Advanced iOS

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 20 — 11/01/2012

Goals of the Lecture

- Present a few additional topics and concepts related to iOS programming
  - persistence
    - serialization
  - advanced view controllers
    - table view controller
  - custom views, touch events, application preferences, animations
- Plus introduce (more formally)
  - Objective-C Categories and Protocols
  - In the example code, the new Objective-C Literal syntax for arrays and dictionaries are also covered

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
```

- The interface goes in a file named `ExistingClass-NameOfCategory.h`
- The implementation goes in a file named `ExistingClass-NameOfCategory.h`

Objective-C Categories (V)

- Related to Categories is the notion of a **class extension**.
  - We've seen class extensions before in some of the example apps in the previous iOS lectures and we'll see examples in today's examples
- Class extensions appear in .m files and they allow you to declare methods and properties of the class that should be private to the class
  - Clients of such a class will not see the methods/properties of the extension because they make use only of the .h file, not the .m file
- A class extension looks like this

Note: no category name

```
@interface ExistingClass ()

        <method signatures and property declarations>

@end
```

Objective-C Categories (VI)

- Returning to categories, here's an example of extending the NSArray class

```
@interface NSArray (NestedArrays)

   - (NSInteger) countOfNestedArray:(NSUInteger)pos;

@end

@implementation NSArray (NestedArrays)

   - (NSInteger) countOfNestedArray:(NSUInteger)pos {

      NSArray *subArray = [self objectAtIndex:pos];

      return [subArray count];

   }

@end
```

Objective-C Categories (VII)

- Now, you simply include the category in new code
  - 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);
...
```

Demo

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 all of the required methods of the protocol; some methods can be optional

Protocols (III)

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

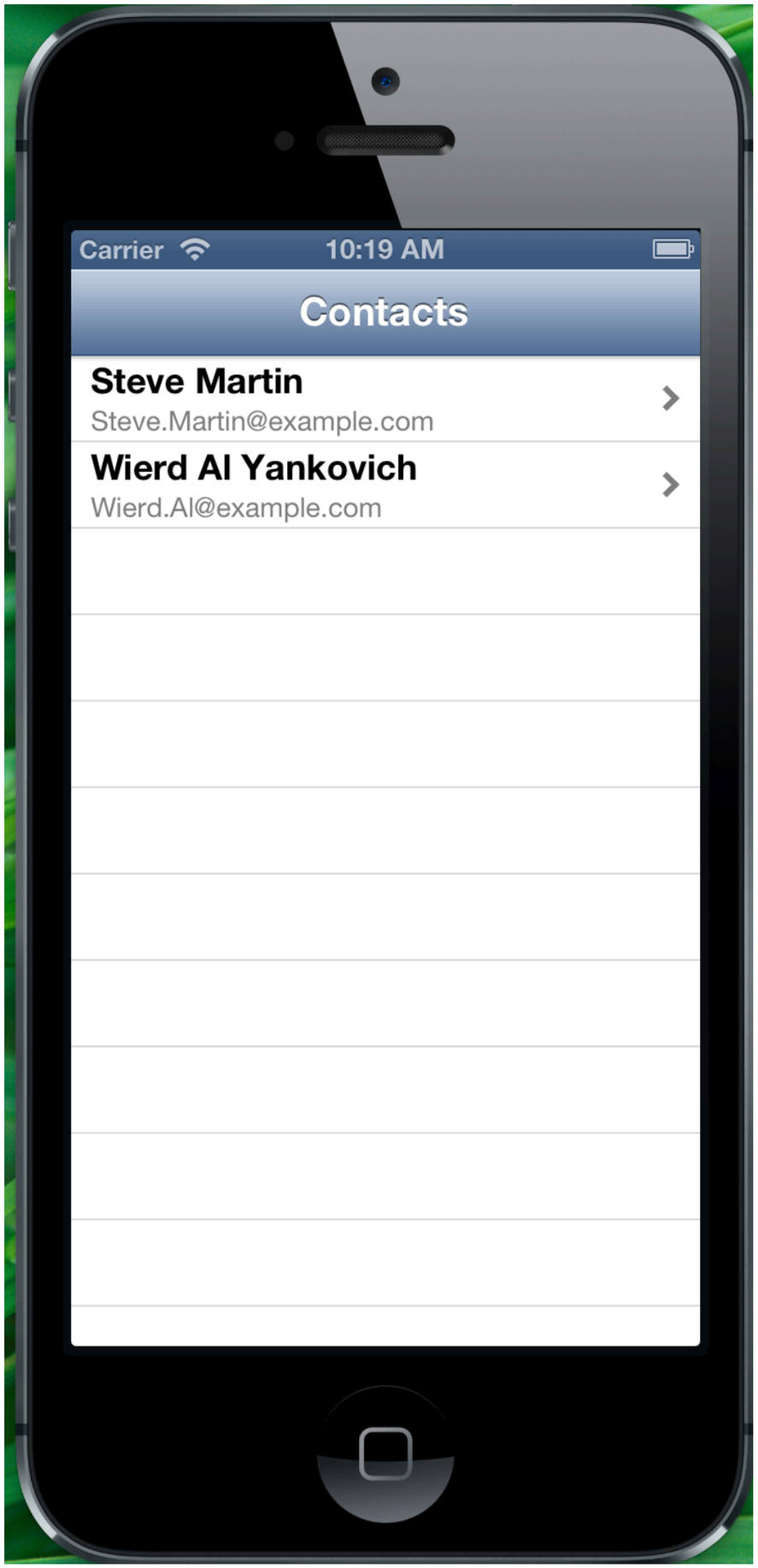
```
id <ProtocolName> foo = objectThatConformsToProtocolName;
```

- You'll see examples of classes conforming to protocols in today's examples
- For example
  - UITableViewDataSource
  - UITableViewDelegate

Returning to The Social

- The last part of Lecture 17 developed an application called The Social
  - It used a navigation controller, a table view controller, and our own custom view controller to allow the display and editing of “social media” contacts
    - See next slide
- We left the app in a state where it
  - auto-generated two contacts on start-up, and
  - allowed those contacts to be edited
- But
  - it was not able to create new contacts or delete existing contacts, and
  - changes to contacts were not persistent

The Social



Editing the Table (I)

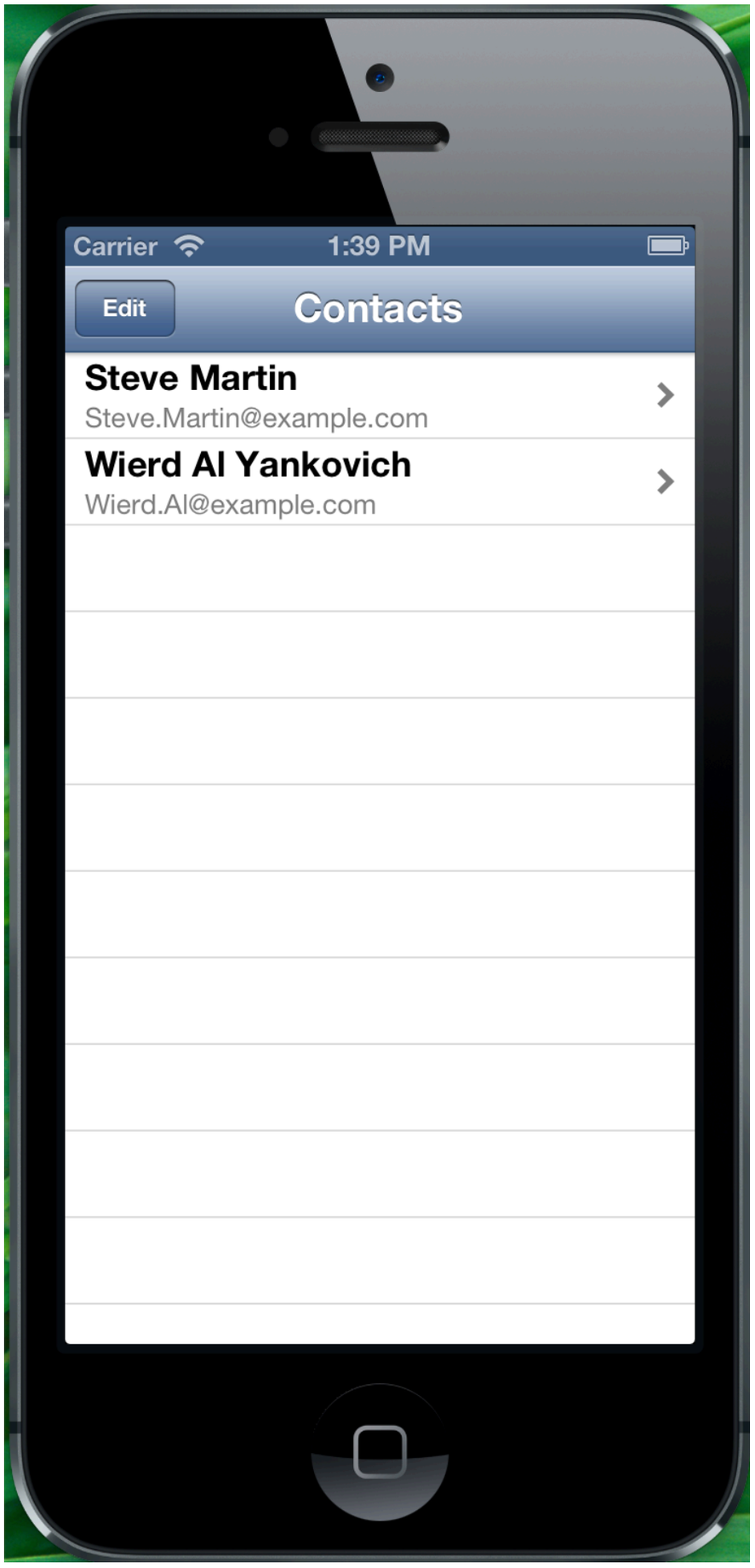
- Let's first add editing features to the the table view
    - We'll add an edit button to the navbar of the table view and let that button switch us in and out of "editing mode" for a table
        - A table in editing mode allows its rows to be rearranged or deleted
            - as long as we implement the appropriate methods of the UITableViewDelegate
                - (as I showed in an example earlier this semester)
- We'll then implement the required methods to allow the table to be edited

Editing the Table (II)

- Step One: Add the following line of code to viewDidLoad

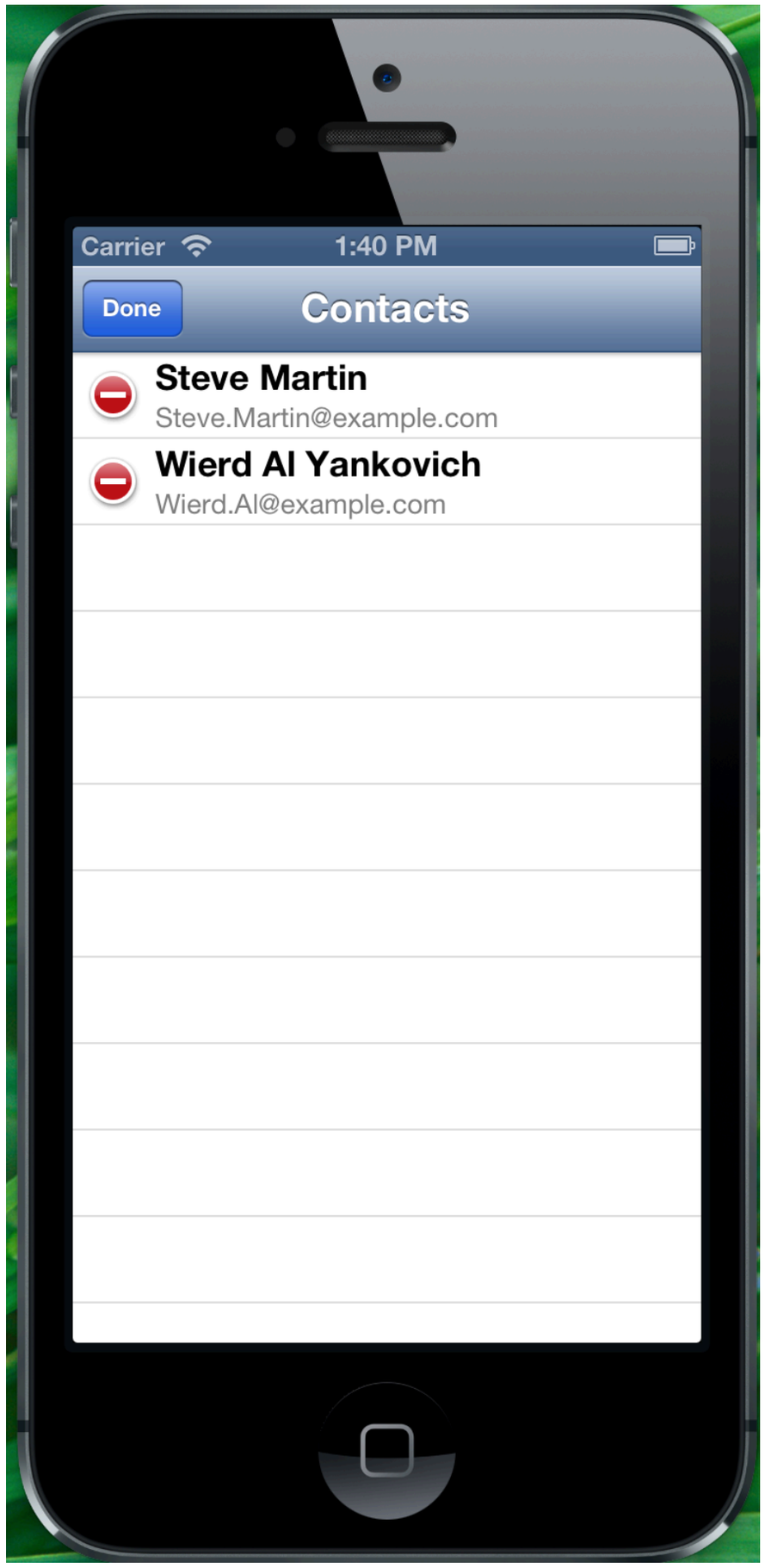
```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

- That's all it takes! What is this doing?
  - self is a reference to our ContactTableViewController
    - which is a subclass of UITableViewController
- A UITableViewController ensures that it's property navigationItem points to the navbar of our Navigation Controller
  - A navbar can have widgets (typically buttons) placed on it
- Finally, editing tables is so common, that UITableViewController maintains an editButtonItem that is pre-configured to "do the right thing"



Single line of code provides Edit button that puts table into “editing mode” when clicked

Changes to Done button automatically



Editing the Table (II)

- Step Two: Handle Delete
  - In order to handle deletion, we need to implement a method from the UITableViewDataSource protocol
  - Our class automatically “conforms” (or implements) this protocol because we are a subclass of UITableViewController and its definition is
    - @interface UITableViewController : UIViewController <UITableViewDelegate, UITableViewDataSource>
  - Since we’re a subclass, we inherit UITableViewController’s conformance to both of those protocols
- The method we need to implement is
  - `tableView:commitEditingStyle:forRowAtIndexPath:`

Demo

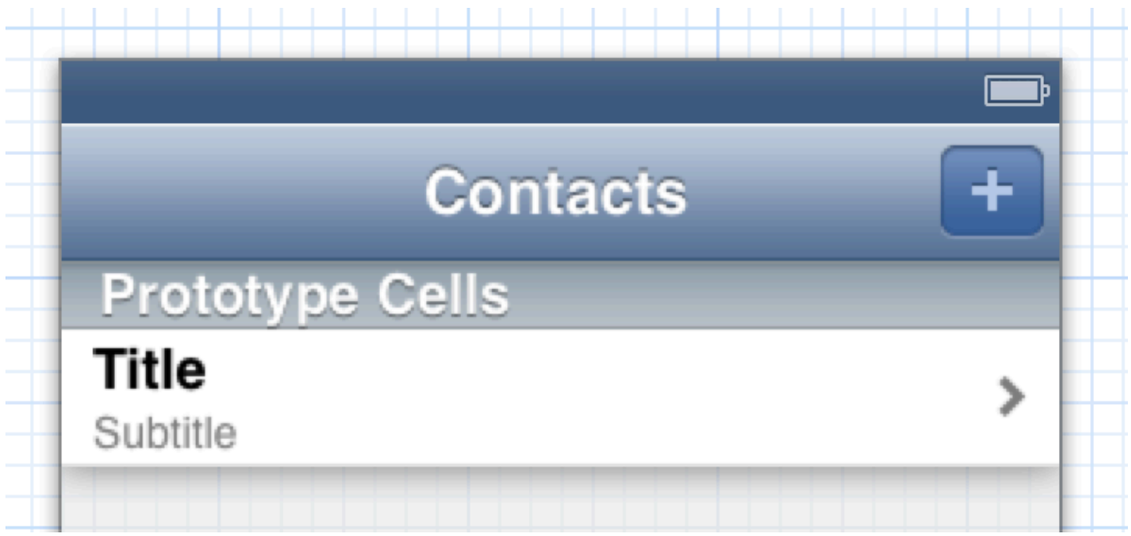
Editing the Table (III)

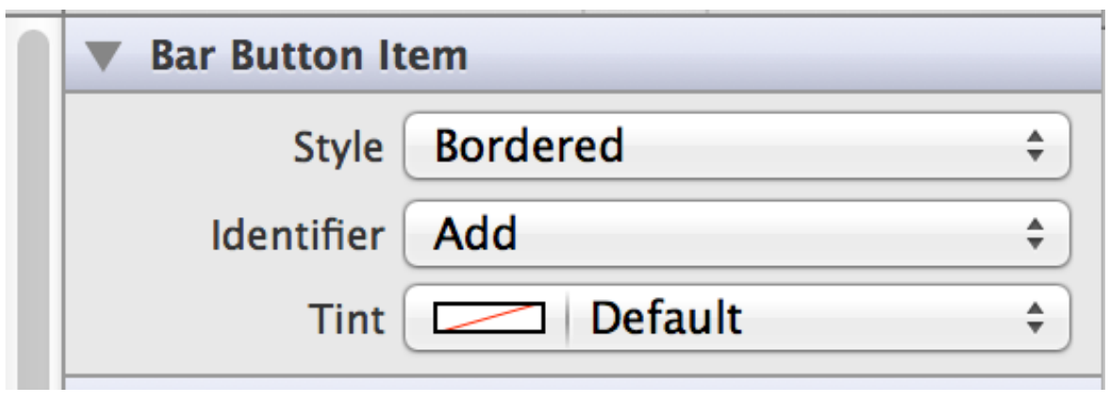
- Step Two: Handle Rearranging Rows
  - In order to handle the rearrangement of table rows, we need to implement a the following method from the UITableViewDataSource protocol
  - `- tableView:moveRowAtIndexPath:toIndexPath:`
- The code is straightforward
  - You get the source row from the index path and retrieve that element out of our collection of ContactInfo objects
  - You then remove that element from the collection
  - You then reinsert it back into the collection at the destination row
- As before, as soon as we implement this method, the movement handles for each row appears when we switch to edit mode

Demo

Adding a New Contact (I)

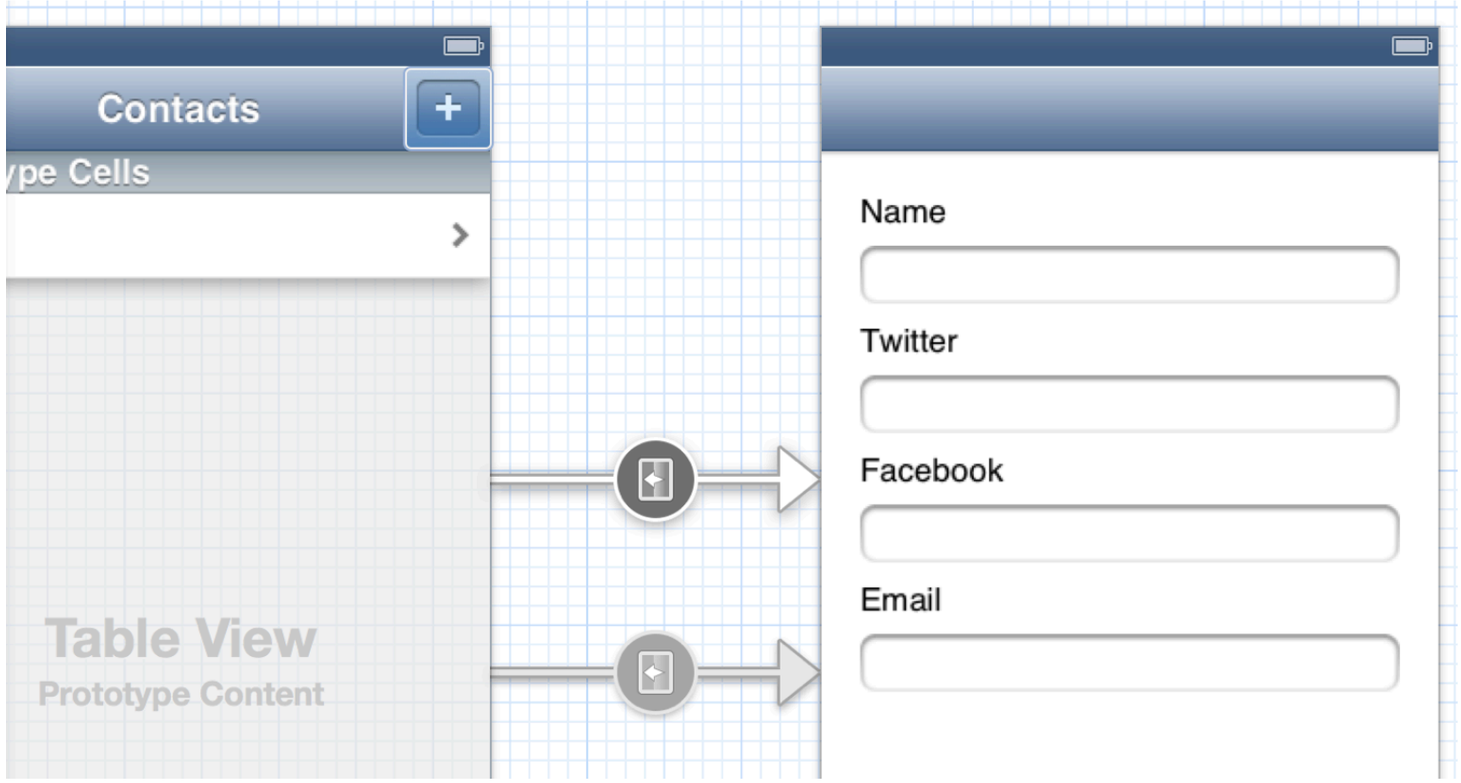
- To add a new contact, we are going to add another button to our navbar
  - We will make use of Interface Builder and our storyboard to handle this interaction
- First, drag a Bar Button Item to the right hand side of our Table View in Interface Builder
- In the attributes editor, make sure it's identifier is Add
- Our Table View now looks like this

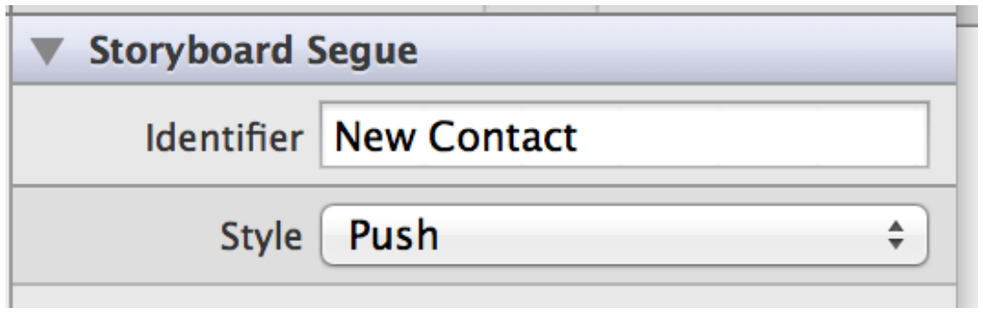




Adding a New Contact (II)

- We now select our Add button and control drag over to the ContactDetailViewController in the storyboard
  - It will create a second segue to this view controller
    - Select the Push transition and set the segue's identifier to be "New Contact"





Adding a New Contact (III)

- We modify our prepareForSegue:sender: method to invoke the ContactDetailViewController with a null ContactInfo property
  - This brings up our detail view with empty fields ready for a new profile
  - We will add code to this controller to detect when it has been launched with an empty profile and to create a new ContactInfo object to store any new data entered for that profile when the user is done
    - We will only create the new contact if we have a non-empty name
- We will also add code to this controller to pass any newly created ContactInfo object back to the Table View controller so that it can be added to the set of profiles to be displayed
  - To do this, we'll add a new method to our table view controller to receive new ContactInfo objects

Demo

Persistence (I)

- The last change we have to make involves making The Social's data persistent
  - To keep things simple, we are going to make use of Objective-C's serialization mechanism, known as Archiving
    - This mechanism is VERY similar to the java.io.Serialization mechanism that we saw in the Advanced Android lecture
- Our ContactInfo class must adopt the NSCoding protocol, this involves
  - declaring support for the NSCoding protocol in its .h file
  - adding a method called encodeWithCoder: to save an instance
  - add a method called initWithCoder: to load an instance

Persistence (II)

- We make use of NSMutableArray to store our ContactInfo instances
  - NSMutableArray implements the NSCoding protocol as well
  - So, just like we saw with our Android example, to save our data we
    - get a handle to our application directory
    - call “save” on the collection class
    - it will, in turn, call “save” on our ContactInfo objects
    - and the whole archive will be stored to disk
- The same is true for loading the data

Persistence (III)

- Just like we saw with Android, iOS provides routines to get access to a directory for our application to read/write files on the device
- Similar to Android, these application directories are private
- To get the directory, we use code like the following

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

    NSString *directory = [directories objectAtIndex:0];
```

- To then create a path for our file, we would do something like

```
    NSString *file = [directory
stringByAppendingPathComponent:“contacts.bin”]
```

- This file path can then be used to read/write the archive from/to disk

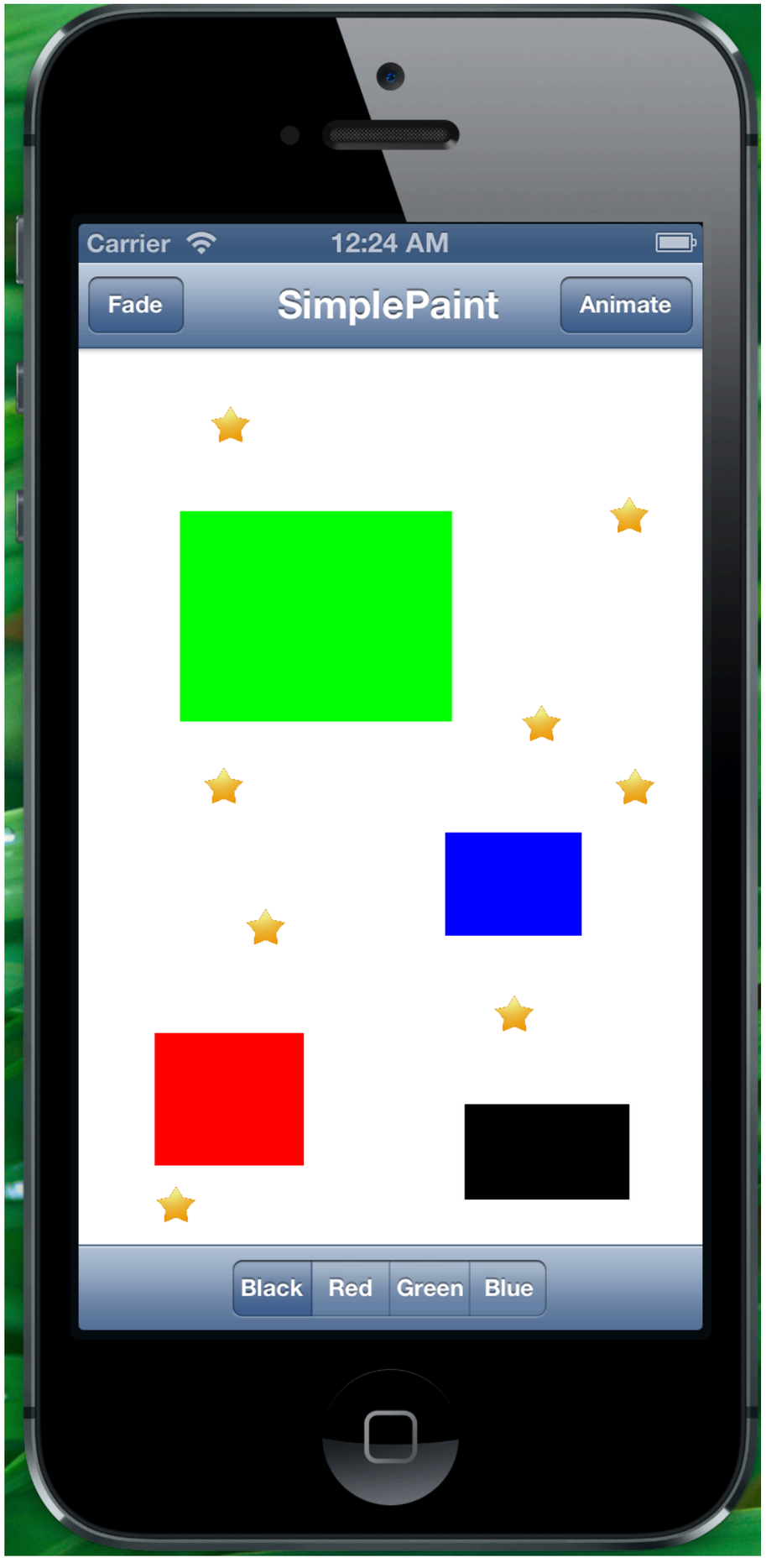
Persistence (IV)

- Note, the code on the previous slide feels more complex due to the way that OS X's file system is arranged
  - There are directory structures that get duplicated across multiple domains (individual users, the system's library, the system itself)
  - The parameters we pass to NSSearchPathForDirectoriesInDomains() guarantees that only a single directory is returned but it is possible for multiple directories to be returned, hence the need for the NSArray
- The last step is to then make sure
  - we load the file when our application is started, and
  - we write the file each time a contact is created, edited, or deleted

Demo

SimplePaint++

- To review the concepts of
  - custom views and 2D graphics
  - touch events
  - application preferences
  - animations
- in iOS, we will create an iOS version of the SimplePaint application that we saw in our Advanced Android lecture
  - I'm calling it SimplePaint++ because I'll also have it perform some simple animations of UIImageView objects in order to show how iOS handles animation



UI of SimplePaint++

Drag to create a rectangle with the selected color

Double tap to clear the display

“Long Press” to add a star

Click Fade to have all stars fade away

Click Animate to have all of the stars move around

Your color selection is remembered between runs of the application (via app preferences)

UIView

- In order to use 2D graphics in iOS, you need to be inside the drawRect: method of a UIView
  - Views in iOS know how to draw themselves
  - Views exist in a hierarchy
    - That means they have one parent view, and
    - they can contain multiple subviews
  - Views know how to handle touch events
- Unlike Android, rather than having an explicit Paint object to store the parameters that influence drawing, you ask for the current graphics context, manipulate it, and then do your drawing

Setting up the UI

- We'll once again use a storyboard and its default UIViewController with a default view
  - We'll create a subclass of UIViewController and a subclass of UIView and configure our storyboard to use each one of them
  - We'll use a UISegmentedControl to specify the drawing color
  - We'll use button bars to trigger animation (both translation and fading)

Drawing the Rectangles

- Drawing is handled similar to how we handled drawing in the Android version
    - We maintain a list of rectangles/colors pairs that have been created as well as a single “motionRect” that tracks the shape of a rectangle while it is created (drawn in the current color)
        - Our drawRect: routine
            - loops through the list of rectangles and draws them (using Quartz library routines)
            - draws the motionRect when it exists
- touch events keep track of the starting point of a touch (in the program we call this point the “anchor point”) and then updates motionRect appropriately and calls “setNeedsDisplay” to trigger a call to drawRect:

Animations (I)

- We make use of iOS's "block animations" to enable the type of simple animations that we saw over on the Android side of the fence
  - More complex animations can be created using Core Animation
- A UIView has the following properties that can be animated when changed
  - frame, bounds, center, transform, alpha, backgroundColor, contentStretch
- If a change is made to one of these properties outside of an animation block then the new value takes effect immediately
  - a change in the center point would cause the view to "jump" to its new location
- If a change is made to one of these properties inside an animation block then the change is animated (tweened) over a specified duration to its final value

Animations (II)

- Animation blocks are created using class methods on UIView
- Here's the code used in SimplePaint to fade a Star image view

```
[UIView
        animateWithDuration:1.0
        animations:^{view.alpha = 0.0;}
        completion:^(BOOL finished){
            [view removeFromSuperview];}];
```

- This method says "change the view's alpha value to zero over the course of one second; let me know when the animation is done, so I can remove the view from its parent"

Application Settings

- Preferences for iOS applications are handled via a class called NSUserDefaults
  - You retrieve a handle to your settings via a call to standardUserDefaults
  - You then treat the returned object like a mutable dictionary
    - You can store any value you want in it as long as it is serializable
    - You can then read that value back by supplying the key
- Keys are constant strings; an iOS convention for preference keys is
  - AppName + “Preference” + PrefKey
  - The preference name for SimplePaint is
    - SimplePaintColorPrefKey

Summary

- Seen quite a few topics on iOS programming
  - persistence/serialization
  - advanced view controllers
    - navigation controller and table view controller
    - custom views (2D graphics)
    - touch events, application preferences, animations
- Along with Objective-C Categories and Protocols

Missing Example

- Downloading a URL in iOS
  - I will try to create an example of that next week

Coming Up Next

- Homework 4 due on Monday
- Lecture 21: Reflections on Design
    - Chapters 12 and 13
- Lecture 22: Principles of Design Patterns
    - Chapters 14