Advanced iOS

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 20 — 03/17/2011

Goals of the Lecture

- Cover additional topics related to iOS programming
  - Additional Objective-C 2.0 techniques
  - Table and Navigation controller
  - Core Data
  - MapKit: MapView and Core Location
  - Retrieving Data from the Web

Credit Where Credit Is Due

- My examples are drawn from two books
  - The excellent "More iPhone 3 Development: Tackling iPhone SDK 3" by Dave Mark and Jeff LaMarche
    - Don't worry, the examples still work with iOS 4.x
  - "Beginning iOS 4 Application Development in Full Color" by Wei-Meng Lee
- Both are useful and highly recommended; I will not be providing extensive coverage of the examples, just highlights

More Objective-C 2.0 Techniques

- Properties
  - Update on best practice surrounding properties
- Categories
  - how to extend existing classes without subclassing them
- Protocols
  - Objective-C's version of Java's Interface

Properties

- Back in Lecture 13, I presented an example of specifying **properties** for Objective-C classes
    - properties were used to auto-generate getter and setter methods for publicly visible attributes
        - note: default visibility in Objective-C is protected, you can use @private and @public to change that default, if needed
- But the example I gave had some duplication in it (and developers hate duplication!)

Example

```
#import <Foundation/Foundation.h>
@interface Greeter : NSObject {
   NSString *greetingText;
}
@property (nonatomic, retain) NSString *greetingText;
@end
```

- The duplication lies with "NSString *greetingText"; why do we have to say this twice?

The Answer: We Don't!

```
#import <Foundation/Foundation.h>

@interface Greeter : NSObject {

}

@property (nonatomic, retain) NSString *greetingText;

@end
```

- Objective-C will auto-generate the attribute definition for us
  - This feature has been available for some time but couldn't be used in iOS programming until recently due to a technical problem with the iOS simulator

Best Practice: property name ≠ att name

```
#import "Greeter.h"

@implementation Greeter

@synthesize greetingText=_greetingText;

@end
```

- Furthermore, best practice dictates that when you synthesize the property, you specify that the attribute name be different than the publicly available property name
    - Now, external code only uses the property and internal code can clearly indicate when it is using the property or the attribute

Use of Property

- When you have a property defined, you can access it using a syntax similar to accessing attributes in Java

```
Greeter *g = [[Greeter alloc] init];
g.greetingText = @“Howdy”
NSLog(@”%@, Ken!”, g.greetingText);
```

- In line 2, we are invoking the setter; In line 3, we are invoking the getter
    - Internal to the Greeter class, we can use the property via self.greetingText or the attribute with _greetingText

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)

- Example of extending 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)

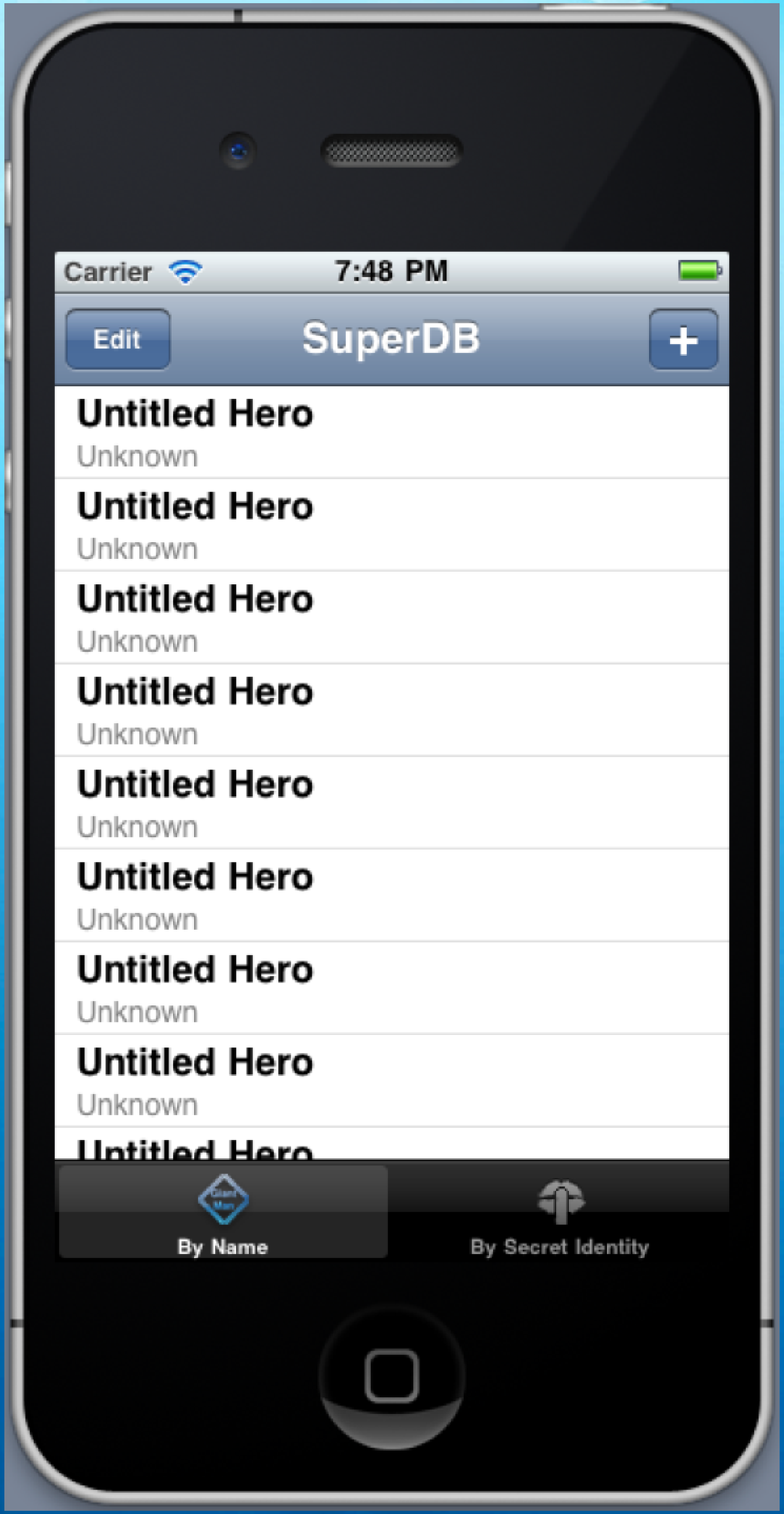
- To declare a variable or parameter to only accept instances of a certain protocol, you use the syntax

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

- You'll see examples of this in today's sample code

View Controllers

- As we learned last time, view controllers are a fundamental concept in iOS programming
  - We looked at basic concepts concerning view controllers last time
  - Now, we'll look at some of the more advanced view controllers that iOS provides
    - Tab Bar View Controller, Table View Controller and Navigation View Controller



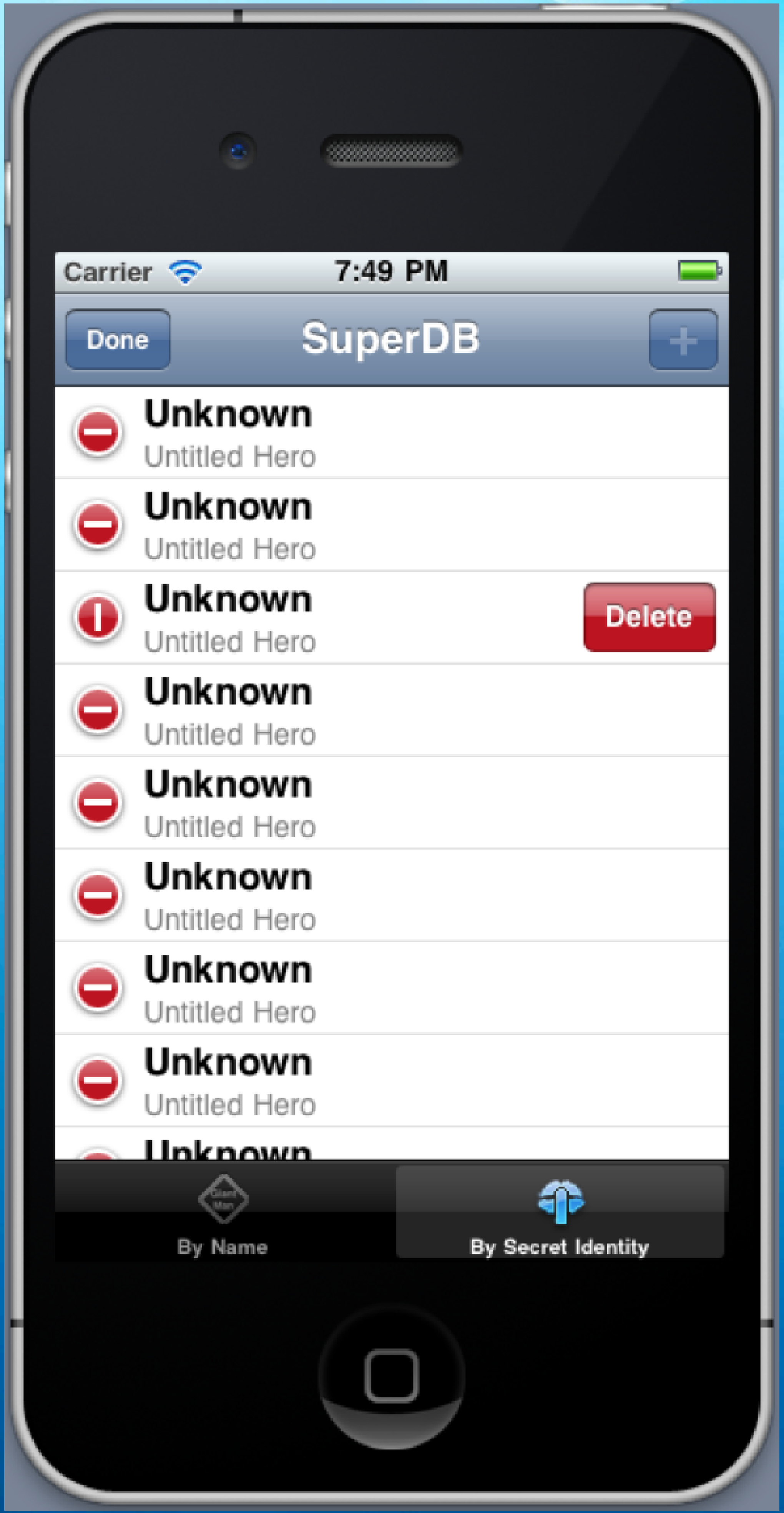
Navigation Bar

Table

- **The Navigation/TabBar/Table View Controllers make it easy to work with the navbar, table, and tabbar widgets**

- **Here we have a Navigation View Controller that is managing all three of these widgets at once**

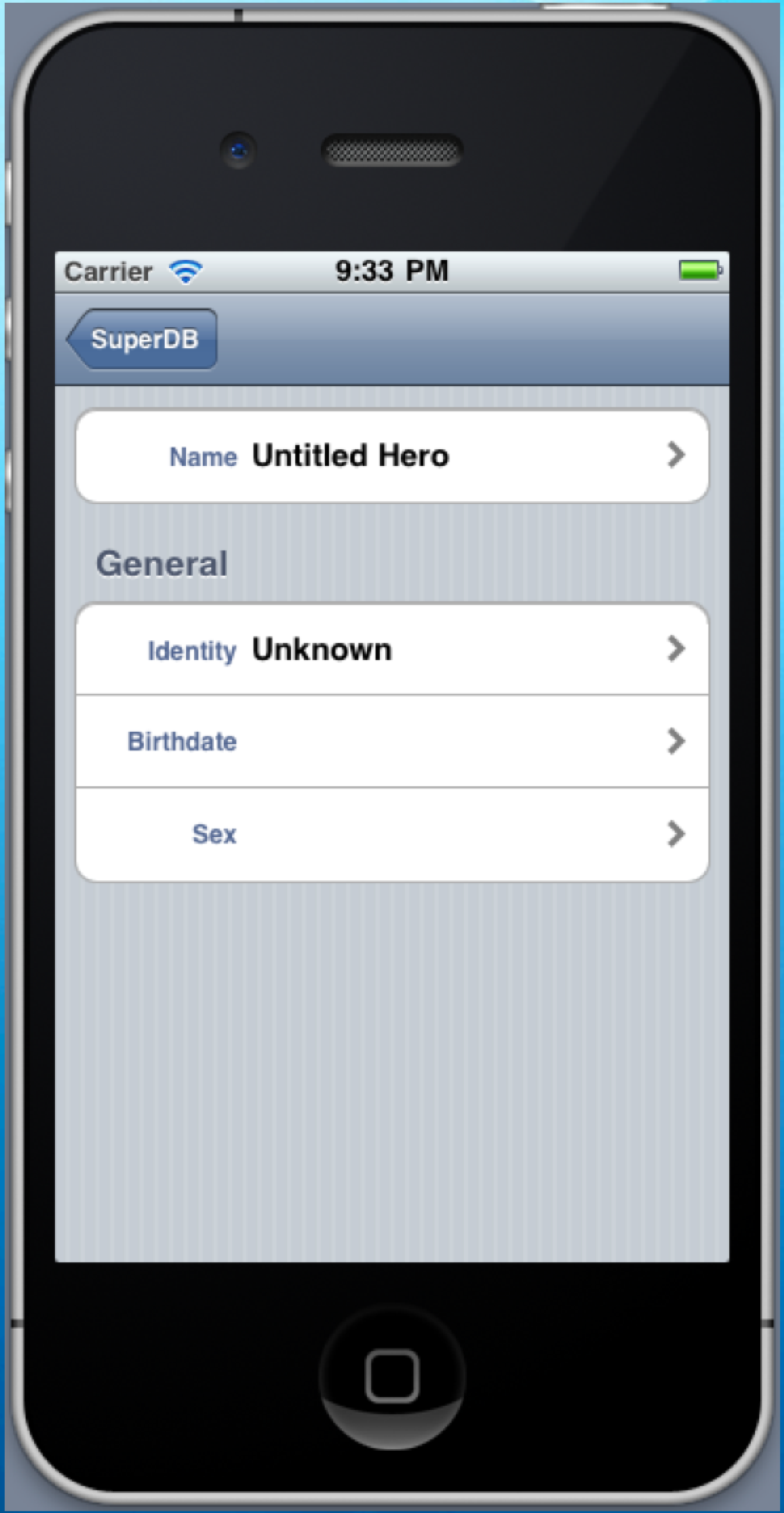
Tab Bar



Here we see that the table can be put in and out of edit mode. The buttons on the left and right are known as accessories; there are several different standard types available plus the ability to create your own

We see that the user can click on different tabs and have the view changed

Finally, the nav bar can have various buttons added to it that can handle both navigation between multiple view controllers as well as invoking commands



Here we see what happens when a view controller gets pushed onto the stack; its view takes over the screen (except for the navbar) and the navigation controller takes care of adding a button to the navbar that will allow the user to return to the previous view

We also see an instance of a "grouped" table, a table with sections, with each section allowed to have a different number of rows

The arrow on the right of each view is called a disclosure indicator and is another example of a built-in accessory for table cells

Tab Bar Controller

- An application that manages a set of view controllers (typically one view controller per tab)
    - Seen this way, all the tab bar view controller does is listen for clicks on the tab bar and swap in the appropriate view controller for the selected tab
        - The tab bar view controller always stays on the screen and is "in control"
            - The sub view controllers don't know about the tab bar and just focus on handling their view

Table View Controller

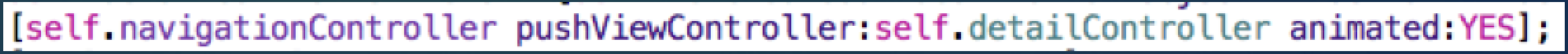
- Manages the display of a single-column, multi-section, multi-row table
  - Typically plays the role of data source for a table view
    - responding to requests for number of sections, number of rows, table cell requests, etc.
  - and plays the role of the delegate for a table view
    - responding to user interactions (selection, move, edit, delete)

Navigation View Controller

- Operates a stack of view controllers to help display hierarchical data structures
  - The navigation view controller presents the nav bar and then provides the rest of the space to another view controller
  - It starts by displaying a view controller that is designated the root view controller
    - other view controllers can then be pushed onto the stack and become visible or popped off the stack to reveal the previous view controller

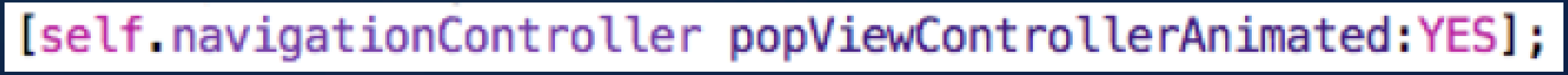
Pushing onto the Stack

- Here's an example of the code that is required to push a new view controller onto the stack



```
[self.navigationController pushViewController:self.detailController animated:YES];
```

- Typically, no code is needed to pop the top view controller off the stack, that's handled automatically by the navcontroller; if you do need to do it programmatically, the code looks like this:



```
[self.navigationController popViewControllerAnimated:YES];
```

The SuperDB Example (I)

- The SuperDB example shows off ways to best use a navigation controller in the presence of hierarchical data
  - It adopts a strategy of storing nested arrays in paired or linked arrays
    - paired arrays have the same number of elements in them but contain different but related content
      - Thus, if first_names and last_names were paired arrays, element 0 would represent one person, element 1 would represent another person, etc.

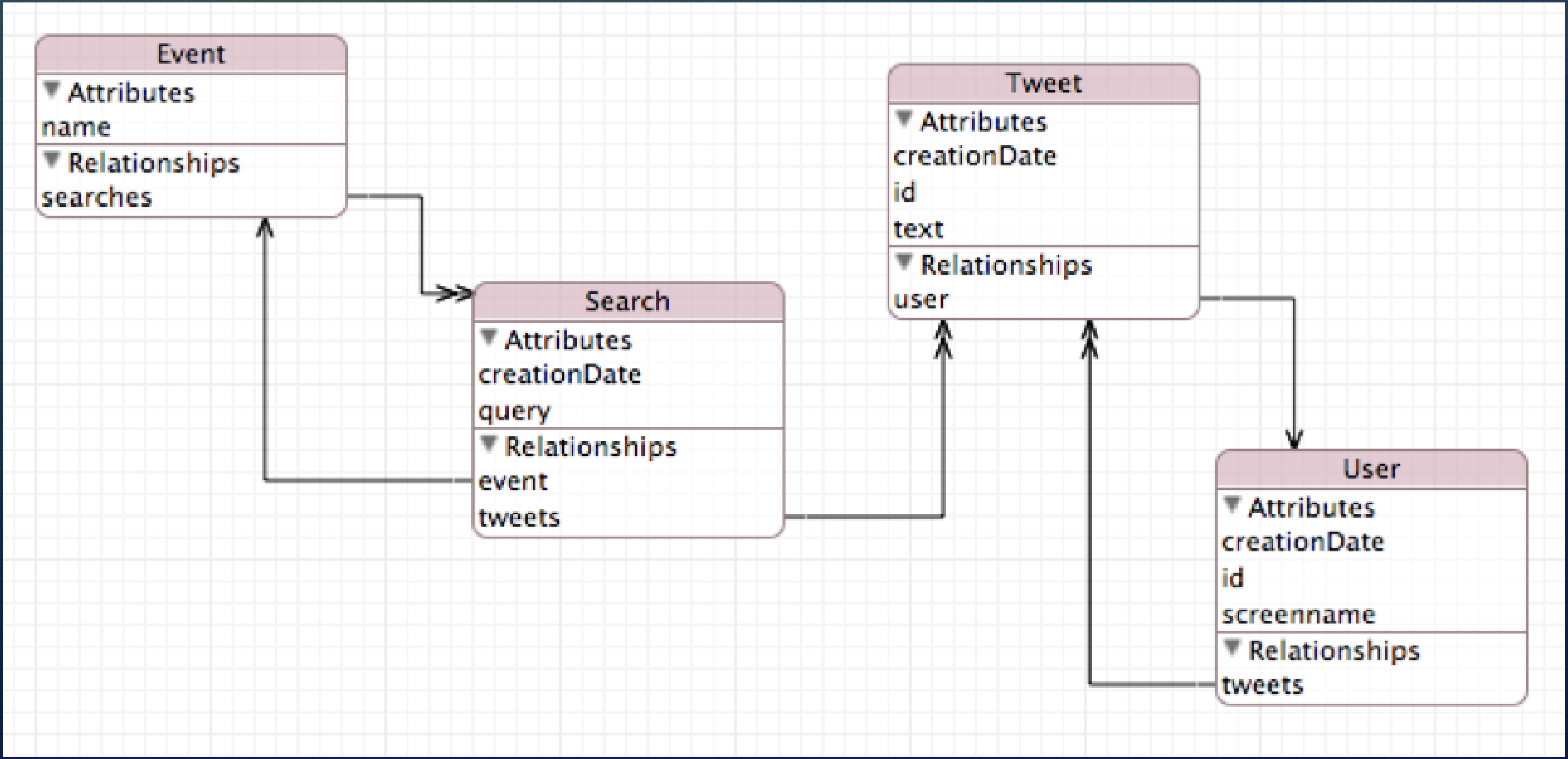
The SuperDB Example (II)

- The SuperDB example also demonstrates
  - dynamic class loading in Objective-C
  - creating objective-c categories and protocols
  - a new approach to Objective-C 2.0 properties
  - selection lists and date pickers
  - the use of navbars and navigation button items
  - the use of a simple Core Data data model

Core Data

- Core Data is Apple’s object-relational mapping framework
    - It allows developers to specify the data model of their application using a graphical editor; one to many and one to one relationships can be defined between entities
    - It then makes it easy to
        - generate an SQLite database, in-memory database, or flat binary file to store that data model
        - create, read, update and destroy objects defined by the data model

Example Data Model



Key Concepts (I)

- **Data Model:** Defines schema of persistent store
- **Persistent Store:** a database or flat binary file
- **Persistent Store Coordinator:** Manages access to a persistent store; ensures that requests are serialized

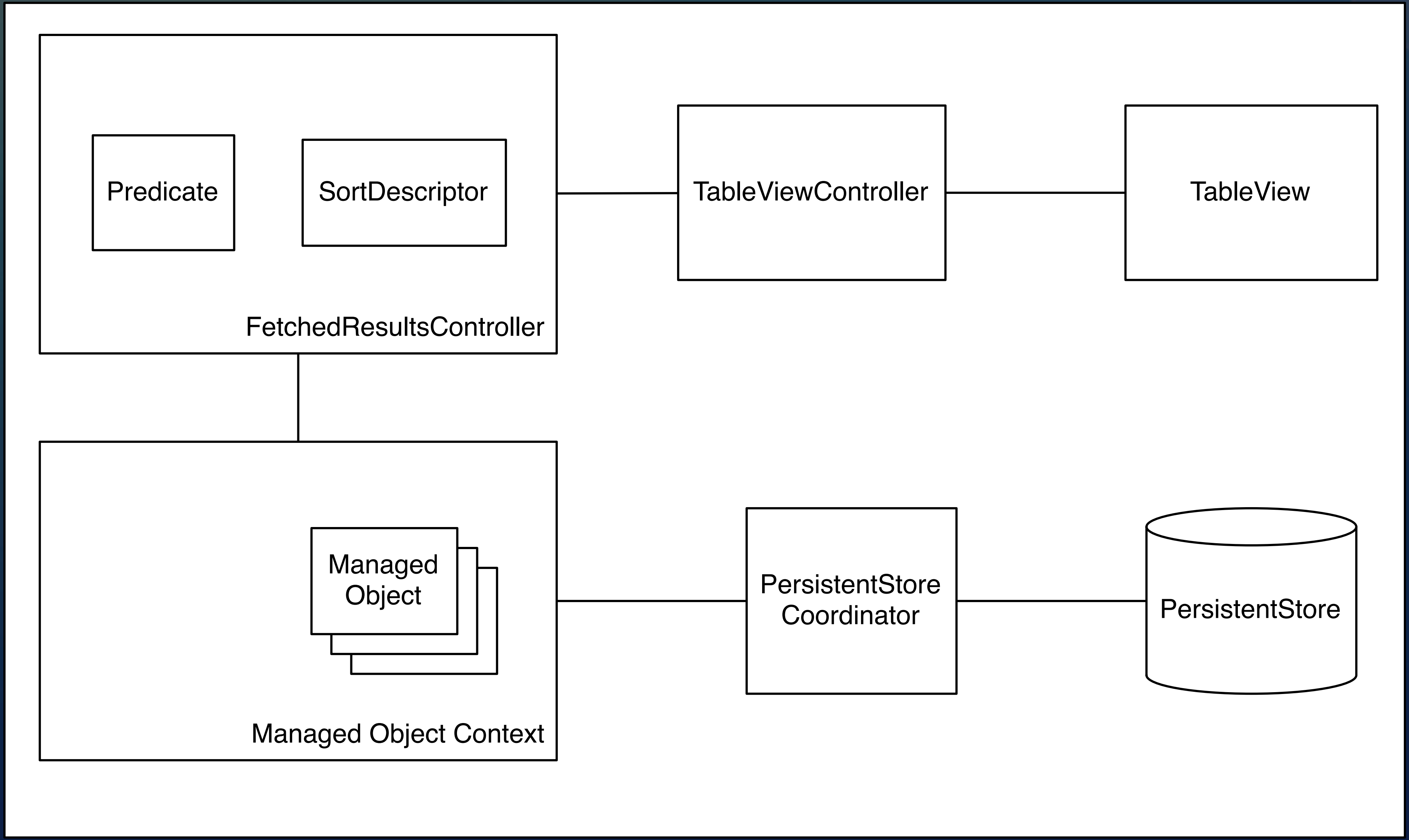
Key Concepts (I)

- **Managed Objects Context:** a context that uses the persistent store coordinator to access, update, or create new objects in the persistent store
- **Managed Objects:** run-time instances of the objects defined by the data model
  - These function as hash tables based on attribute names
- **Fetch Request:** applies predicates to the managed objects of a managed objects context and returns the ones that match

Life Cycle

- The default Core Data template
  - creates a persistent store in your app's document directory
  - creates a persistent store coordinator to access information from the persistent store
  - creates a managed object context to load objects from the persistent store into the application
  - creates a FetchedResultsController to make it straightforward to place managed objects into a UITableView; you configure this controller with query details and sort orders and then tell it to go

Core Data Life Cycle Architecture



Understanding the default template

- The code in the default template related to the fetched results controller is complex but this is due to the fact that it
  - tries to handle all the possible ways in which managed objects can change thus forcing the associated table to update
  - You can vastly simplify the code by simply reloading all of the table data whenever the fetched results controller tells you that a change has occurred
  - Let's look at some code

Dealing with Change: Migrations

- Core Data has the power to deal with changes to a data model over time
    - This is an important issue in that if you have previously distributed an application to users, you can't change the data model on them as the new version of the software won't support the databases or flat files created by the original version of the application
- But, with migrations, you can change the data model and support "migrating" their existing data to the new version

Migrations (I)

- In order to add support for migrations, you first select your existing data model and invoke the "Add Model Version" menu item located in the Editor menu (for XCode 4)
    - This creates a new copy of the data model and renames the old version by appending a version number to it
- You can then safely change the data model knowing that the original data model has been saved

Migrations (II)

- Core Data supports two types of migrations
  - Lightweight migrations and Standard migrations
- A lightweight migration is where having made simple changes to your data model
  - added a new entity or deleted an existing one;
  - added or deleted an attribute
- you have Core Data **automatically infer** the change and **do the migration by itself!**
- Standard migrations are for when significant changes to the data model; these require the developer to specify explicitly how to perform the migration

Migrations (III)

- Adding support for lightweight migrations is straightforward
  - In the code that instantiates a persistent store coordinator, you ask it to turn on two options
    - NSMigratePersistentStoresAutomaticallyOption
    - NSInferMappingModelAutomaticallyOption
  - when it then opens an existing store, it will detect if that store uses an old version of the data model and if so attempt to automatically migrate it to the new version (otherwise it throws an exception)

Core Data Wrap-Up

- Core Data is a very powerful framework
  - the examples here just scratch the surface
- Fortunately, there are plenty of books coming out on Core Data and Apple's extensive documentation of the framework to allow you to move forward
- Features we didn't look at include
  - custom managed objects and relationships between entities
  - queries that fetch entities across those relationships

MapKit and Location Based Services

- MapKit provides
  - a MapView that displays Google Maps
  - the ability to add annotations to a map view
    - think "push pins" on a Google map
  - the ability to display a callout when an annotation is selected
  - the ability to take lat/long and return street, city, state names (called reverse geocoding)

Working with a MapView

- In order to specify what a MapView displays, you need to create a MKCoordinateRegion, a struct of structs
    - its first struct, called center, is CLLocationCoordinate2D which contains lat/long info
    - its second struct, called span, is a MKCoordinateSpan, which contains a latitude and longitude deltas
- The span specifies how much around the center you want to reveal; the framework provides a handy function called regionThatFits which will reshape a MapView to fit within a view or window

Core Location

- Both MapViews and Geocoders can look up your current location
    - They both do this asynchronously
        - You write code that sets a class to be a delegate of the map view or geocoder
        - You then tell them to start searching
        - You then wait for your delegate methods to be called with the results, allowing you to update the MapView to the discovered location

Demo

Working with Web Data

- Example program demonstrates various ways of retrieving data over the Web
  - synchronously via convenience methods on various classes or NSURLConnection
  - or asynchronously via NSURLConnection
    - in this case, the request happens in a separate thread
    - delegate methods are then invoked to track the progress of downloading a resource

Demo

Wrapping Up

- Reviewed additional iOS topics
  - Additional Objective-C 2.0 techniques
    - properties, categories, protocols
  - Table and Navigation controller
  - Core Data
  - MapKit: MapView and Core Location
  - Retrieving Data from the Web

Coming Up Next

- SPRING BREAK!!!!