



COMP327

Mobile Computing

Session: 2014-2015

Lecture Set 4 - Data Persistence, Core Data and Concurrency

In these Slides...

- **We will cover...**
 - An introduction to Local Data Storage
 - The iPhone directory system
 - Property lists
 - Data Modelling using Core Data
 - User Defaults

Storing Data

These slides will allow you to understand how data can be modelled and stored/cached locally. This may be for a local database, or simply saving state or preferences for an application.

Local Data Storage

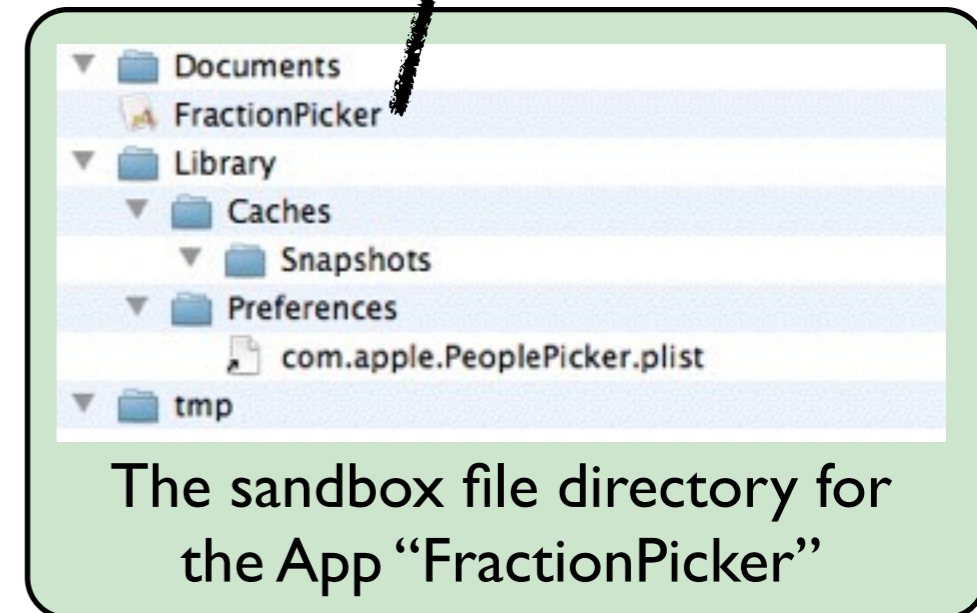
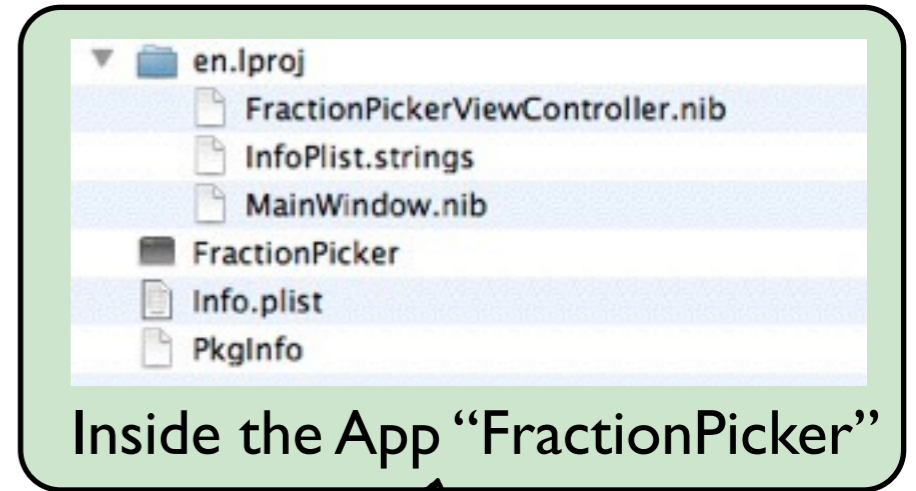
Data Persistence, Core Data and
Concurrency

Intro to data and persistence

- There may be a number of reasons for wanting to read or write data to some form of persistent store
 - Storing preferences
 - Storing simple data in general
 - Storing state
 - Simple values or container classes
 - Serialising custom object data
 - Managing data
 - SQLite
 - Core Data

iPhone File System

- Each application sandboxes its file system
 - Ensures security and privacy
 - Apps can't interfere with other app's data
 - When apps are deleted, all the associated files are deleted
- Each app maintains its own set of directories
 - somewhat reminiscent to a UNIX filestore
 - Files stored in Caches are not backed up during iTunes sync
- Apps cannot write into their own bundles
 - This violates the code-signing mechanism
 - If apps want to include data in the bundle that can later be modified
 - it must copy the data into your documents directory
 - then the data can be modified!
 - Remember to do this only once



File Paths in your Application

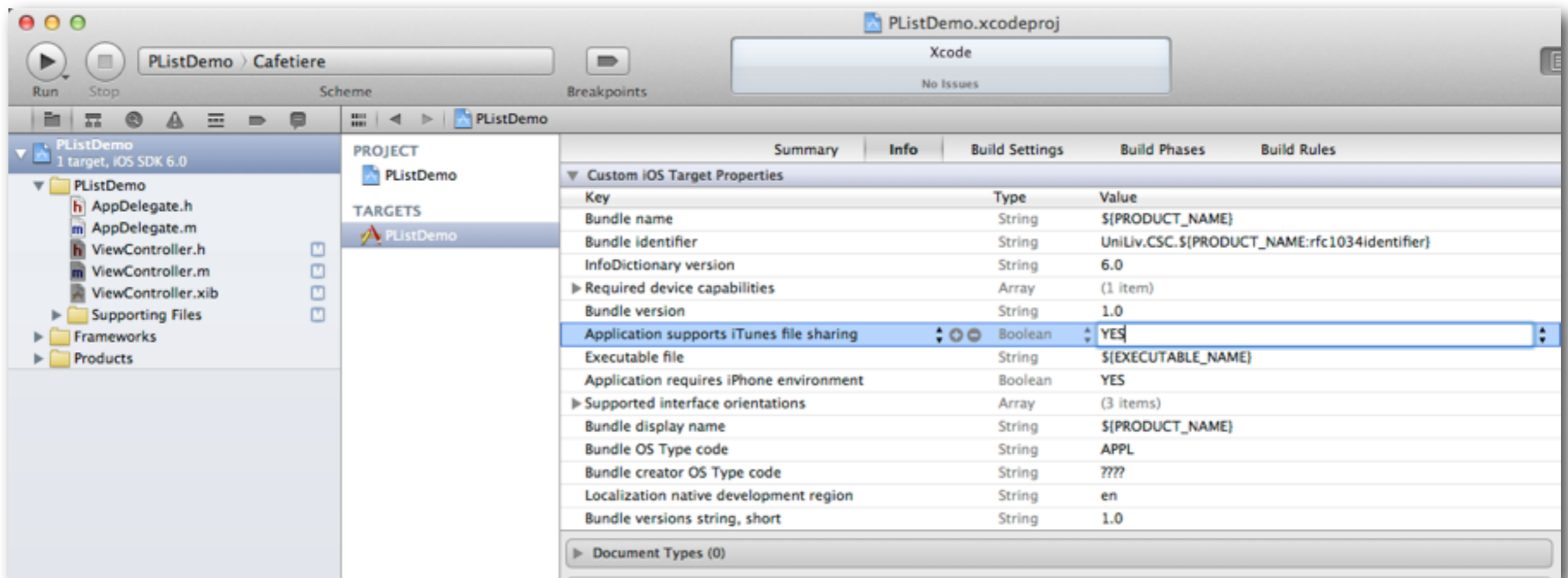
- Getting to the Applications home directory
 - NSHomeDirectory
- Finding directories in the file system
 - NSSearchPathForDirectoriesInDomains
 - Creates an array of path strings for the specified directories in the specified domains
 - Good for discovering where a “known” directory is in the file system

```
// Documents directory
NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsPath = [paths objectAtIndex:0];

// <Application Home>/Documents/myData.plist
NSString *myDataPath = [documentsPath stringByAppendingPathComponent:@"myData.plist"];
```


iTunes File sharing

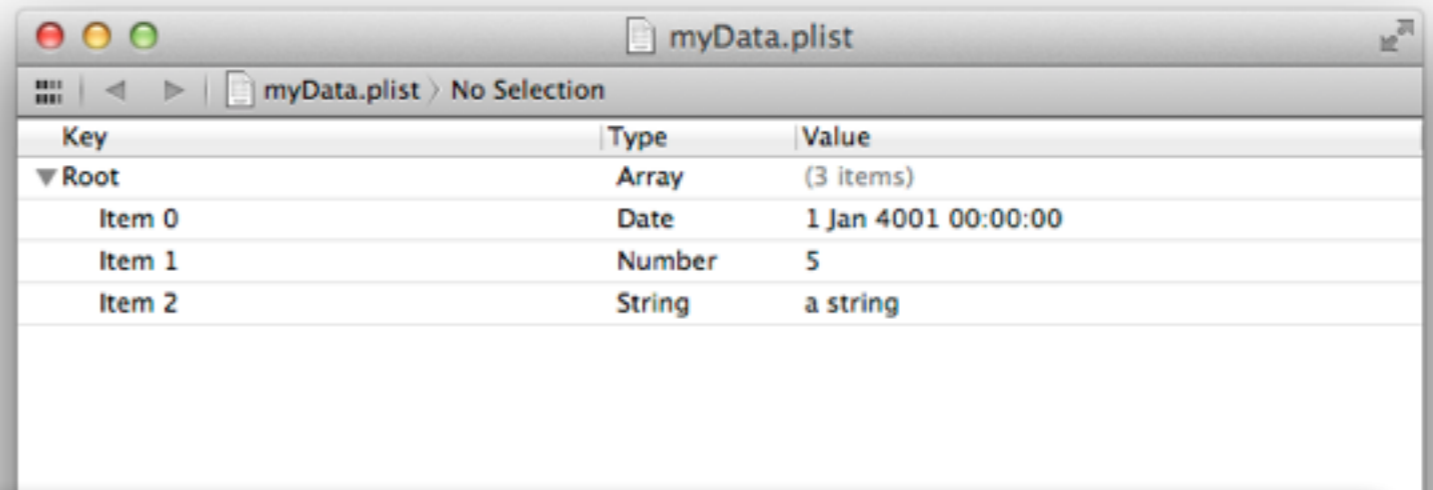
- Applications can expose files in their Documents directory to iTunes (when syncing)
 - Allows users to add and delete files directly to an application
- Capability is set by setting the **Application supports iTunes file sharing** Target Property to YES



iTunes File sharing



iTunes File sharing



Key	Type	Value
▼ Root	Array	(3 items)
Item 0	Date	1 Jan 4001 00:00:00
Item 1	Number	5
Item 2	String	a string

```
- (IBAction)saveAction:(id)sender {  
  
    // Documents directory  
    NSArray *paths = NSSearchPathForDirectoriesInDomains (NSDocumentDirectory, NSUserDomainMask, YES);  
    NSString *documentsPath = [paths objectAtIndex:0];  
  
    // <Application Home>/Documents/myData.plist  
    NSString *myDataPath = [documentsPath stringByAppendingPathComponent:@"myData.plist"];  
  
    NSArray *myArray;  
    NSDate *aDate = [NSDate distantFuture];  
    NSValue *aValue = [NSNumber numberWithInt:5];  
    NSString *aString = @"a string";  
  
    myArray = [NSArray arrayWithObjects:aDate, aValue, aString, nil];  
  
    [myArray writeToFile:myDataPath atomically:YES];  
}
```

Property Lists

- Property lists are structured data used by Cocoa and Core Foundation
 - Used extensively within iOS and MacOS
 - Typically XML-based data format
- Provides support for
 - Primitive data types
 - strings (NSString)
 - numbers - integers (NSNumber:intVal)
 - numbers - floating point (NSNumber:floatVal)
 - binary data - (NSData)
 - dates - (NSDate)
 - boolean values - (NSNumber:boolValue == YES or NO)
 - Collections - which can recursively contain other collections
 - arrays - (NSArray)
 - dictionaries - (NSDictionary)
- Root-level object is almost always either an array or dictionary
 - Could be a single element plist containing a primitive data type

Property Lists

- A really good way to store small, structured persistent data fragments
- **Good for:**
 - less than a few hundred kilobytes of data
 - well defined data elements, that fit the XML data serialisation
- **Bad for:**
 - Large data volumes
 - Loading is “one-shot”
 - Complex objects
 - Blocks of binary data

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Root</key>
  <dict>
    <key>Name</key>
    <string>John Doe </string>
    <key>Phones</key>
    <array>
      <string>408-974-0000</string>
      <string>503-333-5555</string>
    </array>
  </dict>
</dict>
</plist>
```

Key	Type	Value
Root	Dictionary (2 items)	
Name	String	John Doe
Phones	Array (2 items)	
Item 0	String	408-974-0000
Item 1	String	503-333-5555

```
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Data" ofType:@"plist"];
NSDictionary *tmpDict = [NSDictionary alloc initWithContentsOfFile:filePath];

NSLog(@"%@", tmpDict);

NSDictionary *elemDict = [tmpDict objectForKey:@"Root"];
[nameLabel setText:[elemDict objectForKey:@"Name"]];
NSArray *phoneNums = [elemDict objectForKey:@"Phones"];

[phoneLabel1 setText:[phoneNums objectAtIndex:0]];
[phoneLabel2 setText:[phoneNums objectAtIndex:1]];

[tmpDict release];
}
```

```
2011-06-14 23:17:28.034 PropertyListDemo[6532:207] {
  Root = {
    Name = "John Doe ";
    Phones = (
      "408-974-0000",
      "503-333-5555"
    );
  };
};
```

Reading and Writing Property Lists

- Both NSArray and NSDictionary have recursive convenience methods
 - Reading from a file or URL:

```
// Reading Data.plist from the resource bundle
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"Data" ofType:@"plist"];
NSDictionary *tmpDict = [[NSDictionary alloc] initWithContentsOfFile:filePath];

// Reading WrittenArray.plist from the application's file directory
NSArray *tmpArr = [[NSArray alloc] initWithContentsOfFile:@"WrittenArray.plist"];
```

- - (id) initWithContentsOfURL:(NSURL *)aURL;
 - - (id) initWithContentsOfFile:(NSString *)aPath;
- Writing to a file or URL
 - - (BOOL)writeToFile:(NSString *)aPath atomically:(BOOL)flag;
 - - (BOOL)writeToURL:(NSURL *)aURL atomically:(BOOL)flag;
- Property lists can also be serialised from objects into a static format
 - Can be stored in the file system (in different formats) and read back later
 - Uses NSPropertyListSerialization class

Example: Writing an Array to Disk

Create an array of three items - a date, a value and a string

```
NSArray *myArray;  
NSDate *aDate = [NSDate distantFuture];  
NSNumber *aValue = [NSNumber numberWithInt:5];  
NSString *aString = @"a string";  
  
myArray = [NSArray arrayWithObjects:aDate, aValue, aString, nil];  
  
[myArray writeToFile:@"WrittenArray.plist" atomically:YES];
```

Send the message - (BOOL) writeToFile:atomically to the array

The resulting file is written in the application's file space, to be read later

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">  
<plist version="1.0">  
<array>  
  <date>4001-01-01T00:00:00Z</date>  
  <integer>5</integer>  
  <string>a string</string>  
</array>  
</plist>
```

JSON - JavaScript Object Notation

- Lightweight data interchange format with simple BNF structure
 - Similar to a property list
 - Supports:
 - Collections of name/value pairs - **objects**
 - { string : value , string : value , ... }
 - Ordered list of values - **arrays**
 - [value , value , ...]
 - Values can be
 - strings, numbers, objects, arrays, "true", "false", "null"
 - Elements can be nested
- NSJSONSerialization Class
 - Converts JSON data into Foundation classes
 - Converts Foundation classes into JSON data

See <http://json.org> for full definition

What does it look like?

```
{
  "title": "University of Liverpool Computer Science Programme List",
  "groups": [
    {
      "grouptitle": "Agents ART Group",
      "grouplabel": "agentArt",
      "members": [ "Katie", "Trevor", "Frans", "Paul D.", "Floriana", "Wiebe", "Dave J.", "Davide", "Terry", "Valli" ]
    },
    {
      "grouptitle": "Complexity Theory and Algorithmics Group",
      "grouplabel": "ctag",
      "members": [ "Leszek", "Leslie", "Irina", "Dariusz", "Russ", "Igor", "Prudence", "Michele" ]
    },
    {
      "grouptitle": "Logic and Computation Group",
      "grouplabel": "loco",
      "members": [ "Michael", "Frank", "Clare", "Ullrich", "Boris", "Alexei", "Grant", "Vladimir", "Sven" ]
    },
    {
      "grouptitle": "Economics and Computation Group",
      "grouplabel": "ecco",
      "members": [ "Xiaotie", "Paul W.", "Piotr", "Rahul", "Martin", "Mingyu" ]
    }
  ]
}
```

- The JSON structure naturally matches that of the Foundation container classes
 - NSDictionaries that contain key-value pairs
 - NSArray or NSSets that contain lists of values

How to use the NSJSONSerialization Class

- Parsing JSON into Foundation Objects

- Load the JSON data as an NSData object
- Call the method

```
+ (id)JSONObjectWithData:(NSData *)data  
                    options:(NSJSONReadingOptions)opt  
                    error:(NSError **)error
```

- Options allow the construction of mutable objects (`kNilOptions` for no options)
- NSError can report any parsing errors (method also returns `nil`)

- Creating JSON from Foundation Objects

- Creates JSON data from an NSData object
- Call the method

```
+ (NSData *)dataWithJSONObject:(id)obj  
                    options:(NSJSONWritingOptions)opt  
                    error:(NSError **)error
```

- Writing option `NSJSONWritingPrettyPrinted` includes space to make JSON data readable

Archiving Objects (Serialisation)

- A serialisation of some object graph that can be saved to disk
 - and then be loaded later
 - This is what is used by nibs
- Use the `<NSCoding>` protocol
 - Declares two methods that a class must implement so that instances can be encoded or decoded
 - Encode an object for an archiver: `encodeWithCoder`
 - Decode an object from an archive: `initWithCoder`
 - When writing to file, you either use:
 - `NSKeyedArchiver` - to serialise an object to a file
 - `NSKeyedUnarchiver` - to decode a serialisation from a file

Using Web Services

- A lot of data is “in the cloud”
 - data is available from some web server
 - data can then be sent back to a web server
 - A number of APIs available
 - Google, Flickr, Ebay, etc...
 - Typically exposed via RESTful services
 - Uses XML or JSON data formats
 - Parsing XML
 - iOS provides some XML support - e.g. NSXMLParser
 - Several open-source XML parsers available
- JSON is also used by Apple Push Notifications

User Defaults and Settings Bundles

- Often applications need to save a small number of settings
 - e.g. preferences or last used settings
- NSUserDefaults provides a “registry” for storing values
 - Generated on a per user / per application basis
 - Register the different settings
 - Normally done in the app default
 - Determines the default values and types of the settings for first use, or if reset

```
#define USERDEFAULTS_LASTSELECTEDCELL @"selected_cell"
#define USERDEFAULTS_VERSION @"version_number"

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.

    NSDictionary *appDefaults = @{USERDEFAULTS_LASTSELECTEDCELL:[NSNumber numberWithInt:0],
                                  USERDEFAULTS_VERSION:@"V1.0",
                                  };
    [[NSUserDefaults standardUserDefaults] registerDefaults:appDefaults];
    ...
}
```


User Defaults and Settings Bundles

- NSUserDefaults can be set and retrieved at any point in the application
 - Access the defaults object, and treat as a dictionary

```
// Select the correct cell
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
int selectedCellNum = [[defaults objectForKey:USERDEFAULTS_LASTSELECTEDCELL] integerValue];

...

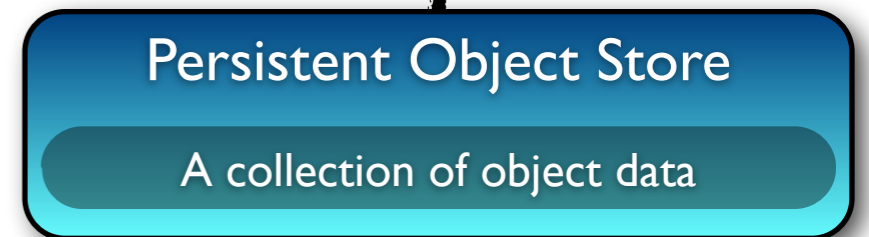
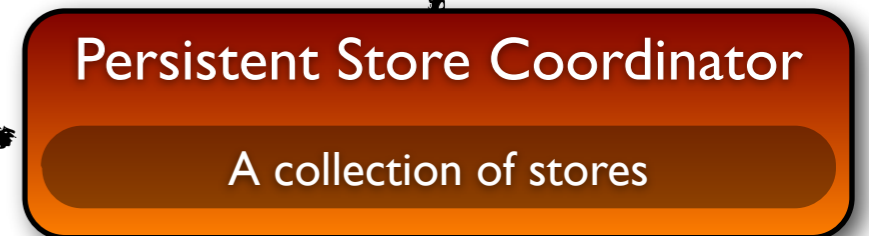
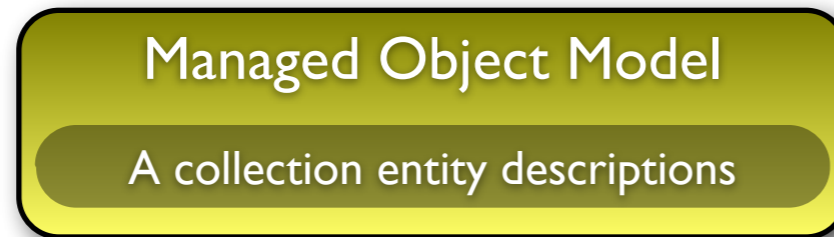
// Save setting in User Defaults
[defaults setObject:[NSNumber numberWithInt:[indexPath row]] forKey:USERDEFAULTS_LASTSELECTEDCELL];
```

- Values are cached in the application and periodically calls the method *synchronize*, to store values to the defaults database
 - As this is called periodically, you should only use it if you cannot wait - e.g. if the application is about to exit immediately after changes.

Core Data

- A schema-driven object graph management and persistence framework
 - Model objects can be saved to the file store...
 - ..and then retrieved later
- Specifically, Core Data
 - provides an infrastructure for managing all the changes to your model objects
 - Provides automatic support for undo and redo
 - supports the management of a subset of data in memory at any time
 - Good for managing memory and keeping the footprint low
 - uses a diagrammatic schema to describe your model objects and relationships
 - Supports default values and value-validation
 - maintains disjoint sets of edits on your objects
 - Can allow the user to edit some of your objects, whilst displaying them unchanged elsewhere

The Core Data Stack



- A collection of Core Data framework objects that access a persistent store
 - This could be a database, but doesn't have to be.
- Core Data allows the developer to manage data at the top of this stack
 - abstracts away the storage mechanism
 - Allows the developer to focus on
 - **Managed Objects** (i.e. records from tables in databases)
 - **Managed Object Context** (i.e. work area which manages objects from a Core Data store)

Managed Objects

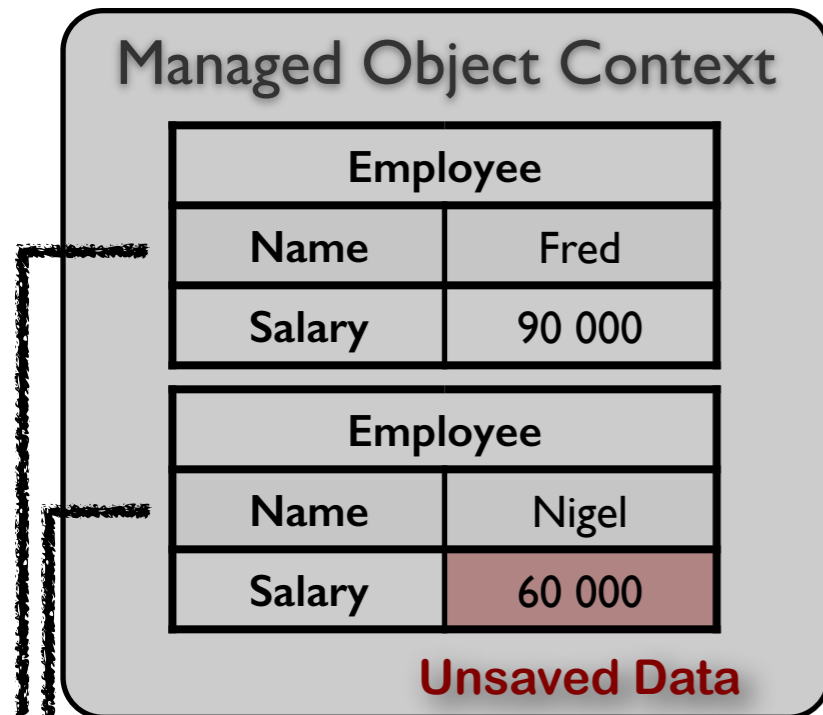
- An object representation of a record from a table in a database
 - The model object (from the MVC pattern) managed by Core Data
 - The data used within the application
 - shapes, lines, groups of elements in a drawing program
 - artists, tracks, albums in a music database
 - people and departments in a HR application
- An instance of the `NSManagedObject`
 - or a `NSManagedObject` subclass

Managed Object Contexts

- A context represents a single object space in an application
 - Responsible for managing a collection of Managed Objects
 - Managed objects form a group of related model objects that represent an internally consistent view of a data store
 - e.g. records and their relationships
 - Also responsible for:
 - the complete life-cycle management
 - validation of data
 - relationship maintenance
 - Undo and Redo of actions
- Instance of an `NSManagedObjectContext`
 - or an `NSManagedObjectContext` subclass

Managed Object Contexts

- Managed Objects exist within a **Managed Object Context**
 - New managed objects are inserted into context
 - Existing records in a database are fetched into a context as managed objects
 - Changes to records are kept in memory until they are committed to the store by saving the context
 - Includes insertion or deletion of complete objects
 - Includes manipulation of property values



Employee	
Name	Salary
Fred	90 000
Julie	97 000
Nigel	50 000
Tanya	56 000

Current Data

This managed object context contains two managed objects corresponding to two records in an external database.

Other records exist in the database, but there are no corresponding managed objects.

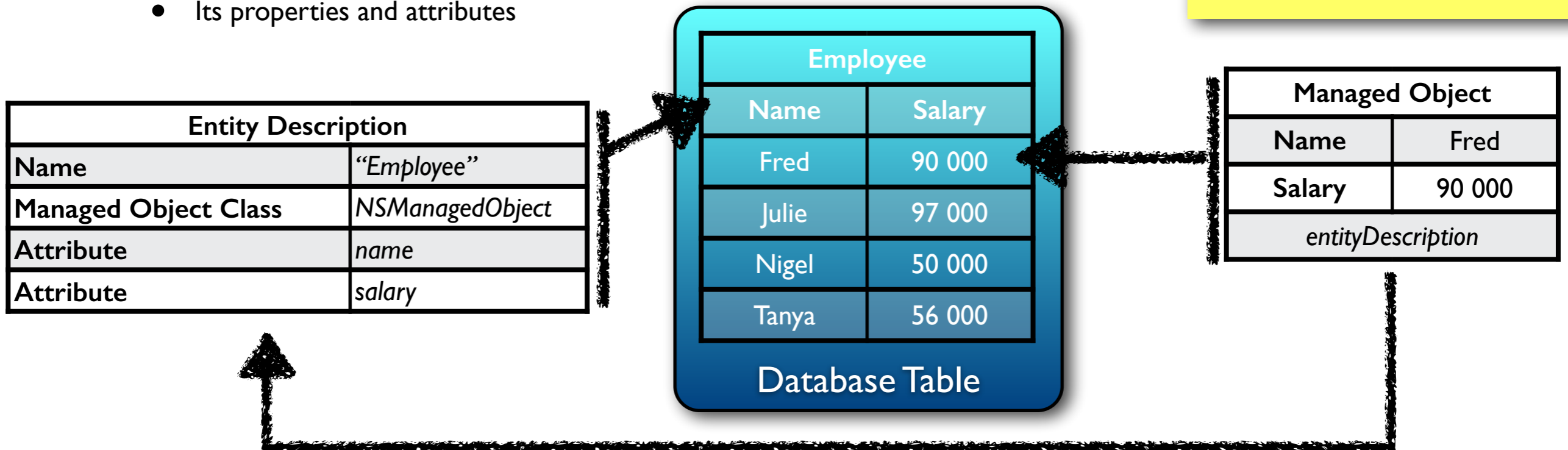
Note that Nigel's salary has been increased, but that the change has not been committed to the database.

Managed Object Model

- A Managed Object model represents a schema that describes the data (and hence the database)
 - And hence the managed objects in the application
- A model is a collection of **entity description objects**
 - Instances of `NSEntityDescription`
 - Describes an entity or table in a database, in terms of:
 - Its name
 - Name of the class used to describe the entity in the app
 - Its properties and attributes

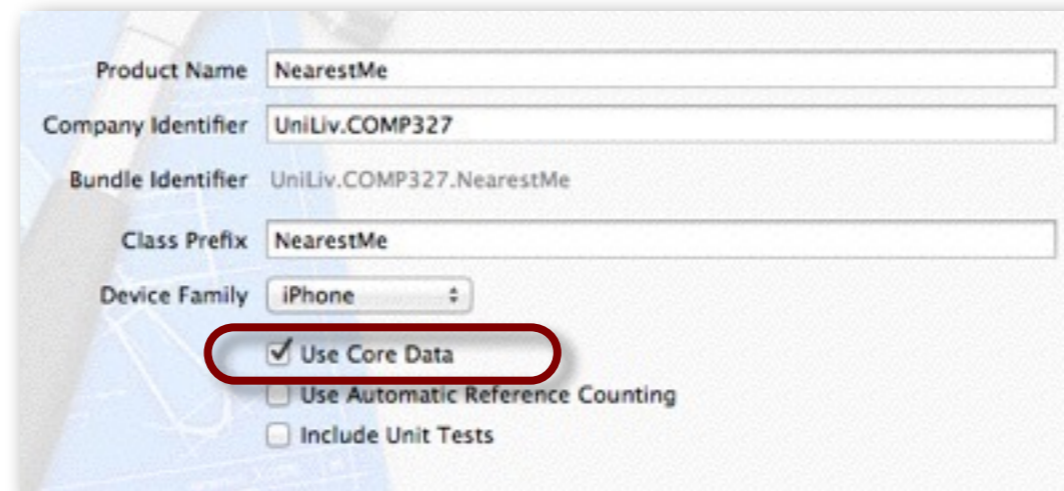
Core Data

It uses the model to map between managed objects in your app to records in the database



Using Core Data: The Basics

- Core Data support can be included in several of the project templates
 - Select “*Use Core Data*” when creating the project



- Includes additional code within the app delegate implementation file
 - Manages files in the applications Documents Directory
 - Provides a persistent store coordinator for the app
 - Returns the managed object model for the app
 - Also provides the managed object context for the app
- Also includes a Core Data Model (.xcdatamodeld) file
 - Known as the **managed object model**

Getting started with Core Data

- Any use of code data will need access to the Managed Object Context
 - Get this from the app delegate and pass to the main view controller

The mutable array stores the collection of event objects we'll load from the persistent store

```
// RootViewController.h

@interface RootViewController : UITableViewController {
    NSMutableArray *eventsArray;
    NSManagedObjectContext *managedObjectContext;
}

@property (nonatomic, retain) NSMutableArray *eventsArray;
@property (nonatomic, retain) NSManagedObjectContext *managedObjectContext;
```

This is the “gateway” to the Core Data Stack

- Ensure that the controller has an ivar and property to store this

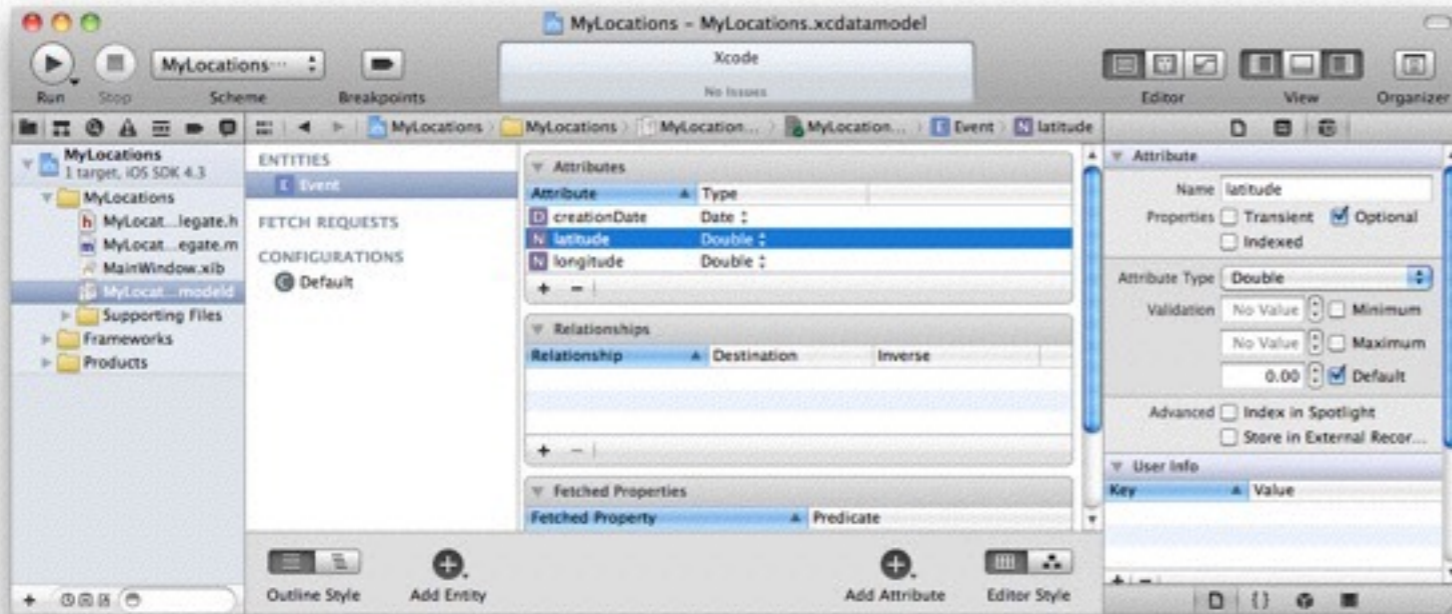
```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    RootViewController *rootViewController = [[RootViewController alloc] initWithStyle:UITableViewStylePlain];

    NSManagedObjectContext *context = [self managedObjectContext];
    if (!context) {
        // We have some error here that needs handling!!!
    }

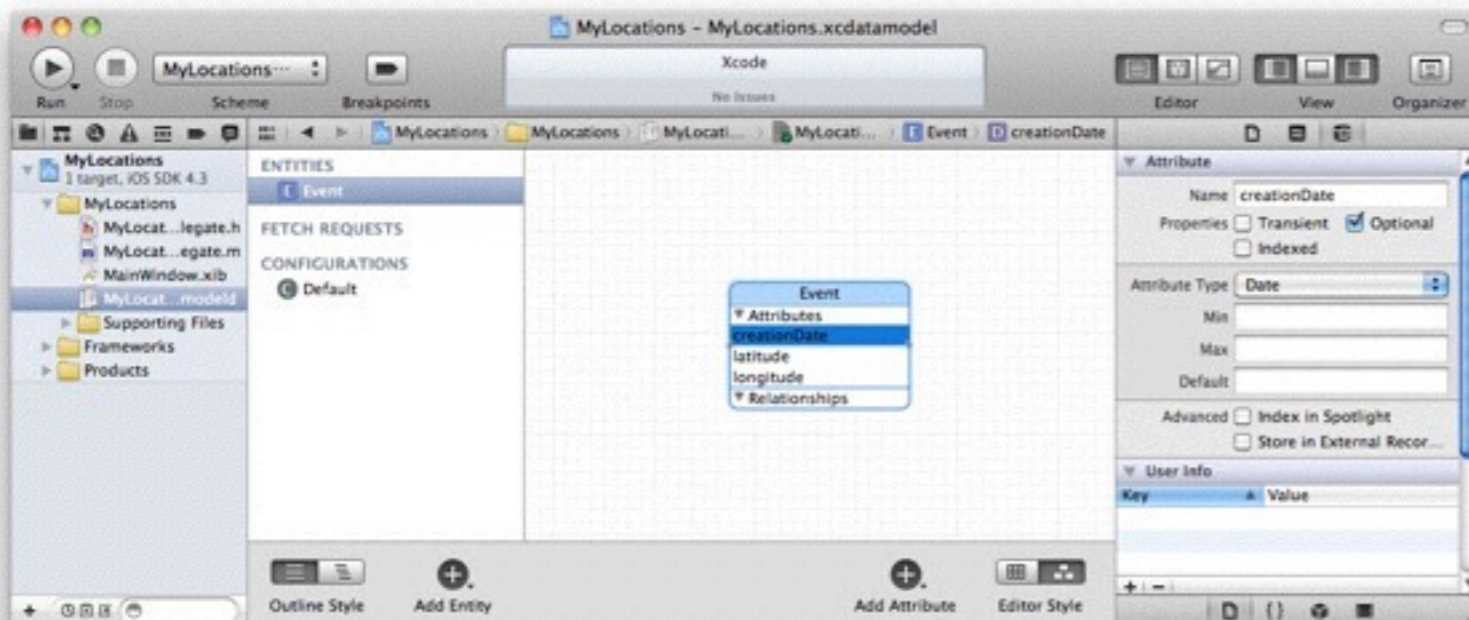
    // Pass the managed object context to the root controller
    [rootViewController setManagedObjectContext:context];

    ....
}
```

Modelling your data



- A data model is a collection of entity and property description objects that describe the Managed Objects
 - This model can be described programmatically
 - Alternatively, the graphical modelling tool can be used
 - Different editor styles can be selected
 - Entity and Attribute properties can be viewed by opening the right-most pane (the **Data Model Inspector**)



- Entities can be added to the model
 - This should be an object of type **NSObject**
 - (see later)
 - The entity name doesn't have to be the same as class that will represent instances of this object
- Attributes can then be defined
 - complete with name and type

Custom Managed Object Class

- In principle, entities can be represented using NSObject
 - Typically better to use a Custom Class
 - Provides better development tool support
 - Completion for property accessor methods, and compile-time checking
 - Easier to tailor validation methods or derived properties
- Xcode can be used to generate custom classes from the Model
 1. Select the entity in the graphical editor
 2. Create a new File of type Managed Object Class
 3. A new class for the entity is added to the project
 4. Import the new header in the view controller's .m file

```
// Event.m

#import "Event.h"

@implementation Event
@dynamic creationDate;
@dynamic latitude;
@dynamic longitude;

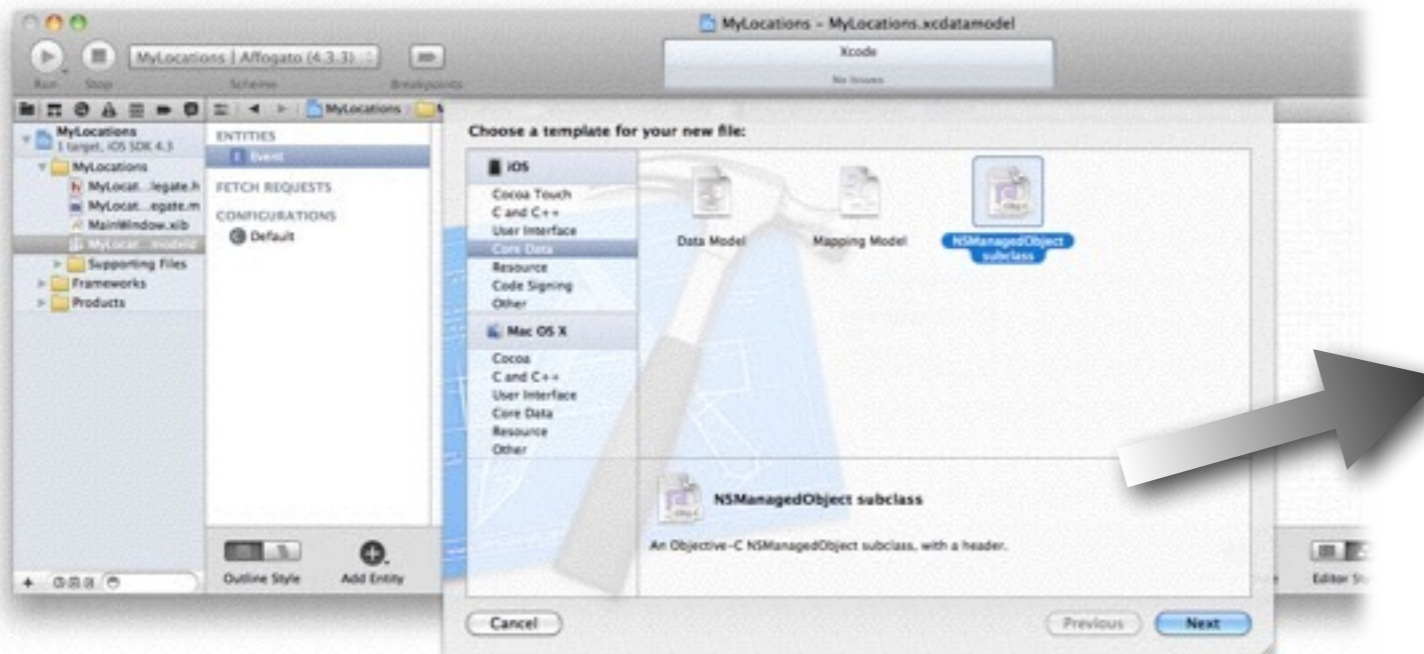
@end
```

```
// Event.h

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

@interface Event : NSObject {
@private
}
@property (nonatomic, retain) NSDate * creationDate;
@property (nonatomic, retain) NSNumber * latitude;
@property (nonatomic, retain) NSNumber * longitude;

@end
```



Custom Managed Object Class

- Things worth noting about the new class
 - Attributes are represented by objects
 - The double type has been replaced by NSNumber
 - In the implementation file, the properties are not synthesized, but are “dynamic”
 - Accessor methods are created at runtime not compile time
 - The implementation file has no dealloc method declared
 - Core data is responsible for the life-cycle of all modelled properties of a managed object
 - The editor now represents the “Event” as the new type
 - As opposed to **NSManagedObject** - see earlier

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

@interface Event : NSManagedObject {
@private
}
@property (nonatomic, retain) NSDate * creationDate;
@property (nonatomic, retain) NSNumber * latitude;
@property (nonatomic, retain) NSNumber * longitude;

@end
```

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

@implementation Event
@dynamic creationDate;
@dynamic latitude;
@dynamic longitude;

@end
```

Managing instances of the new custom managed class

- **Create and configure the “Event” object**
 - A convenience `NSEntityDescription` method returns a properly initialised instance of the specified entity
 - This is inserted in the managed object context
 - Once created, set the property values
 - Note that changes to the instance are not pushed to the persistent store until the context is saved
- **Save the context**

```
// RootViewController.m
- (void) addEvent {
    CLLocation *location = [locationManager location];

    // Create and configure a new instance of the Event entity
    Event *event = (Event *)[NSEntityDescription insertNewObjectForEntityForName:@"Event"
                                     inManagedObjectContext:managedObjectContext];

    CLLocationCoordinate2D coordinate = [location coordinate];

    [event setLatitude:[NSNumber numberWithInt:coordinate.latitude]];
    [event setLongitude:[NSNumber numberWithInt:coordinate.longitude]];
    [event setCreationDate:[NSDate date]];

    NSError *error = nil;

    if (![managedObjectContext save:&error]) {
        // We need to handle this error at some point
        // However, if we did get here, then this is probably a catastrophic error
    }
    ....
}
```

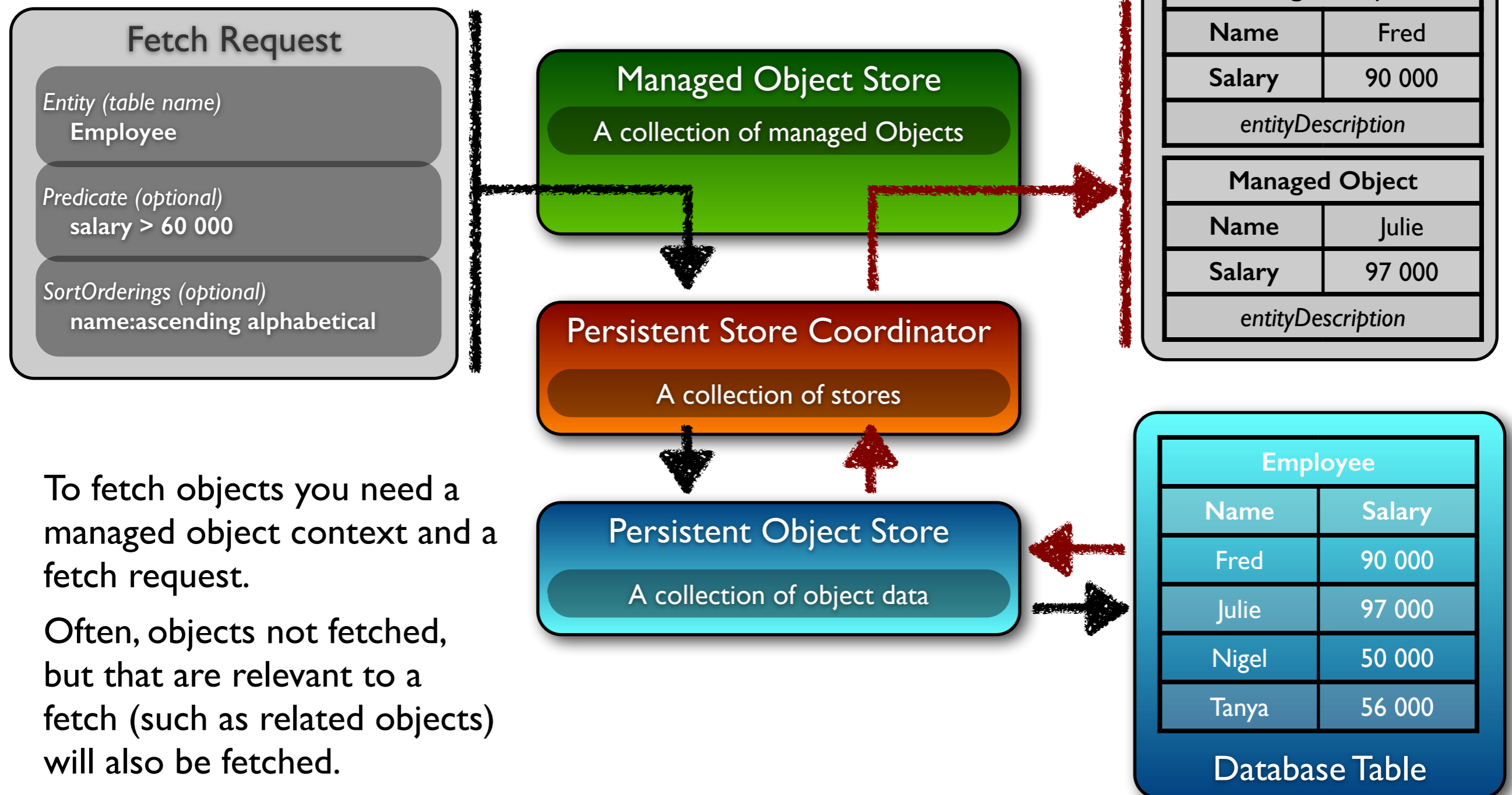
Create an instance of the new event object.

Note that we identified our object using the name from the model

Use setter methods to set the values of the object's attributes

Save the context to commit the managed objects into the store

Fetching objects from the store



To fetch objects you need a managed object context and a fetch request.

Often, objects not fetched, but that are relevant to a fetch (such as related objects) will also be fetched.

Creating and Executing a Request

Create the fetch request, and identify the entity. Provide the name of the entity (“Event”) to the managed context.

```
// RootViewController.m
- (void) viewDidLoad {
    ....
    // Create the request
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
                                inManagedObjectContext:managedObjectContext];

    [request setEntity:entity];

    // Set the sort descriptor
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"creationDate" ascending:NO];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor, nil];
    [request setSortDescriptors:sortDescriptors];
    [sortDescriptors release];
    [sortDescriptor release];

    // Execute the Request
    NSError *error = nil;
    NSMutableArray *mutableFetchResults = [[managedObjectContext executeFetchRequest:request
                                             error:&error] mutableCopy];

    ....
}
```

Set the sort descriptor; otherwise the order the objects are returned will be undefined.

As multiple sort orderings may be specified, these are given in an array.

Execute the request - in this case we store it in a mutable array, as the results may be modified within our application.

Deleting Managed Objects

In this case, we have received an edit request from the UITableView to delete an entry in the table, that corresponds to an entry we want to delete from our store.

Identify the NSManagedObject which is to be deleted.

In this case, it is held within the eventsArray, which is also used to fill in entries in the table.

```
// RootViewController.m

- (void)tableView:(UITableView *)tableView commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
    if (editingStyle == UITableViewCellEditingStyleDelete) {

        // Delete the managed object at the given index path
        NSManagedObject *eventToDelete = [eventsArray objectAtIndex:indexPath.row];
        [managedObjectContext deleteObject:eventToDelete];

        // Update the array and table view
        [eventsArray removeObjectAtIndex:indexPath.row];
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath] withRowAnimation:YES];

        // Commit the change
        NSError *error = nil;
        if (![managedObjectContext save:&error]) {
            // Handle This error somehow
        }
    }
}

....
```

In this app, we also need to delete the entry from our array, and from the table view

Save the context, to push the changes down to the persistent store!

Questions?

Data Persistence, Core Data and
Concurrency