInteger subarray_count =

    [foo countOfNestedArray:2];

NSLog(@“%d items in subarray”, subarray_count);
```

Protocols (I)

- Protocols are Objective-C's version of Java's Interfaces
  - They allow you to define a type that is guaranteed to implement a particular set of methods
    - A class can be declared as "conforming" to a particular protocol
      - You can then refer to all objects that conform to a protocol in a uniform manner
- Protocols are typically used to define the interface of a delegate

Protocols (II)

- To define a protocol, you use the following syntax

```
@protocol ProtocolName
    <method signatures>
@end
```

- To conform to a protocol, you use the following syntax

```
@interface MyClass <ProtocolName1, ProtocolName2>
    ...
@end
```

The compiler will then make sure that you implement the methods of the protocol

Protocols (III)

- To declare a variable or parameter to only accept instances of a certain protocol, you use the syntax

```
id <ProtocolName> foo = objectThatConformsToProtocolName;
```

- You'll see examples of this in today's sample code
  - In particular, ProfilesViewController conforms (or implements) the UITableViewDataSource and UITableViewDelegate protocols and ProfileViewController conforms to the UIImagePickerControllerDelegate protocol

Example Application

- Profile Viewer
  - Similar to what we saw on the Android side of the fence
- Will develop complete application step-by-step
  - taking care of the data model via serialization first
  - adding view controllers one-by-one
  - then updating the data model to use Core Data

iOS 5

- Will also feature a few aspects of iOS 5 in that
  - I'll be using an iOS 5 project template
  - I'll be taking advantage of automatic reference counting

Automatic Reference Counting (I)

- In previous lectures, I introduced you to Objective-C 2.0 memory management
  - alloc, init, retain, release, autorelease
- and discussed the various patterns that need to be followed to ensure that no memory leaks occur in your iOS programs
- With automatic reference counting (ARC) all of that goes away!

Automatic Reference Counting (II)

- Well, sort of.
- All of it still happens BUT the compiler does it for you!
- There are some "corner cases" that advanced developers need to be aware of
  - but for the most part, you delete all of your retain-release-autorelease related code
- Indeed, invoking those functions when ARC is turned on results in a compiler error!

Automatic Reference Counting (III)

- For the curious, the "corner cases" involve
  - retain cycles: "A retains B and B retains A"
  - use of Core Foundation Objects
  - and certain uses of blocks
- I won't discuss the latter two but with the former the workaround is that developers must either
  - add code to explicitly break the cycle or annotate one of the pointers with the qualifier "weak"

Automatic Reference Counting (IV)

- For more information, see Mike Ash's post for details
  - http://www.mikeash.com/pyblog/friday-qa-2011-09-30-automatic-reference-counting.html
- Mike Ash's website is an excellent resource for information on advanced Mac OS X and iOS programming topics

Let's Get Started

- Bring up the New Project dialog in XCode 4.2
  - Select the "Empty Application" template under the iOS Application category
  - Configure the project to be for the iPhone only
  - Name the project ProfileViewer
  - Turn on ARC
  - Indicate a location to store the project on disk

Initial Clean-Up (I)

- The "Empty Application" template is very straightforward
  - It contains a single AppDelegate class which gets instantiated by the single main.m class
  - There is no default .xib file as we've seen in other project templates
- All we need to do at first is clean-up AppDelegate.m by deleting all of the code in it except for
  - application:didFinishLaunchingWithOptions:

Initial Clean-Up (II)

- Note the presence of ARC in AppDelegate.h
- The property that maintains a pointer to the application window is defined as follows
  - @property (strong, nonatomic) UIWindow *window;
- The "strong" keyword tells ARC that we want to retain any instance passed to this property and keep it around.
  - If we ever switch to a new window (which we won't) ARC would make sure to automatically release the previous instance

Next the Model

- We need a Profile class to store
  - First Name
  - Last Name
  - e-mail address
  - (and eventually a photo)
- We need a Profiles class to store the instances of Profile created at run-time

Profile

- Use the New File Dialog to add Profile.h and Profile.m to the Project.
  - Select the Objective-C class from the iOS Cocoa Touch category
  - Name the class "Profile" and make sure its a subclass of NSObject
- This provides you with a skeleton class definition

Profile Properties

- Add the following properties in Profile.h
  - @property (nonatomic, strong) NSString* firstName;
  - @property (nonatomic, strong) NSString* lastName;
  - @property (nonatomic, strong) NSString* email;
- Now synthesize these properties in Profile.m
  - e.g. @synthesize firstName = _firstName;

Create the designated initializer

- - (id) initWithFirstName:(NSString*)first
  LastName:(NSString*)last
  Email:(NSString*)email;
  - See example code for details
- Then override the inherited initializer to call the designated initializer
  - - (id) init;
  - See example code for details; Note, we'll never actually call this method but it's good form to define it

Create a randomProfile method

- This method is a class method that can be used to generate a random profile
- + (id) randomProfile
- The "+" indicates that this is not an instance method but a class method that can be invoked on Profile directly
  - See example code for details

Additional methods

```
- (NSString*) fullName {
    return [NSString stringWithFormat:@"%@ %@",
                self.firstName, self.lastName];
}
- (NSString*) description {
    return [NSString stringWithFormat:@"%@ <%@>",
                [self fullName], self.email];
}
```

Demo (I)

- We now have enough code in Profile to test it
  - We will
    - temporarily import Profile into our AppDelegate
    - modify the didFinishLaunchingWithOptions: method
    - to create and print out some random Profile objects

Demo (II)

```
for (int i = 0; i < 10; i++) {

    NSLog(@"%@", [Profile randomProfile]);

}
```

- Believe it or not, the above code tests all of the methods of our Profile class, save one

Profiles (I)

- Now, we need to add a class to store a collection of Profile objects
  - These two classes will serve as the initial model for our Profile Viewer application
- Use the New File menu command to add a Profiles class to the project, just as you did for the Profile class

Profiles (II)

- We will provide a method to retrieve an immutable array of Profiles and to create a new random profile
- #import <Foundation/Foundation.h>
- @class Profile; // Note simply declares that Profile is a class
- @interface Profiles : NSObject
- - (NSArray*) allProfiles;
- - (Profile*) createProfile;
- @end

Profiles (III)

- We will store the profiles in a mutable array that is declared as a private property in a class extension that appears in the .m file
  - See example code for details
- We can quickly modify our test code to verify that this provides enough functionality to keep track of a set of profiles
  - See example code for details

Initial User Interface

- We will first create a view controller that displays profiles in a table view
  - This class will be called ProfilesViewController
    - It will be a subclass of UITableViewController
    - BUT when we create it, we will tell XCode that it is a subclass of NSObject
      - The NSObject subclass template is clean
        - We will create our table view "bottom up"

The steps

- Create a new NSObject subclass ProfilesViewController
  - Change its .h file by replacing NSObject with UITableViewController
  - Set up its initializers
  - Change AppDelegate to create an instance and set it as the rootViewController
    - Delete all Profile and Profiles-related code
  - Run to see empty table view

Connect to Data Source

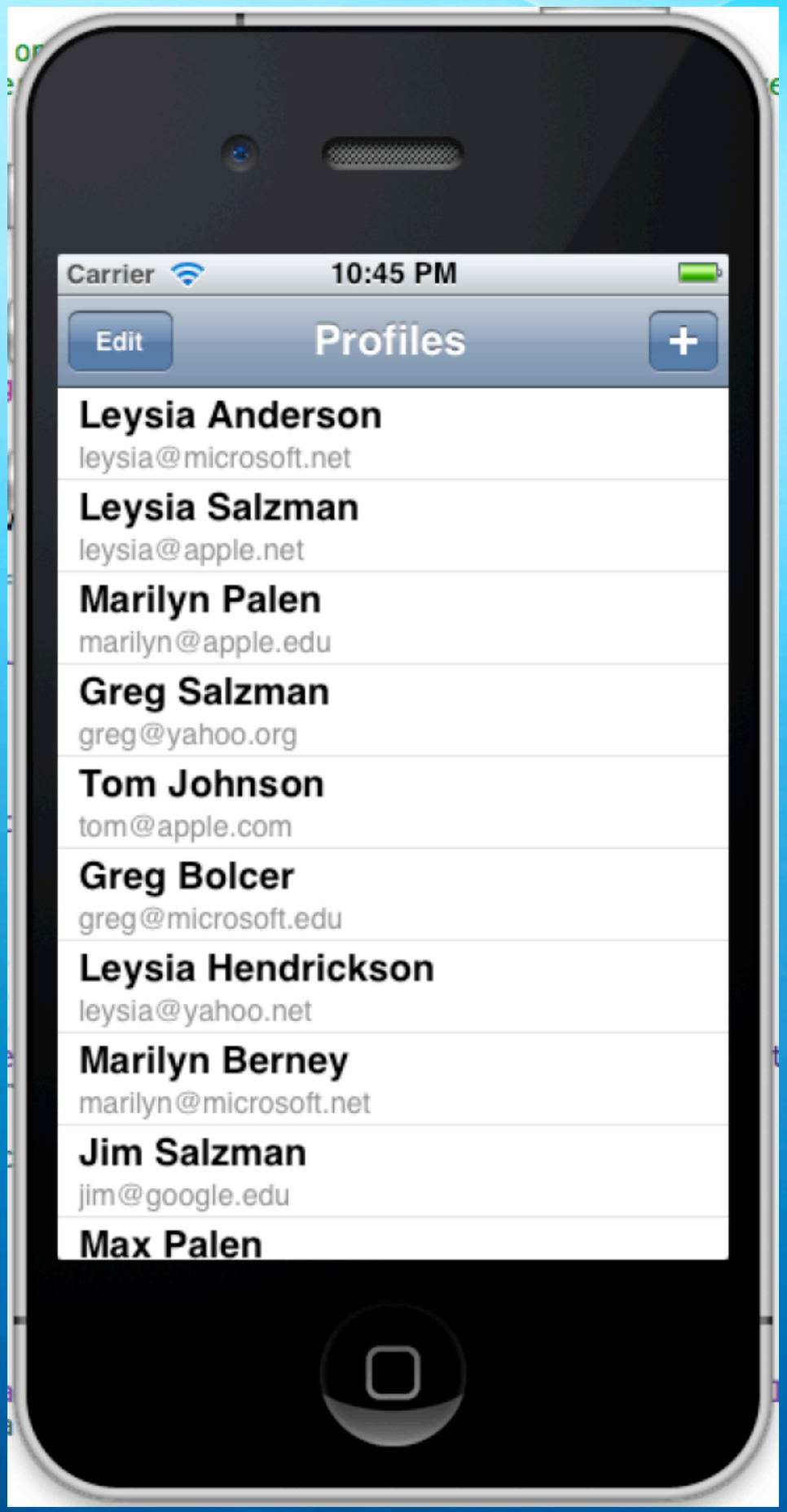
- Now we need to connect ProfilesViewController to its data source: the Profiles object created earlier
  - We will instantiate Profiles in ProfilesViewController's init method and add 20 random profiles to it
- We will also implement the core UITableViewDataSource methods by calling Profiles as needed
  - See example code for details

Ready for Navigation (I)

- We are going to lay the ground work for being able to edit profiles by adding ProfilesViewController to a UINavigationController
- This will provide us with a navigation bar that will allow us to add a title, and two buttons: "edit" and "new"
  - Let's get this all configured; we'll then implement the event handlers for our two buttons

Ready for Navigation (II)

- To accomplish this, we must
  - Create a navigation controller in AppDelegate.m
  - Set ProfilesViewController as its rootViewController
  - Set the navigation controller as the rootViewController of the window
  - Modify ProfilesViewController to configure its navigation item with a title and the two buttons
  - See example code for details



The previous steps results in an app that looks like this

The UINavigationController provides the navigation bar at the top and space for the ProfilesViewController

By configuring its navigationItem, ProfilesViewController populates the navigation bar with a title and two buttons

The edit button is functional in that it puts the table into or out of edit mode; this is functionality built right into the navigation controller and the table view controller

The plus button is not yet functional

Activate the Plus Button

- To make the "add profile" button functional we will
  - implement the handleAdd: event handler
    - - (void) handleAdd:(id) sender {
      - [self.profiles createProfile];
      - [self.tableView reloadData];
      - }
  - we will remove the code that creates 20 random profiles in the initializer

Handle Editing

- In order to handle "edit mode", we need to
  - be able to delete profiles
  - be able to move them around
- Showed similar code earlier in the semester
  - As a result, I won't go into detail on this step
  - Changes are made to Profiles and to ProfilesViewController

Detail View (I)

- Now we need a way to edit the individual values of the Profile
- We will create a new view controller called ProfileViewController.
  - It will display each of the three values of a Profile and let us edit their values
- To get started, create a new file using the UIViewController subclass template. Make sure to request that a .xib file be created. Delete all the code in the .m file.

Detail View (II)

- Drag three labels and three text fields to the user interface in the .xib file
  - Call the fields First Name, Last Name, and Email.
- Bring up the assistant editor and control drag from the text fields to the ProfileViewContoller.h file to autogenerate the property definitions
- Configure the viewDidLoad method to set the background to the same color as a UITableView
  - this will provide some continuity between views

Detail View (III)

- Next, edit ProfilesViewController's init method to use a grouped table
  - This ensures that the UITableView's background color is shown; it set's up the visual coherence between the two views
- Let's also add a disclosure triangle to each table cell to indicate that you click the row to see more details

Detail View (IV)

- Finally, we have to
  - detect when a row has been selected
  - get its associated Profile object
  - create an instance of ProfileViewController
  - configure it (we'll add a Profile property to do this)
  - push it onto the UIViewController
  - implement viewWillAppear and viewWillDisappear

Let's add images to profiles

- Design will involve
  - Adding a UIImageView to ProfileViewController
  - Adding a button to invoke UIImagePickerController
- Using UIImagePickerController to take/select a picture
- Add an image attribute to Profile to keep track of image data

Configuring ProfileViewController

- Add UIImageView and UIToolbar to ProfileViewController's xib file
- Ensure that the proper connections are made
  - We'll have an event handler named takePicture: and we'll have a property that points at the UIImageView
    - In takePicture:, we'll assign the UIImage object to the profile

Persistence (I)

- So far, our app can create and edit data
  - but that data does not persist between runs of the system
- We will first take advantage of Objective-C's serialization mechanism, which is known as Archiving

Persistence (II)

- To add support for archiving, we must change Profile so
  - it declares support for the NSCoding protocol
    - implements a method called encodeWithCoder:
      - this is called when saving an object
    - implements a method called initWithCoder:
      - this is called when loading an object
- We also must get a handle to a directory for our app

Persistence (III)

- Finally, we need to configure our app to save our changes
  - We will add calls to save changes at appropriate spots in ProfilesViewController
- Such as
  - after creating, deleting or editing a Profile

Coming Up Next

- Presentations due this Friday
- Homework 5 due on Monday
- Lecture 23: Commonality and Variability Analysis & The Analysis Matrix
  - Chapters 15 and 16
- Lecture 24: Decorator, Observer, Template Method
  - Chapters 17, 18 and 19