

Developing Applications for iOS



Lecture 10: Core Data and Categories

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

- Core Data and Documents

This is how you store something serious in iOS.

Easy entry point into iCloud.

- `NSNotificationCenter`

The little “radio station” we talked about in the first lecture.

- Objective-C Categories

A way to add methods to a class without subclassing.

Core Data

- We are object-oriented programmers and we don't really like C APIs. We want to store our data using object-oriented programming.

Welcome to Core Data

- This is an object-oriented database.
- It's a way of creating an object graph backed by a database (usually SQL).

How does it work?

- Create a visual mapping (using Xcode tool) between database and objects. Create and query for objects using object-oriented API.
- Access the “columns in the database table” using `@property`s on those objects.

Core Data

The screenshot shows the Xcode interface with the 'Choose a template for your new file' dialog open. The dialog is divided into two main sections: 'iOS' and 'Mac OS X'. Under 'iOS', the 'Core Data' option is selected, and a callout box points to the 'Data Model' icon. The callout box contains the text: 'For creating a visual map of your application's database objects go to "New File ..." then Data Model under Core Data section.'

The background shows the Xcode project 'CoreDataDemo.xcodeproj' with the following structure:

- CoreDataDemo (1 target, iOS SDK 5.1)
 - AppDelegate.h
 - AppDelegate.m
 - MainStoryboard.storyboard
 - ViewController.h
 - ViewController.m
 - Supporting Files
 - Frameworks
 - Products

The right sidebar shows the 'Identity' panel with the following information:

- Project Name: CoreDataDemo
- Location: Not Applicable
- Full Path: /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo.xcodeproj

The 'Project Document' panel shows:

- Project Format: Xcode 3.2-compatible
- Organization: [empty]
- Class Prefix: [empty]

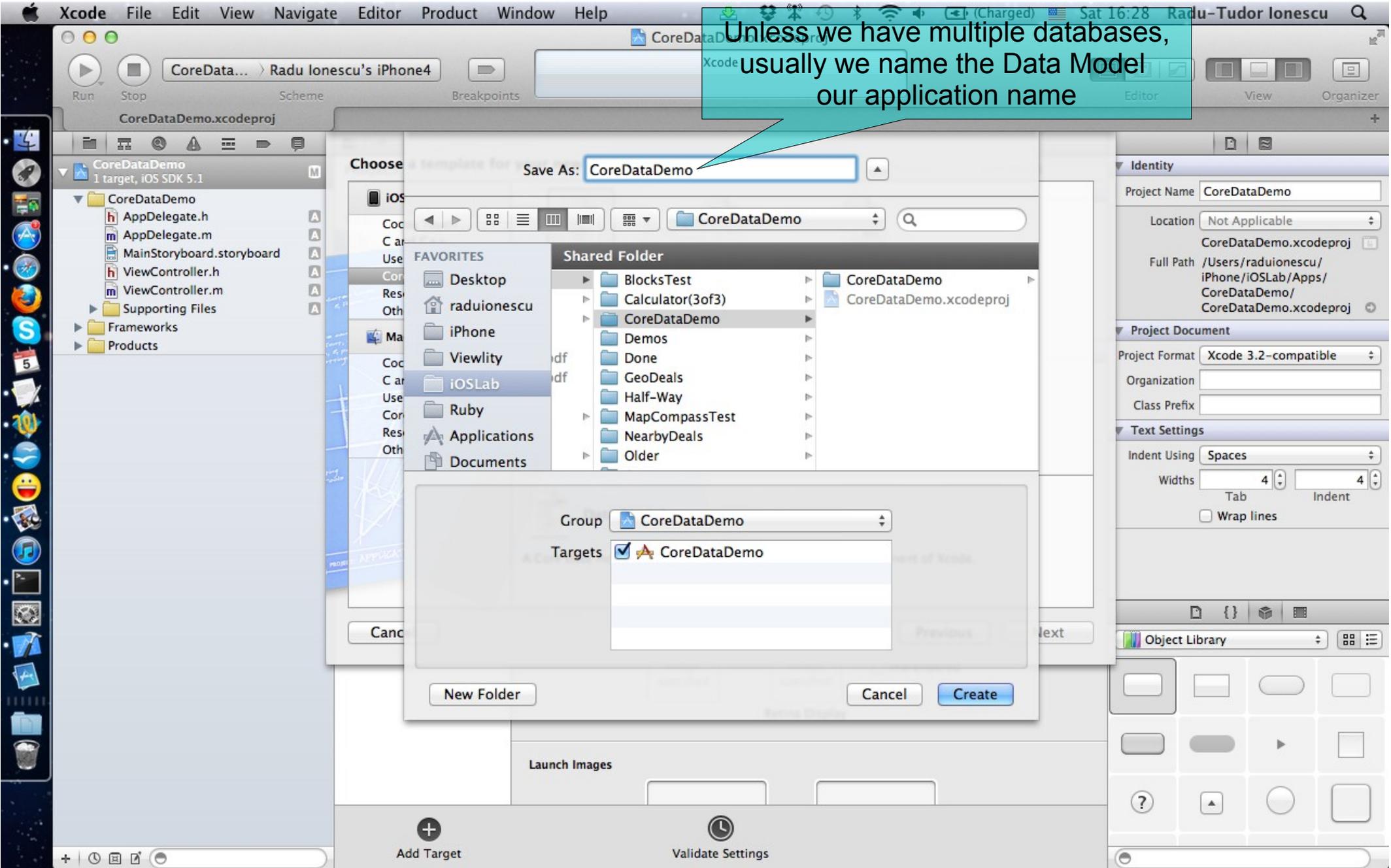
The 'Text Settings' panel shows:

- Indent Using: Spaces
- Widths: Tab (4), Indent (4)
- Wrap lines: [unchecked]

The 'Object Library' panel is visible at the bottom right, showing various UI components.

Core Data

Unless we have multiple databases, usually we name the Data Model our application name



Core Data

The screenshot shows the Xcode IDE interface. The top menu bar includes 'Xcode', 'File', 'Edit', 'View', 'Navigate', 'Editor', 'Product', 'Window', and 'Help'. The status bar at the top right shows 'Sat 16:30' and 'Radu-Tudor Ionescu'. The main workspace is divided into several panes:

- Project Navigator (Left):** Shows the project structure for 'CoreDataDemo'. The file 'CoreDataDemo.xcdatamodeld' is selected and highlighted in blue.
- Entity Inspector (Middle-Left):** Displays the 'Core Data Model' configuration. It has sections for 'ENTITIES', 'FETCH REQUESTS', and 'CONFIGURATIONS'. The 'Attributes' section is expanded, showing a table with columns 'Attribute' and 'Type'. Below it, the 'Relationships' section is expanded, showing a table with columns 'Relationship', 'Destination', and 'Inverse'. The 'Fetched Properties' section is also expanded, showing a table with columns 'Fetched Property' and 'Predicate'.
- Identity Inspector (Right):** Shows the 'Identity' and 'Core Data Model' settings. The 'Group Name' is 'CoreDataDemo.xcdatamodeld'. The 'Path' is 'Relative to Project'. The 'Full Path' is '/Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo.xcdatamodeld'. The 'Core Data Model' section shows the 'Identifier' as 'Model Version Identifier'. The 'Tools Version' is 'Xcode 4.1'. The 'Versioned Core Data Model' section shows the 'Current' model as 'CoreDataDemo'. The 'Deployment Targets' section shows 'Mac OS X' as 'Target Default' and 'iOS' as 'Target Default'.

A callout box with a light blue background and a white border points to the 'CoreDataDemo.xcdatamodeld' file in the Project Navigator. The text inside the callout box reads: 'The Data Model file. Sort of like a storyboard for databases.'

At the bottom of the Xcode window, there are several icons: 'Outline Style', 'Add Entity', 'Add Attribute', and 'Editor Style'.

Core Data

The Data Model consists of:

- Entities
- Attributes
- Relationships

The screenshot shows the Xcode interface for editing a Core Data model. The central editor area is divided into several sections: 'Entities' (currently empty), 'Attributes' (with a table header 'Attribute' and 'Type'), 'Relationships' (with a table header 'Relationship', 'Destination', and 'Inverse'), and 'Fetched Properties' (with a table header 'Fetched Property' and 'Predicate'). The right-hand inspector shows the 'Identity' section with 'Group Name' set to 'CoreDataDemo.xcdatamodeld' and 'Path' set to 'Relative to Project'. The 'Core Data Model' section shows 'Identifier' set to 'Model Version Identifier'. The 'Versioned Core Data Model' section shows 'Current' set to 'CoreDataDemo'. The 'Deployment Targets' section shows 'Mac OS X' and 'iOS' both set to 'Target Default'. The bottom toolbar includes 'Outline Style', 'Add Entity', 'Add Attribute', and 'Editor Style' buttons.

Core Data

The screenshot shows the Xcode interface for configuring a Core Data model. The 'Entities' pane on the left shows a table with one entity named 'Movie'. The 'Attributes' pane is empty. The 'Relationships' pane is empty. The 'Fetches Properties' pane is empty. The 'Identity and Type' pane shows the file name 'CoreDataDemo.xcdatamodel' and the full path. The 'Core Data Model' pane shows the identifier 'Model Version Identifier' and the tools version 'Xcode 4.1'. The 'Target Membership' pane shows the target 'CoreDataDemo' is checked. The 'Object Library' pane is visible at the bottom right.

Then type the name here. We will call this first Entity Movie.

An Entity will appear in our code as an (or a subclass of an) `NSManagedObject`.

Click here to add an Entity.

Core Data

Xcode File Edit View Navigate Editor Product Window Help

CoreDataDemo.xcodeproj CoreDataDemo.xcdatamodel

Run Stop Scheme Breakpoints Xcode Project 1

Untitled

By File By Type

CoreDataDemo project 1 issue

CoreDataDemo.xcdatamodel

Misconfigured Property
Movie.title must have a defined type

ENTITIES

Movie

FETCH REQUESTS

CONFIGURATIONS

Default

Attributes

Attribute	Type
U title	Undefined

Relationships

Relationship

Destination Inverse

Fetch Request Properties

Fetches Property

Predicate

Identity and Type

File Name CoreDataDemo.xcdatamodel

File Type Default - Core Data Model

Location Relative to Group

CoreDataDemo.xcdatamodel

Full Path /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo.xcdatamodel/CoreDataDemo.xcdatamodel

Core Data Model

Identifier Model Version Identifier

Tools Version

Minimum Xcode 4.1

Target Membership

CoreDataDemo

Object Library

Outline Style Add Entity Add Attribute Editor Style

Notice that we have an error. That's because our Attribute needs a type.

Now we will add some Attributes. We will start with title. Click here to add an Attribute.

Then edit the name of the Attribute here.

Core Data

Attributes are accessed on our `NSManagedObjects` via the methods `valueForKey:` and `setValueForKey:`. Or, if we subclass `NSManagedObject`, we can access Attributes as `@property`s.

Set the type of the `title` Attribute to `String`. Note that all Attributes are objects:

- Numeric ones are `NSNumber`.
- Boolean is also `NSNumber`.
- Binary Data is `NSData`.
- Date is `NSDate`.
- String is `NSString`.
- Don't worry about `Transformable`.

Core Data

CoreDataDemo.xcodeproj — CoreDataDemo.xcdatamodel

CoreDataDemo > iOS Device

Run Stop Scheme Breakpoints Xcode

By File By Type

ENTITIES

- E Movie

FETCH REQUESTS

CONFIGURATIONS

- Default

Attributes

Attribute	Type
N duration	Integer 16
posterData	Binary Data
N rating	Float
S synopsis	String
S title	String
N year	Integer 16

Relationships

Relationship	Destination	Inverse
--------------	-------------	---------

Fetches Properties

Fetches Property	Predicate
------------------	-----------

Identity and Type

File Name CoreDataDemo.xcdatamodel

File Type Default - Core Data Model

Location Relative to Group

Full Path /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo.xcdatamodel/CoreDataDemo.xcdatamodel

Core Data Model

Identifier Model Version Identifier

Tools Version

Minimum Xcode 4.1

Target Membership

- CoreDataDemo

No Issues

Outline Style Add Entity Add Attribute Editor Style

Here are a whole bunch of more Attributes.

You can see your Entities and Attributes in graphical form by clicking here.

Core Data

The screenshot shows the Xcode interface with the Core Data Model editor open. The main canvas displays a 'Movie' entity with the following attributes: duration, posterData, rating, synopsis, title, and year. The right-hand sidebar contains the 'Identity and Type' panel (showing file name and path), the 'Core Data Model' panel (showing identifier and tools version), and the 'Target Membership' panel (showing the model is included in the 'CoreDataDemo' target). The bottom of the interface features a toolbar with icons for 'Outline Style', 'Add Entity', 'Add Attribute', and 'Editor Style'. A teal callout box with a pointer to the entity diagram contains the text: 'This is the same thing we were just looking at, but in a graphical view.'

CoreDataDemo.xcodeproj — CoreDataDemo.xcdatamodel

This is the same thing we were just looking at, but in a graphical view.

CoreDataDemo.xcdatamodel

Movie

- Attributes
 - duration
 - posterData
 - rating
 - synopsis
 - title
 - year
- Relationships

Identity and Type

File Name: CoreDataDemo.xcdatamodel

File Type: Default - Core Data Model

Location: Relative to Group

Full Path: /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo.xcdatamodel/CoreDataDemo.xcdatamodel

Core Data Model

Identifier: Model Version Identifier

Tools Version

Minimum: Xcode 4.1

Target Membership

- CoreDataDemo

Outline Style Add Entity Add Attribute Editor Style

Core Data

The image shows the Xcode interface for editing a Core Data model. The main workspace is a grid where entities are represented as boxes. A 'Genre' entity box is currently selected, showing its 'Attributes' and 'Relationships' sections. To the left, the 'ENTITIES' list shows 'Genre' and 'Movie'. The right-hand pane displays the 'Entity' inspector for 'Genre', showing its name, class (NSManagedObject), and other settings. At the bottom, there are buttons for 'Add Entity' and 'Add Attribute'. Several light blue callout boxes with white text provide instructions: 'And set its name to Genre.' points to the 'Genre' entity in the list; 'A graphical version will appear.' points to the 'Genre' box in the graph; 'These can be dragged around and positioned around the center of the graph.' points to the 'Genre' box; 'Add another Entity.' points to the 'Add Entity' button; and 'Attributes can be added in the Graphic Editor too.' points to the 'Add Attribute' button.

And set its name to Genre.

A graphical version will appear.

These can be dragged around and positioned around the center of the graph.

Add another Entity.

Attributes can be added in the Graphic Editor too.

Core Data

We can edit the attribute directly by double-clicking on it or on the (Data Model) Inspector if we prefer.

Here we add an Attribute called name to Genre.

Let's set its type to String as well.

The screenshot shows the Xcode interface with the following elements:

- Entities:** Genre, Movie
- Fetch Requests:** Default
- Configurations:** Default
- Movie Entity Attributes:** duration, posterData, rating, synopsis, title, year
- Genre Entity Attributes:** name
- Attribute Inspector:** Name: name, Properties: Transient (unchecked), Optional (checked), Indexed (unchecked), Attribute Type: String (selected)

Core Data

The screenshot shows the Xcode interface for editing a Core Data model. The main canvas displays two entities: 'Movie' and 'Genre'. The 'Movie' entity has attributes: duration, posterData, rating, synopsis, title, and year. The 'Genre' entity has an attribute: name. A relationship arrow points from the 'newRelationship' relationship field of the 'Movie' entity to the 'newRelationship' relationship field of the 'Genre' entity. A light blue callout box with a pointer to the relationship arrow contains the text: "Similar to outlets and actions, we can CTRL-drag to create Relationships between Entities." The interface includes a top menu bar, a toolbar with Run and Stop buttons, a left sidebar with 'By File' and 'By Type' views, and a right sidebar with 'Entity' details (Name: Multiple Values, Class: NSObject) and 'User Info'.

Similar to outlets and actions, we can CTRL-drag to create Relationships between Entities.

Core Data

The screenshot shows the Xcode interface for editing a Core Data model. The main workspace displays two entities: **Movie** and **Genre**. The **Movie** entity has attributes: duration, posterData, rating, synopsis, title, and year. It also has a relationship named **newRelationship**. The **Genre** entity has an attribute named **name** and a relationship named **newRelationship**. A line connects the **newRelationship** attribute of the **Movie** entity to the **newRelationship** attribute of the **Genre** entity. A callout box with a teal background and white text points to the **newRelationship** attribute in the **Movie** entity, containing the text: "Click on the newRelationship in Movie."

The interface includes a top menu bar (Xcode, File, Edit, View, Navigate, Editor, Product, Window, Help), a toolbar with Run and Stop buttons, and a sidebar with a list of entities (Genre, Movie) and configurations (Default). The right-hand pane shows the details for the selected entity, including Name (Multiple Values), Class (NSObject), and various options like Abstract Entity and Parent Entity. The bottom toolbar contains icons for Outline Style, Add Entity, Add Attribute, and Editor Style.

Core Data

This Relationship to the Genre is “what kind” of Movie, so we will call this Relationship whatKind.

The screenshot displays the Xcode interface for configuring a Core Data model. The main canvas shows two entities: 'Movie' and 'Genre'. The 'Movie' entity has attributes: duration, posterData, rating, synopsis, title, and year. The 'Genre' entity has attributes: name and a relationship named 'newRelationship'. A relationship named 'whatKind' is being configured between 'Movie' and 'Genre'. The 'Genre' entity is selected in the 'Relationship' inspector on the right, showing the following configuration:

- Name: whatKind
- Destination: Genre
- Inverse: newRelationship
- Properties: Transient, Optional
- Arranged: Ordered
- Plural: To-Many Relationship
- Count: 1 (with Minimum and Maximum)
- Delete Rule: Nullify
- Advanced: Index in Spotlight, Store in External Record File

The 'User Info' section shows a table with 'Key' and 'Value' columns. The 'Versioning' section shows 'Hash Modifier' set to 'Version Hash Modifier'. The 'Object Library' is visible at the bottom right.

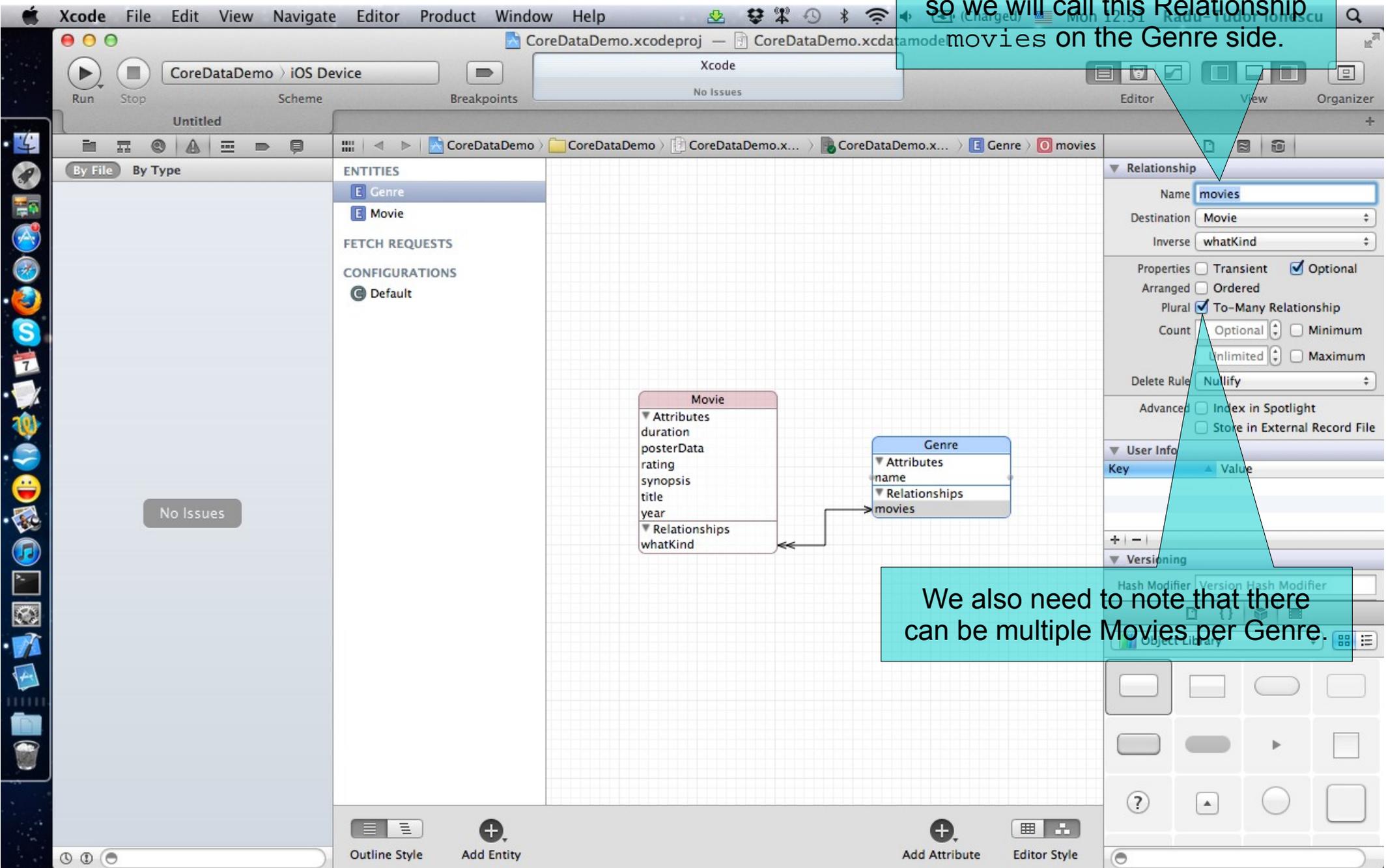
Core Data

The screenshot shows the Xcode interface for configuring a Core Data model. The main canvas displays two entities: 'Movie' and 'Genre'. The 'Movie' entity has attributes: duration, posterData, rating, synopsis, title, and year. It also has a relationship named 'whatKind' that points to the 'Genre' entity. The 'Genre' entity has an attribute named 'name' and a relationship named 'newRelationship' that points back to the 'Movie' entity. The right-hand pane is set to the 'Relationship' configuration view for the 'whatKind' relationship. The configuration includes: Name: whatKind, Destination: Genre, Inverse: newRelationship, Properties: Optional (checked), Arranged: Ordered, Plural: To-Many Relationship, Count: 1 (checked for Minimum and Maximum), and Delete Rule: Nullify. The bottom toolbar shows 'Add Entity', 'Add Attribute', and 'Editor Style' buttons.

Now click on the newRelationship in Movie.

Core Data

A Genre can have many Movies, so we will call this Relationship movies on the Genre side.



We also need to note that there can be multiple Movies per Genre.

Core Data

The screenshot shows the Xcode interface for editing a Core Data model. The main canvas displays two entities: 'Movie' and 'Genre'. The 'Movie' entity has attributes: duration, posterData, rating, synopsis, title, and year. It also has a relationship named 'whatKind' that points to the 'Genre' entity. The 'Genre' entity has an attribute named 'name' and a relationship named 'movies' that points back to the 'Movie' entity. The right-hand pane shows the configuration for the 'movies' relationship, including its name, destination (Movie), inverse (whatKind), and other properties like 'Optional' and 'To-Many Relationship'. The bottom of the screen shows the 'Outline Style' and 'Add Entity' buttons.

Note the Data Model's recognition of the "inverse" of this Relationship.

The type of this Relationship in our Objective-C code will be `NSManagedObject` (or a subclass of `NSManagedObject`).

The type of this Relationship in our Objective-C code will be `NSSet` (since it is a "to many" Relationship).

Core Data

So how do you access all of this stuff in your code?

- You need an `NSManagedObjectContext`.
- It is the hub around which all Core Data activity turns.

How do you get one?

- There are two ways:
 1. Create a `UIManagedDocument` and ask for its `managedObjectContext` (`a @property`).
 2. Click the “Use Core Data” button when you create an Empty Application Project. Then your `AppDelegate` will have a `managedObjectContext @property`.
- We are going to focus on doing the first one.

UIManagedDocument

UIManagedDocument

- It inherits from UIDocument which provides a lot of mechanism for the management of storage.
- If you use UIManagedDocument, you'll be on the fast-track to iCloud support.
- Think of a UIManagedDocument as simply a container for your Core Data database.
- Creating a UIManagedDocument:

```
UIManagedDocument *document =  
    [[UIManagedDocument alloc] initWithFileURL:url];
```

UIManagedDocument

But you must open/create the document to use it

- Check to see if it exists:

```
[[NSFileManager defaultManager] fileExistsAtPath:[url path]]
```

- If it does, open the document using:

```
- (void)openWithCompletionHandler:  
    (void (^)(BOOL success))completionHandler;
```

- If it does not, create it using:

```
- (void)saveToURL:(NSURL *)url  
forSaveOperation:(UIDocumentSaveOperation)operation  
completionHandler:(void (^)(BOOL success))completionHandler;
```

What is that `completionHandler`?

- Just a block of code to execute when the open/save completes.
- That's needed because the open/save is asynchronous. Do not ignore this fact!

UIManagedDocument

- Example:

```
self.document = [[UIManagedDocument alloc]
                 initWithFileURL:(NSURL *)url];
if ([[NSFileManager defaultManager]
    fileExistsAtPath:[url path]])
{
    [document openWithCompletionHandler:^(BOOL success) {
        if (success) [self documentIsReady];
        else NSLog(@"Couldn't open document at %@", url);
    }];
}
else
{
    [sourceDocument saveToURL:url
                     forSaveOperation:UIDocumentSaveForCreating
                     completionHandler:^(BOOL success) {
        if (success) [self openDocument];
        else NSLog(@"Couldn't create document at %@", url);
    }];
}
/* Can't do anything with the document yet.
 * Do it in documentIsReady. */
```

UIManagedDocument

- Once document is open/created, you can start using it. But you might want to check its `documentState` when you do:

```
- (void)documentIsReady
{
    if (self.document.documentState == UIDocumentStateNormal)
    {
        NSManagedObjectContext *context =
            self.document.managedObjectContext;
        // do something with the Core Data context
    }
}
```

UIManagedDocument

Other documentStates

- UIDocumentStateClosed (not opened or file does not exist yet).
- UIDocumentStateSavingError (success will be NO).
- UIDocumentStateEditingDisabled (temporarily unless failed to revert to saved).
- UIDocumentStateInConflict (e.g., because some other device changed it via iCloud).

The documentState is often “observed”

- So it's about time we talked about using NSNotifications to observe other objects.

NSNotification

NSNotificationCenter

- Get the default notification center via:

```
[NSNotificationCenter defaultCenter]
```

- Then send it the following message if you want to observe another object:

```
- (void)addObserver:(id)observer  
    selector:(SEL)methodToSendIfSomethingHappens  
    name:(NSString *)name  
    object:(id)sender;
```

The meaning of the arguments

- **observer** is the object to get notified;
- **name** is what you are observing (a constant somewhere);
- **sender** is the object whose changes you're interested in (`nil` is anyone's).

NSNotification

NSNotificationCenter

- You will then be notified when the named event happens:

```
- (void)methodToSendIfSomethingHappens:
    (NSNotification *)notification
{
    NSString* name = notification.name
    // the name passed above

    id obj = notification.object
    // the object sending you the notification

    NSDictionary *info = notification.userInfo;
    // notification-specific information about what happened
}
```

NSNotification

Example

```
NSNotificationCenter *center =  
    [NSNotificationCenter defaultCenter];
```

- Watching for changes in a document's state:

```
[center addObserver:self  
         selector:@selector(documentChanged:)  
         name:UIDocumentStateChangedNotification  
         object:self.document];
```

- Don't forget to remove yourself when you're done watching:

```
[center removeObserver:self];  
  
[center removeObserver:self  
         name:UIDocumentStateChangedNotification  
         object:self.document];
```

- Failure to remove yourself can sometimes result in crashes.
- This is because the NSNotificationCenter keeps an “unsafe unretained” pointer to you.

NSNotification

Another Example

- Watching for changes in a CoreData database (made via a given `NSManagedObjectContext`):

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [center addObserver:self
               selector:@selector(contextChanged:)
               name:NSManagedObjectContextObjectsDidChangeNotification
               object:self.document.managedObjectContext];
}

- (void)viewWillDisappear:(BOOL)animated
{
    [center removeObserver:self
               name:NSManagedObjectContextObjectsDidChangeNotification
               object:self.document.managedObjectContext];
    [super viewWillDisappear:animated];
}
```

- There's also an `NSManagedObjectContextDidSaveNotification`.

NSNotification

Receiving the `NSManagedObjectContext` notifications

- `NSManagedObjectContextObjectsDidChangeNotification` or `NSManagedObjectContextDidSaveNotification`:
 - `(void)contextChanged:(NSNotification *)notification`

```
{  
    NSDictionary *info = notification.userInfo;  
}
```

The `info` `NSDictionary` contains the following keys

- `NSInsertedObjectsKey` gives an array of objects which were inserted.
- `NSUpdatedObjectsKey` gives an array of objects whose attributes changed.
- `NSDeletedObjectsKey` gives an array of objects which were deleted.

NSNotification

Other things to observe

- Look in the documentation for various classes in iOS.
- They will document any notifications they will send out.
- You can post your own notifications too. We did this in the NearbyDeals app that we created in our Labs:

```
[[NSNotificationCenter defaultCenter]  
 postNotificationName:@"locationUpdateNotification"  
 object:self];
```

```
[[NSNotificationCenter defaultCenter]  
 addObserver:self  
 selector:@selector(showMapRegionForNotification:)  
 name:@"locationUpdateNotification"  
 object:[DealsModel sharedModel]];
```

- See `NSNotificationCenter` documentation for more information.
- Don't abuse this mechanism!
- Don't use it to essentially get "global variables" in your application.

UIManagedDocument

Saving a document (like creating or opening) is also asynchronous

- Documents are auto-saved, but you can explicitly save as well.
- You use the same method as when creating, but with a different “save operation”:

```
[self.doc saveToURL:self.doc.fileURL
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success) {

    if (!success)
        NSLog(@"Save failed for %@", self.doc.localizedName);
}];

/* The document is not saved at this point in the
 * code (only once the block above executes). */
```

- Note the two UIManagedDocument properties used:

```
@property (nonatomic, strong) NSURL *fileURL;
// specified originally in initWithFileURL:

@property (readonly) NSString *localizedName;
```

UIManagedDocument

Closing a document is also asynchronous

- The document will be closed if there are no strong pointers left to the UIManagedDocument.
- But you can close it explicitly as well:

```
[self.doc closeWithCompletionHandler:^(BOOL success) {  
    if (!success)  
        NSLog(@"Close failed for %@", self.doc.localizedName);  
}]  
  
/* The document is not closed at this point in the  
* code (only once the block above executes). */
```

UIManagedDocument

Multiple instances of UIManagedDocument on the same document

- This is perfectly legal, but understand that they will **not** share an `NSManagedObjectContext`.
- Thus, changes in one will not automatically be reflected in the other.
- You'll have to refetch in other `UIManagedDocuments` after you make a change in one.
- Conflicting changes in two different `UIManagedDocuments` would have to be resolved by you!
- It's exceedingly rare to have two "writing" instances of `UIManagedDocument` on the same file.
- But a single writer and multiple readers? Not so rare. Just need to know when to refetch.

Core Data

Inserting objects into the database

- We grabbed an `NSManagedObjectContext` from an open `UIManagedDocument`'s `managedObjectContext` @property.
- Now we use it to insert/delete objects in the database and query for objects in the database:

```
NSManagedObject *movie = [NSEntityDescription  
    insertNewObjectForEntityForName:@"Movie"  
    inManagedObjectContext:managedObjectContext];
```

- Note that this `NSEntityDescription` class method returns an `NSManagedObject` instance.
- All objects in the database are represented by `NSManagedObjects` or by subclasses of `NSManagedObjects`.
- An instance of `NSManagedObject` is a manifestation of an Entity in our Core Data model (the model that we just graphically built in Xcode).
- All the Attributes of a newly-inserted object will be `nil` (unless you specify a default value in Data Model Inspector).

Core Data

How to access Attributes in an `NSManagedObject` instance

- You can access the Attributes using the following two `NSKeyValueObserving` protocol methods:
 - `(id)valueForKey:(NSString *)key;`
 - `(void)setValue:(id)value forKey:(NSString *)key;`
- You can also use `valueForKeyPath:/setValue:forKeyPath:` and it will follow your Relationships!

Core Data

How to access Attributes in an `NSManagedObject` instance

- The **key** is an Attribute name in your data mapping.

For example, `@"posterData"`.

- The **value** is whatever is stored (or to be stored) in the database.

It will be `nil` if nothing has been stored yet (unless Attribute has a default value in Xcode).

- Note that all values are objects (numbers and booleans are `NSNumber` objects).
- Binary data values are `NSData` objects.
- Date values are `NSDate` objects.
- “To-many” mapped relationships are `NSSet` objects (or `NSOrderedSet` if ordered).
- Non-“to-many” relationships are `NSManagedObjects`.

Core Data

Changes (writes) only happen in memory, until you save

- Yes, `UIManagedDocument` auto-saves.
- But explicitly saving when a batch of changes is made is good practice.

Core Data

Calling `valueForKey:` and `setValue:forKey:` is pretty messy

- There's no type-checking.
- And you have a lot of literal strings in your code (e.g. `@"postData"`).

What we really want is to set/get using `@property`s

- The solution is to create a subclass of `NSManagedObject`.
- The subclass will have `@property`s for each attribute in the database.
- We name our subclass the same name as the Entity it matches (not strictly required, but it is recommended to do so).
- And, as you might imagine, we can get Xcode to generate both the header file `@property` entries, and the corresponding implementation code (which is **not** `@synthesize`, so **be careful** with this).

Core Data

The screenshot shows the Xcode interface for a Core Data model. The main canvas displays two entities: **Movie** and **Genre**. The **Movie** entity has attributes: duration, posterData, rating, synopsis, title, and year. It also has a relationship named **whatKind**. The **Genre** entity has an attribute named **name** and a relationship named **movies**. A relationship line connects the **whatKind** relationship of the **Movie** entity to the **movies** relationship of the **Genre** entity. A light blue callout box points to the **Genre** and **Movie** entities in the left sidebar, containing the text: "Select both Entities. We are going to have Xcode generate NSObject subclasses for them for us."

Entities:

- Genre
- Movie

Movie Entity Attributes:

- duration
- posterData
- rating
- synopsis
- title
- year

Movie Entity Relationships:

- whatKind

Genre Entity Attributes:

- name

Genre Entity Relationships:

- movies

Entity Inspector (Right Panel):

- Name: Multiple Values
- Class: NSObject
- Abstract Entity:
- Parent Entity: [Dropdown]
- Indexes: [Empty List]
- User Info: Key-Value pairs
- Versioning: Hash Modifier (Version Hash Modifier), Renaming ID (Renaming Identifier)
- Entity Sync: [Options]

Object Library (Bottom Right):

- Buttons, Text Fields, Labels, etc.

Core Data

The screenshot shows the Xcode interface for editing a Core Data model. The menu is open, and the 'Create NSManagedObject Subclass...' option is highlighted. A callout box points to this option with the text: 'Ask Xcode to generate NSManagedObject subclasses for our Entities.'

The model diagram shows two entities: 'Movie' and 'Genre'. The 'Movie' entity has attributes: duration, posterData, rating, synopsis, title, year, and relationships: whatKind. The 'Genre' entity has attributes: name and relationships: movies. A relationship arrow points from 'Genre' to 'Movie'.

The right sidebar shows the 'Entity' inspector for the selected 'Genre' entity. It displays the following settings:

- Name: Multiple Values
- Class: NSManagedObject
- Abstract Entity
- Parent Entity: (empty)

The bottom of the screen shows the 'Object Library' with various UI components.

Core Data

The screenshot shows the Xcode interface for creating a new Core Data entity. The 'Entity' inspector on the right shows the entity name 'Multiple Values' and class 'NSObject'. The 'Options' dialog is open, with the 'Use scalar properties for primitive data types' checkbox checked. The 'Group' is set to 'CoreDataDemo' and the 'Targets' list includes 'CoreDataDemo'. A teal callout box points to the 'Group' dropdown, and another points to the 'Use scalar properties...' checkbox.

Pick where you want your new classes to be stored (default is often one directory level higher, so watch out).

This will make your `@properties` be scalars (e.g. `int` instead of `NSNumber` *) where possible. Be careful if one of your Attributes is an `NSDate`, you will end up with an `NSTimeInterval` @property.

Core Data

Here are the two classes that were generated:
Movie.h/Movie.m and Genre.h/Genre.m

The screenshot displays the Xcode IDE interface for a project named 'CoreDataDemo'. The top menu bar includes 'Xcode', 'File', 'Edit', 'View', 'Navigate', 'Editor', 'Product', 'Window', and 'Help'. The status bar at the top right shows the time as 'Tue 14:52' and the user as 'Radu-Tudor Ionescu'. The main workspace is divided into several panels:

- Left Panel (Project Navigator):** Shows the project structure for 'CoreDataDemo'. It includes files like 'Genre.h', 'Genre.m', 'Movie.h', 'Movie.m', 'AppDelegate.h', 'AppDelegate.m', 'MainStoryboard.storyboard', 'ViewController.h', and 'ViewController.m'. The 'CoreDataDemo.xcdatamodeld' folder is expanded, showing the 'Entities' section with 'Genre' and 'Movie' entities.
- Center Panel (Entity Inspector):** Displays the 'Entities' section with 'Genre' and 'Movie' entities. Below it, there are sections for 'Fetch Requests' and 'Configurations'.
- Right Panel (Diagram):** Shows a diagram of the Core Data model. The 'Movie' entity is connected to the 'Genre' entity. The 'Movie' entity has attributes: 'duration', 'posterData', 'rating', 'synopsis', 'title', and 'year'. It also has a relationship named 'whatKind'. The 'Genre' entity has attributes: 'name' and a relationship named 'movies'.
- Bottom Panel (Object Library):** Shows a grid of UI components for adding to the interface.

The 'Movie' entity diagram shows the following details:

- Attributes:** duration, posterData, rating, synopsis, title, year
- Relationships:** whatKind

The 'Genre' entity diagram shows the following details:

- Attributes:** name
- Relationships:** movies

The diagram shows a bidirectional relationship between 'Movie' and 'Genre'.

Core Data

The screenshot shows the Xcode IDE with the CoreDataDemo project open. The left sidebar shows the project structure with Genre.h selected. The main editor displays the code for Genre.h, which includes two interfaces: Genre (NSManagedObject) and Genre (CoreDataGeneratedAccessors). The code defines properties for name and movies, and methods for adding and removing movies. Three callout boxes provide explanations: one for the @property annotations, one for the convenience methods, and one for the mutableCopy approach.

```
1 //
2 // Genre.h
3 // CoreDataDemo
4 //
5 // Created by Radu-Tudor Ionescu on 5/8/12.
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7 //
8
9 #import <Foundation/Foundation.h>
10 #import <CoreData/CoreData.h>
11
12 @class Movie;
13
14 @interface Genre : NSManagedObject
15
16 @property (nonatomic, retain) NSString * name;
17 @property (nonatomic, retain) NSSet *movies;
18 @end
19
20 @interface Genre (CoreDataGeneratedAccessors)
21
22 - (void)addMoviesObject:(Movie *)value;
23 - (void)removeMoviesObject:(Movie *)value;
24 - (void)addMovies:(NSSet *)values;
25 - (void)removeMovies:(NSSet *)values;
26
27 @end
28
```

We have @properties for all of Genre's Attributes and Relationships. That's great!

These convenience methods are for putting Movie objects in and out of the movies Attribute.

But you can also just make a mutableCopy of the movies @property (creating an NSMutableSet) and modify it. Then put it back by setting the movies @property.

Core Data

The screenshot shows the Xcode IDE interface for a project named 'CoreDataDemo'. The left sidebar displays the project structure, including files like 'Genre.h', 'Genre.m', 'Movie.h', 'Movie.m', 'AppDelegate.h', 'AppDelegate.m', 'MainStoryboard.storyboard', 'ViewController.h', and 'ViewController.m'. The main editor window shows the code for 'Movie.h'. The code includes comments, imports for 'Foundation/Foundation.h' and 'CoreData/CoreData.h', and an interface for 'Movie' that inherits from 'NSObject' and conforms to 'NSManagedObject'. The properties are:

```
1 //  
2 // Movie.h  
3 // CoreDataDemo  
4 //  
5 // Created by Radu-Tudor Ionescu on 5/8/12.  
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.  
7 //  
8  
9 #import <Foundation/Foundation.h>  
10 #import <CoreData/CoreData.h>  
11  
12  
13 @interface Movie : NSObject  
14  
15 @property (nonatomic, retain) NSNumber * duration;  
16 @property (nonatomic, retain) NSData * postData;  
17 @property (nonatomic, retain) NSNumber * rating;  
18 @property (nonatomic, retain) NSString * synopsis;  
19 @property (nonatomic, retain) NSString * title;  
20 @property (nonatomic, retain) NSNumber * year;  
21 @property (nonatomic, retain) NSManagedObject *whatKind;  
22  
23 @end  
24
```

A callout box points to the `@property (nonatomic, retain) NSManagedObject *whatKind;` line, stating: "It seems that Xcode did not generate the proper class here for the whatKind @property. It should have been a Movie *."

On the right side of the interface, there is a 'Quick Help' panel with a 'No Quick Help' button, and an 'Object Library' panel at the bottom right showing various UI components.

Core Data

The screenshot shows the Xcode interface for a project named 'CoreDataDemo'. The 'Editor' menu is open, showing options like 'Add Entity', 'Add Fetch Request', 'Add Configuration', 'Add Attribute', 'Add Fetched Property', 'Add Relationship', 'Create NSManagedObject Subclass...', 'Add Model Version...', and 'Import...'. The 'Genre' entity is selected in the 'Entity Inspector' on the right, showing its attributes and relationships. The 'Movie' entity is also visible in the diagram, with a relationship to 'Genre' named 'movies'. A callout box points to the 'Create NSManagedObject Subclass...' option, stating: 'Easy fix. Just generate the classes again. Clearly there is an "order of generation" problem (Movie was generated before Genre was).'

Canvas

- Add Entity
- Add Fetch Request
- Add Configuration
- Add Attribute
- Add Fetched Property
- Add Relationship
- Create NSManagedObject Subclass...
- Add Model Version...
- Import...

Entity Inspector:

- Entity: Genre
 - Name: Multiple Values
 - Attributes: (empty)
 - Relationships: (empty)
- User Info: (empty)
- Versioning: Hash Modifier: Version Hash Modifier, Renaming ID: Renaming Identifier
- Entity Sync: (empty)

Diagram:

```
graph LR; Movie[Movie] -- movies --> Genre[Genre];
```

Movie Entity:

- Attributes: duration, postData, rating, synopsis, title, year
- Relationships: whatKind

Genre Entity:

- Attributes: name
- Relationships: movies

Callout Box:

Easy fix. Just generate the classes again. Clearly there is an "order of generation" problem (Movie was generated before Genre was).

Core Data

The following files already exist and will be replaced:

- /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo/Movie.h
- /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo/Movie.m
- /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo/Genre.h
- /Users/raduionescu/iPhone/iOSLab/Apps/CoreDataDemo/CoreDataDemo/Genre.m

Replace Cancel

Click here to replace the old Movie.h/Movie.m and Genre.h/Genre.m files with the new ones.

Options Use scalar properties for primitive data types
Group CoreDataDemo
Targets CoreDataDemo

New Folder Cancel Create

Entity Name: Multiple Values
Class: Multiple Values
Abstract Entity:
Parent Entity:
Indexes:
User Info:
Versioning: Hash Modifier: Version Hash Modifier, Renaming ID: Renaming Identifier
Entity Sync:
Object Library:
Outline Style Add Entity Add Attribute Editor Style

Core Data

The screenshot shows the Xcode IDE with the following components:

- Top Bar:** Xcode menu, File, Edit, View, Navigate, Editor, Product, Window, Help. System status: (Charged), Tue 15:19, Radu-Tudor Ionescu.
- Toolbar:** Run, Stop, Scheme (CoreDataDemo > iOS Device), Breakpoints, Xcode.
- Left Panel (Project Navigator):** CoreDataDemo project structure including CoreDataD...cdatamodeld, Movie.h, Genre.h, AppDelegate.h, ViewController.h, Supporting Files, Frameworks, and Products.
- Center Panel (Editor):** Contents of Movie.h:

```
1 //
2 // Movie.h
3 // CoreDataDemo
4 //
5 // Created by Radu-Tudor Ionescu on 5/8/12.
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7 //
8
9 #import <Foundation/Foundation.h>
10 #import <CoreData/CoreData.h>
11
12 @class Genre;
13
14 @interface Movie : NSObject
15
16 @property (nonatomic, retain) NSNumber * duration;
17 @property (nonatomic, retain) NSData * postData;
18 @property (nonatomic, retain) NSNumber * rating;
19 @property (nonatomic, retain) NSString * synopsis;
20 @property (nonatomic, retain) NSString * title;
21 @property (nonatomic, retain) NSNumber * year;
22 @property (nonatomic, retain) Genre *whatKind;
23
24 @end
25
```
- Right Panel (Quick Help):** No Quick Help.
- Bottom Panel (Object Library):** UI components like buttons, sliders, and text fields.

Callout Box: Now this is correct. Note that you should regenerate these NSObject subclasses any time you change your schema.

Core Data

Now let's look at the Movie implementation file.

```
1 //
2 // Movie.m
3 // CoreDataDemo
4 //
5 // Created by Radu-Tudor Ionescu on 5/8/12.
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7 //
8
9 #import "Movie.h"
10 #import "Genre.h"
11
12 @implementation Movie
13
14
15 @dynamic duration;
16 @dynamic posterData;
17 @dynamic rating;
18 @dynamic synopsis;
19 @dynamic title;
20 @dynamic year;
21 @dynamic whatKind;
22
23 @end
24
```

What does @dynamic mean? It means "I do not implement the setter or getter for this property, but send me the message anyway and I will use the Objective-C runtime to figure out what to do". There is a mechanism in the Objective-C runtime to trap a message sent to you that you don't implement. NSObject does this and calls valueForKey: or setValue:forKey:. Pretty cool!

Core Data

So how do I access my Attributes with dot notation?

- Here are some examples:

```
Movie *movie = [NSEntityDescription  
                insertNewObjectForEntityForName:@"Movie"  
                inManagedObjectContext:context];
```

```
NSData *posterData = movie.posterData;  
UIImage *posterImage = [UIImage  
                        imageDataWithData:posterData];
```

```
movie.whatKind = ...;  
// a Genre object we created or got by querying
```

```
movie.whatKind.name = @"Comedy";  
// multiple dots will follow relationships
```

Core Data

What if I want to add code to my `NSManagedObject` subclass?

- That's a problem because you might want to modify your schema and re-generate the subclasses!
- But it would be really cool to be able to add code (very object-oriented).
- Especially code to create an object and set it up properly.
- Or maybe to derive new `@property`s based on ones in the database (for example, a `UIImage` based on a URL in the database).
- Time to introduce an Objective-C feature called **Categories**.

Categories

Categories are an Objective-C syntax for adding code to a class

- Without subclassing it.
- Without even having to have access to the code of the class (for example, its .m file).

Examples

- NSString's `drawAtPoint:withFont:` method.

This method is added by UIKit (since it's a UI method) even though NSString is in Foundation.

- NSIndexPath's `row` and `section` properties (used in UITableView-related code) are added by UIKit too, even though NSIndexPath is also in Foundation.

Categories

Syntax

- Example: Adding the AddOn category to Movie.

```
@interface Movie (AddOn)
- (UIImage *)posterImage;
@property (readonly) BOOL isRecommended;
@end
```

- Categories have their own .h and .m files. They are usually named like this: ClassName+PurposeOfExtension.[mh].
- Categories cannot have instance variables, so **no** @synthesize allowed in its implementation.

Categories

Implementation

```
@implementation Movie (AddOn)
- (UIImage*)posterImage // is not in the database
{
    return [UIImage imageData:self.posterData];
}
- (BOOL)isRecommended // based on rating and year
{
    NSDateFormatter *df = [[NSDateFormatter alloc] init];
    df.dateFormat = @"yyyy";
    NSString *year = [df stringFromDate:[NSDate date]];
    return [self.rating floatValue] > 7.5
        && [self.year intValue] >= [year intValue] - 1;
}
@end
```

- Sometimes we add `@property`s to an `NSObject` subclass via categories to make accessing `BOOL` attributes (which are `NSNumber`s) cleaner. Or we add `@property`s to convert `NSData` objects to whatever the bits represent.

Categories

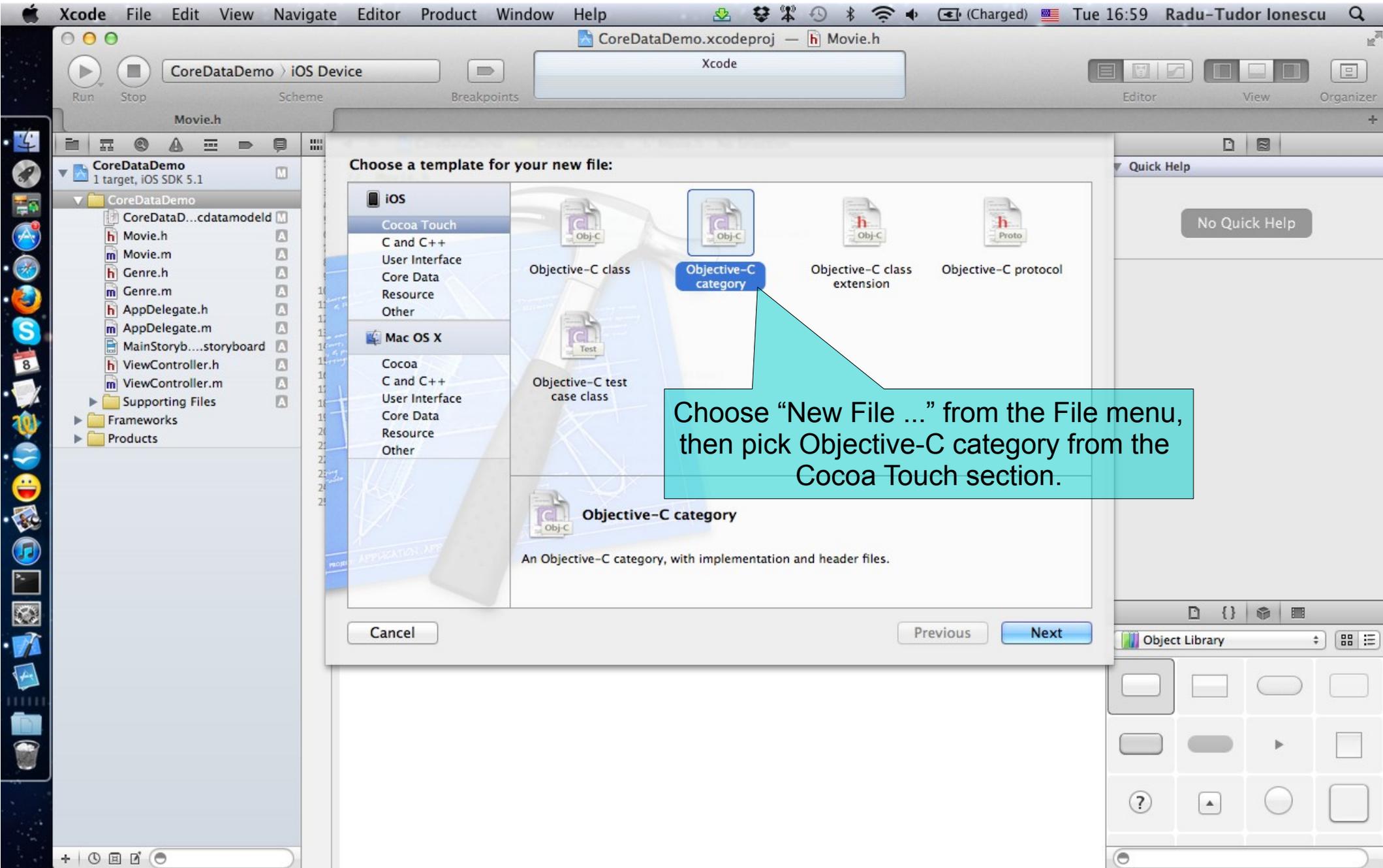
Most common category on an `NSManagedObject` subclass? Creation

```
@implementation Movie (Create)
+ (Movie *)movieWithData:(NSDictionary *)movieData
  inManagedObjectContext:(NSManagedObjectContext *)context
{
    Movie *movie = ...;
    /* See if a Movie for that data is already in the
     * database. We don't know how to query yet. */
    if (!movie)
    {
        movie = [NSEntityDescription
                 insertNewObjectForEntityForName:@"Movie"
                 inManagedObjectContext:context];

        /* Initialize the movie from the movieData.
         * Perhaps even create other database objects. */
    }
    return movie;
}
@end
```

- Any class can have a category added to it, but don't overuse or abuse this mechanism.

Core Data



Core Data

Choose options for your new file:

Category

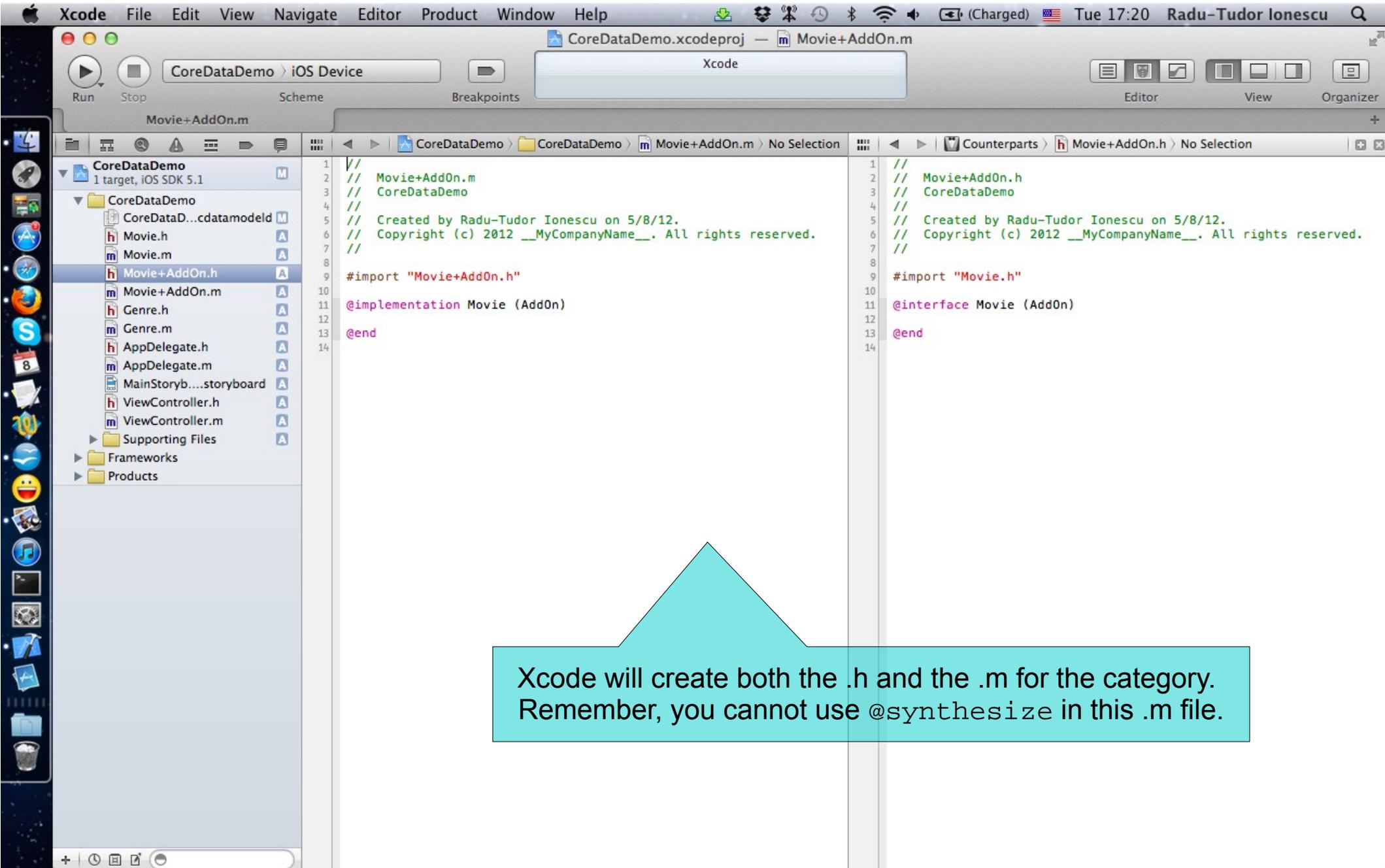
Category on

Which class to add a category to

Cancel Previous Next

Enter the name of the category, as well as the name of the class the category's methods will be added to.

Core Data



The screenshot shows the Xcode IDE with a project named 'CoreDataDemo'. The left sidebar displays the project structure, including a category named 'Movie+AddOn'. The main editor area shows the implementation of the category in 'Movie+AddOn.m'. The code includes an import statement for 'Movie+AddOn.h', an implementation block for the 'Movie' category, and an '@end' statement. The right sidebar shows the counterpart header file 'Movie+AddOn.h', which includes an interface block for the 'Movie' category and an '@end' statement. A teal callout box at the bottom of the screen contains the following text:

Xcode will create both the .h and the .m for the category. Remember, you cannot use @synthesize in this .m file.

Core Data

The screenshot shows the Xcode IDE with the CoreDataDemo project open. The project navigator on the left shows the file structure, with Movie+AddOn.m selected. The toolbar at the top shows the Run button and the Xcode interface. The left editor displays the implementation of the Movie (AddOn) category, and the right editor displays the Movie+AddOn.h header file. A callout box points to the implementation code.

```
1 //
2 // Movie+AddOn.m
3 // CoreDataDemo
4 //
5 // Created by Radu-Tudor Ionescu on 5/8/12.
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7 //
8
9 #import "Movie+AddOn.h"
10
11 @implementation Movie (AddOn)
12
13 - (UIImage*)posterImage
14 {
15     return [UIImage imageWithData:self.posterData];
16 }
17
18 - (BOOL)isRecommended
19 {
20     NSDateFormatter *df = [[NSDateFormatter alloc] init];
21     df.dateFormat = @"yyyy";
22     int year = [[df stringFromDate:[NSDate date]] intValue];
23     return [self.rating floatValue] > 7.0 && [self.year intValue] >= year - 1;
24 }
25
26 @end
27
```

```
1 //
2 // Movie+AddOn.h
3 // CoreDataDemo
4 //
5 // Created by Radu-Tudor Ionescu on 5/8/12.
6 // Copyright (c) 2012 __MyCompanyName__. All rights reserved.
7 //
8
9 #import "Movie.h"
10
11 @interface Movie (AddOn)
12
13 - (UIImage *)posterImage;
14
15 @property (readonly) BOOL isRecommended;
16
17 @end
18
```

Finally, add implementation to the AddOn category.

Deletion

- Deleting objects from the database is easy:

```
[self.doc.managedObjectContext deleteObject:movie];
```

- Make sure that the rest of your objects in the database are in a sensible state after this.
- Relationships will be updated for you (if you set Delete Rule for relationship attributes properly).
- And don't keep any `strong` pointers to `movie` after you delete it!
- Here is another method we sometimes put in a category of an `NSManagedObject` subclass:

```
@implementation Movie (Deletion)
- (void)prepareForDeletion
{
    self.whatKind.movieCount--;
}
@end
```

We don't need to set our `whatKind` to `nil` or anything here (that will happen automatically). But if `Genre` had a "number of movies" attribute, we might adjust it down by one here.

Core Data

What do you know so far?

- **Create** objects in the database with:

`insertNewObjectForEntityForName:inManagedObjectContext:`

- **Get** or **set** properties with `valueForKey:` or `setValue:forKey:`.
Or using `@property`s in a custom subclass.
- **Delete** objects in the database using the `deleteObject:` method of the `NSManagedObjectContext`.

Core Data

One very important thing left to know how to do: **Query**

- Basically you need to be able to retrieve objects from the database, not just create new ones.
- You do this by executing an `NSFetchRequest` in your `NSManagedObjectContext`.
- Four important things involved in creating an `NSFetchRequest`:
 1. Entity to fetch (required).
 2. `NSPredicate` specifying which of those Entities to fetch (optional, default is all of them).
 3. `NSSortDescriptors` to specify the order in which fetched objects are returned.
 4. How many objects to fetch at a time and/or maximum to fetch (optional, default is all of them).

Querying

Creating an NSFetchRequest

- We will consider each of these lines of code one by one:

```
NSFetchRequest *request = [NSFetchRequest  
                           fetchRequestWithEntityName:@"Movie"];  
request.fetchBatchSize = 20;  
request.fetchLimit = 100;  
request.predicate = ...;  
request.sortDescriptors = [NSArray  
                           arrayWithObject:sortDescriptor];
```

Specifying the kind of Entity we want to fetch

- A given fetch returns objects all of the same Entity. You can't have a fetch that returns some Movies and some Genres (one or the other).

Setting fetch sizes/limits

- If you created a fetch that would match 500 objects, the request above faults 20 at a time. And it would stop fetching after it had fetched 100 of the 500.

NSSortDescriptor

- When we execute a fetch request, it's going to return an NSArray of NSManagedObjects.
- NSArrays are ordered, so we have to specify the order when we fetch.
- We do that by giving the fetch request a list of “sort descriptors” that describe what to sort by:

```
NSSortDescriptor *sortDescriptor =  
    [NSSortDescriptor sortDescriptorWithKey:@"title"  
                                     ascending:YES  
                                     selector:@selector(caseInsensitiveCompare:)];
```

- There's another version with no selector: argument (default is the method compare:). The selector: argument is just a method sent to each object to compare it to others.
- Some of these “methods” might happen on the database side.
- We give a list of these to the NSFetchRequest because sometimes we want to sort first by one key, then sort by another (e.g. lastName, firstName).

NSPredicate

NSPredicate

- You use predicates to represent logical conditions.
- This is the basis of how we specify exactly which objects we want from the database.

Predicate formats

- Creating one looks a lot like creating an NSString, but the contents have semantic meaning.
- Example:

```
NSString *series = @"Harry Potter";
```

```
NSPredicate *predicate = [NSPredicate  
predicateWithFormat:@"title contains %@", series];
```

NSPredicate

Other examples

- Unique movie in the database:

```
@"uniqueId = %@", [movieData objectForKey:@"id"]
```

- Matches title case insensitively:

```
@"title contains[c] %@", (NSString *)
```

- If we had the Date of the release of a Movie in the data mapping:

```
@"releaseDate > %@", [NSDate date]
```

- Movie search by Genre:

```
@"whatKind.name = %@", (NSString *)
```

- Genre search (not a Movie search here):

```
@"any movies.title contains %@", (NSString *)
```

- Many more options. Look at the NSPredicate class documentation.

NSPredicate

Combined predicates

- You can use AND and OR inside a predicate string:

```
@"(year = %@) OR (title = %@)" // same with ||
```

```
@"(year = %@) && (title = %@)" // same with AND
```

- Or you can use the alternative to combine NSPredicate objects with special NSCompoundPredicates:

```
NSArray *array = [NSArray arrayWithObjects:  
    predicate1,  
    predicate2, nil];
```

```
NSPredicate *predicate = [NSCompoundPredicate  
    andPredicateWithSubpredicates:array];
```

- This predicate is “predicate1 AND predicate2”.
- OR predicate also available, of course.

Querying

Putting it all together

- Let's say we want to query for all Genres:

```
NSFetchRequest *request = [NSFetchRequest  
    fetchRequestWithEntityName:@"Genre"];
```

- That have movies with a rating greater than 8:

```
request.predicate = [NSPredicate  
    predicateWithFormat:@"any movies.rating > %@", 8];
```

- Sorted by the Genre's name:

```
NSSortDescriptor *sortByName =  
    [NSSortDescriptor sortDescriptorWithKey:@"name"  
        ascending:YES];  
request.sortDescriptors =  
    [NSArray arrayWithObject:sortByName];
```

Querying

Executing the fetch

- Use the `executeFetchRequest:` method:

```
NSManagedObjectContext *managedObjectContext =  
    self.doc.managedObjectContext;  
  
NSError *error;  
NSArray *genres =  
    [managedObjectContext executeFetchRequest:request  
                             error:&error];
```

- Returns `nil` if there is an error (check the `NSError` for details).
- Returns an empty array (not `nil`) if there are no matches in the database.
- Returns an array of `NSManagedObjects` (or subclasses thereof) if there were any matches.
- You can pass `NULL` for `error:` if you don't care why it fails.

Querying

Faulting

- The above fetch does not necessarily fetch any actual data.
- It could be an array of “as yet unfaulted” objects, waiting for you to access their attributes.
- Core Data is very smart about “faulting” the data in as it is actually accessed.
- For example, if you did something like this:

```
for (Genre *genre in genres)
{
    NSLog(@"fetched genre %@", genre);
}
```

You may or may not see the names of the genres in the output

(you might just see “unfaulted object”, depending on whether it prefetched them).

Faulting

Faulting

- But if you did this:

```
for (Genre *genre in genres)
{
    NSLog(@"fetched genre named %@", genre.name);
}
```

Then you would definitely fault all the Genres in from the database.

There is so much more (that we don't have time to talk about)

- Optimistic locking (`deleteConflictsForObject:`).
- Rolling back unsaved changes.
- Undo and redo changes.

What should you study next?

- Modal View Controllers
- Core Motion (gyro, accelerometer, magnetometer)
Measuring the device's movement.
- UITextField, UITextView, UIActionSheet
- UIView Animation
- UIImagePickerController
Getting images from the camera or photo library.
- NSTimer
Perform scheduled tasks on the main thread.
- iPad and Universal Applications
There are specific Navigation and View Controllers.
- Open GL ES