Seneca College of Applied Arts & Technology

PLAIN HTML VERSION: lecture7.161.txt.html
Xcode Project Archive: lecture7.161.zip

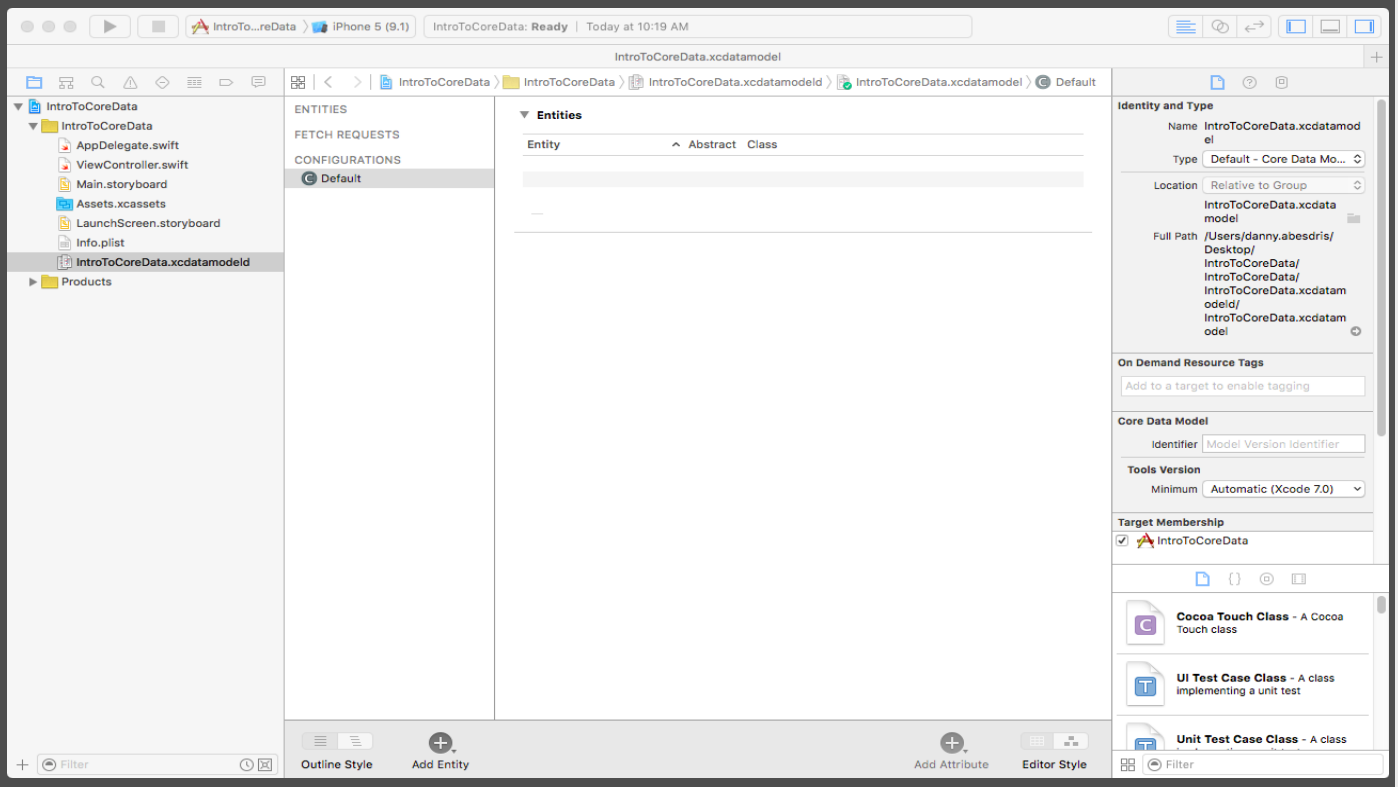
```
/*
file:    lecture7.161.swift
author:  Danny Abesdris
date:    March 9, 2016
purpose: MAP523AA/DPS923AA lecture #7

             Swift programming language topics:
             Inheritance, Polymorphism.
             Introduction to Core Data
             Core Data GUI example.
*/
/*
Swift Inheritance:
When creating new classes, Swift allows the new class to be derived from an existing cl
The existing class is referred to as the superclass, and the new class is the subclass
A subclass can add its own properties, initializers and methods, and it can customize a
methods that it inherits. A subclass is therefore more specific than its superclass and
represents a more specialized object type. Each superclass can have many subclasses, ho
each subclass has exactly one superclass as Swift does not allow multiple inheritance.
In Swift, only classes can inherit from another class. In other words, structures and
enumerations do not support inheritance and this is another key difference between
classes and structures.

Inheritance creates an is-a relationship between classes, and enables an object of a
subclass to be treated as an object of its superclass. For example, a Car (subclass) is
Vehicle (superclass), or an Eagle is a Bird, a Triangle or Circle are Shape's, etc.

A familiar example of inheritance would be in ViewController.swift, when the ViewContro
class was derived from UIViewController:
*/
class ViewController: UIViewController {
   override func viewDidLoad() {
      super.viewDidLoad()
   }
   override func didReceiveMemoryWarning() {
      super.didReceiveMemoryWarning()
   }
}

/*
Here, the ViewController is the base class and is said to derive from UIViewController
Also recall that the override keyword indicates that the function being coded has
been inherited from the base class or from a class higher up in the inheritance hierarc
Recall also that the "super" keyword is used to refer explicitly a property or method i
a base (or higher up) class.

Another example:
*/
import Foundation

class BankAccount {
```

```
    var balance: Float
    var accountNumber: String

    init(number: String, balance: Float) {
         accountNumber = number
         self.balance = balance
    }

    func displayBalance( ) {
       print("Acct. Number    : \(accountNumber)")
       print("Current balance: \(balance)")
    }
}

class SavingsAccount: BankAccount {
    var rateOfInterest: Float
    var balanceWithInterest: Float

    init(number: String, balance: Float, rate: Float) {
         rateOfInterest = rate
         balanceWithInterest = balance * (1 + rateOfInterest)
         super.init(number: number, balance: balance)
    }
    func calculateInterest( ) -> Float {
       return rateOfInterest * balance
    }
    override func displayBalance( ) { // an example of polymorphism
       super.displayBalance( )
       print("Interest rate   : \(rateOfInterest)")
       print("Updated balance: \(balanceWithInterest)")
    }
}
var savingsAcct = SavingsAccount(number: "081993617", balance: 600.00, rate: 0.07)

savingsAcct.displayBalance( )
print("Total Interest : \(savingsAcct.calculateInterest( ))")


/*
Polymorphism:
Polymorphism enables you to write programs which process objects that share the same
superclass, either directly or indirectly, as if they were all objects of the superclas
thus simplifying programming.

For example, given the classes Fish, Frog and Bird that are all derived from a base (s
class Animal which contains a method called move which maintains an animals current
location as x-y coordinate properties. If each subclass implements its own specialized
version of method move, then if an array of Animal objects is created that stores
references to objects of the various Animal subclasses, for each call to that object's
move( ) the program would invoke the move that is specific to each object type.
*/
class Animal {
    var xCoord: Int
    var yCoord: Int
    var name: String

    init(name: String, x: Int, y: Int) {
       self.name = name
       xCoord = x
       yCoord = y
    }
```

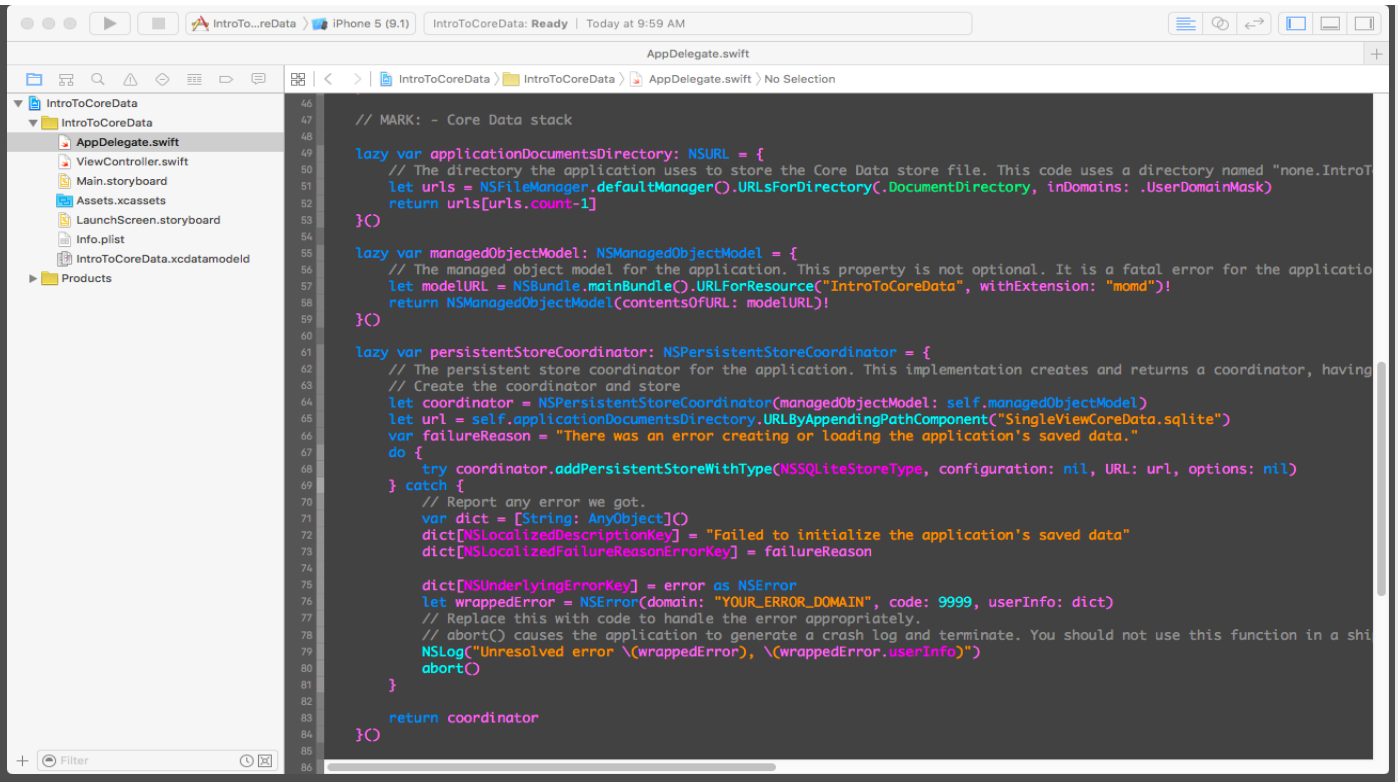
```
    func move( ) {
        print("\(name)'s current location is: (\(xCoord), \(yCoord))")
    }
}

class Fish: Animal {
    init(nm: String, x: Int, y: Int) {
        super.init(name:nm, x:x, y:y)
    }
    override func move( ) {
        super.move( )
        print("A \(name) swims!")
    }
}

class Frog: Animal {
    init(nm: String, x: Int, y: Int) {
        super.init(name:nm, x:x, y:y)
    }
    override func move( ) {
        super.move( )
        print("A \(name) Jumps and is amphibious!")
    }
}

class Bird: Animal {
    init(nm: String, x: Int, y: Int) {
        super.init(name:nm, x:x, y:y)
    }
    override func move( ) {
        super.move( )
        print("A \(name) flies!")
    }
}

var animals : [Animal] = [ ]

animals.append(Fish(nm: "Salmon", x:8, y:9))
animals.append(Frog(nm: "Green Toad", x:18, y:21))
animals.append(Bird(nm: "Falcon", x:197, y:352))

for i in 0..>animals.count {
    animals[i].move( )
}

/*
Core Data:
Core Data is the standard way to persist and manage data in both iPhone and Mac applica
and has been made easier with the use of Swift. When an app uses Core Data to store dat
information is persistent and lasts even through a complete shut down of your phone or

Core Data is not a relational database, and although it uses SQLite as a backing engine
it is not a relational database either. The SQLite backend is more of an implementation
detail, and in fact binary files or plists can be used instead.

The official Apple documentation describes Core Data as:
"Core Data is a framework that you use to manage the model layer objects in your applic
It provides generalized and automated solutions to common tasks associated with object
cycle and object graph management, including persistence."

Apple Core Data
```

Creating a Core Data based Application:
Xcode does much of the preparatory work when developing an iOS application that will us
Core Data. To create a sample application project, launch Xcode and select the option t
create a new Single View Application called "IntroToCodeData" with Devices=iPhone and
language=Swift.
Make sure to check the the "Use Core Data" checkbox before saving your app.

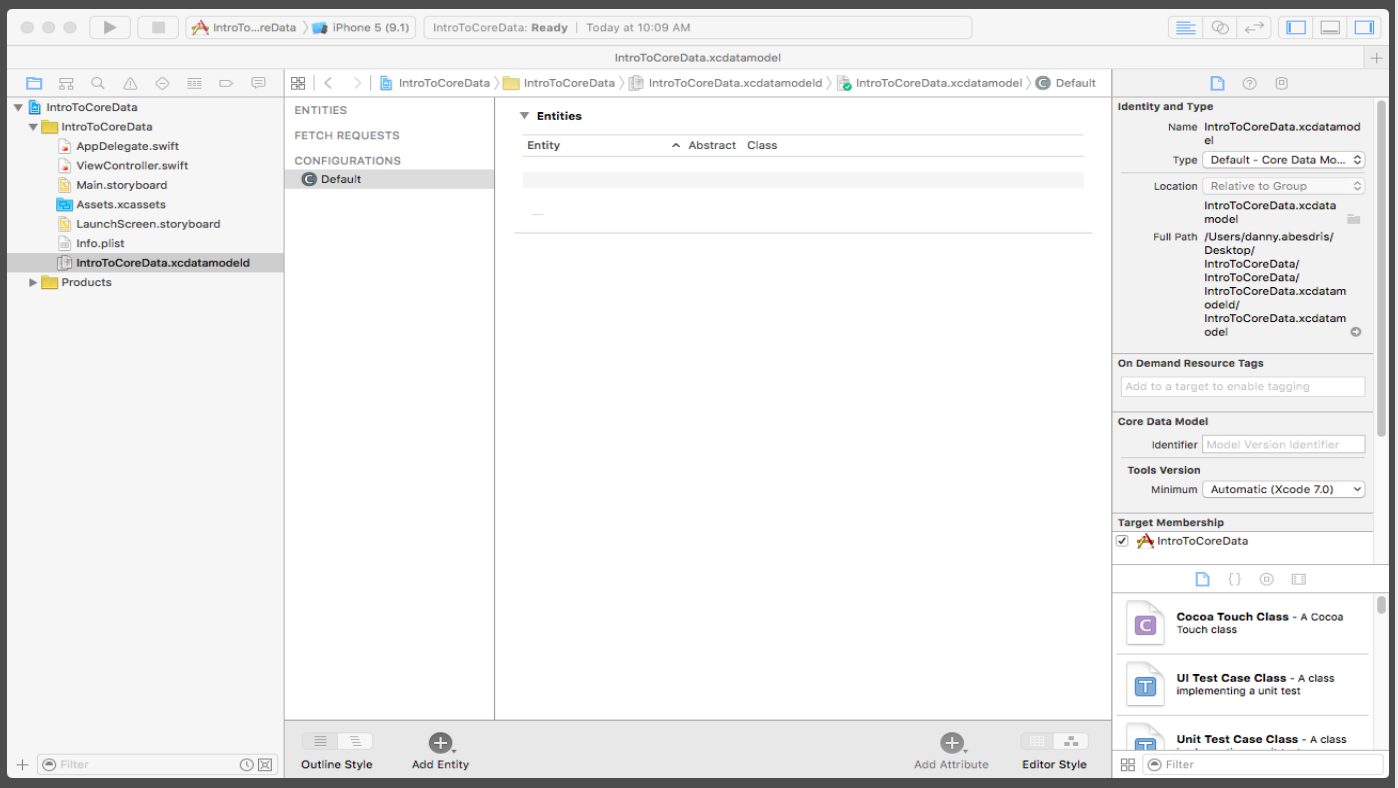
Note, Xcode (in addition to the usual files that are present when creating a new projec
has created an additional file named IntroToCoreData.xcdatamodeld. This is the file whe
the entity descriptions for the data model are going to be stored.



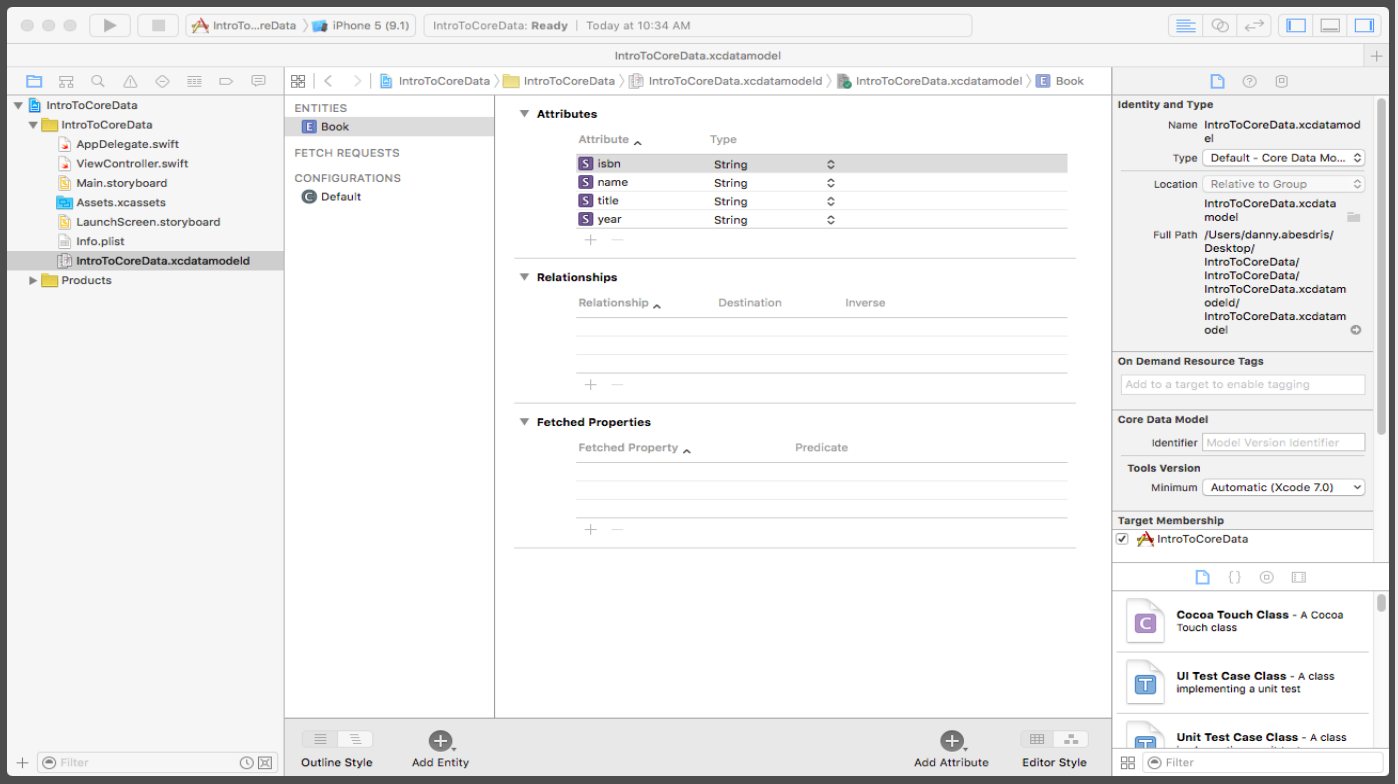
Notice also Xcode has now added to the AppDelegate.swift file and included several
properties and a method related to the Core Data Stack.



Creating the Entity Description:
The entity description defines the model for your data in much the way that a schema defines the model of a database table. To create an entity for the Core Data applicati select the IntroToCoreData.xcdatamodeld file to load the entity editor:

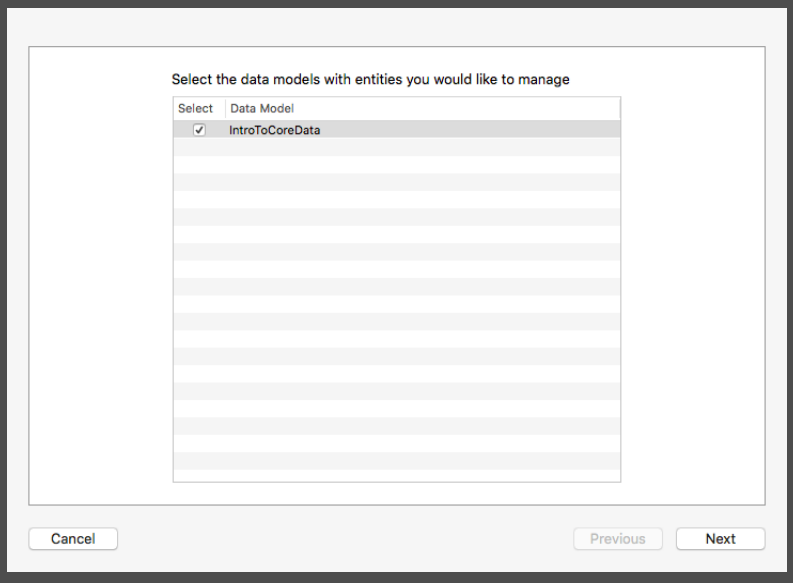


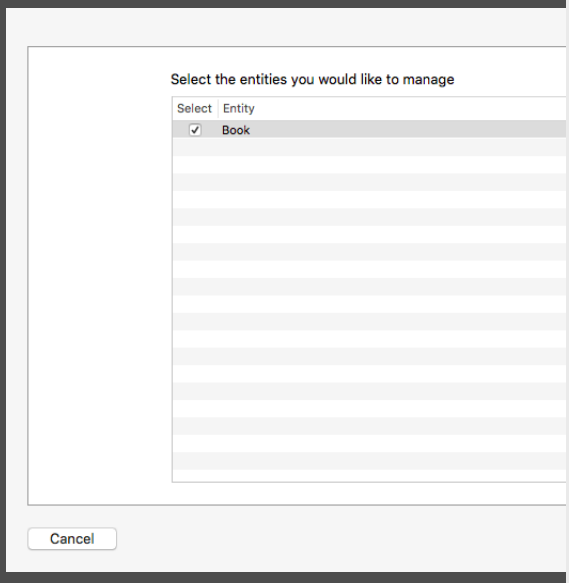
To create a new entity, click on the Add Entity button located in the bottom panel. Double click on the new Entity item that appears beneath the ENTITIES: heading and cha entity name to Book. With the entity created, the next step is to add some attributes represent the data that is to be stored. To do so, click on the Add Attribute button o bottom panel. In the Attribute pane, name the attribute "name" and set the Type to Str Repeat these steps to add three other String attributes named title, year, and isbn re



Generating the Managed Object Subclass:

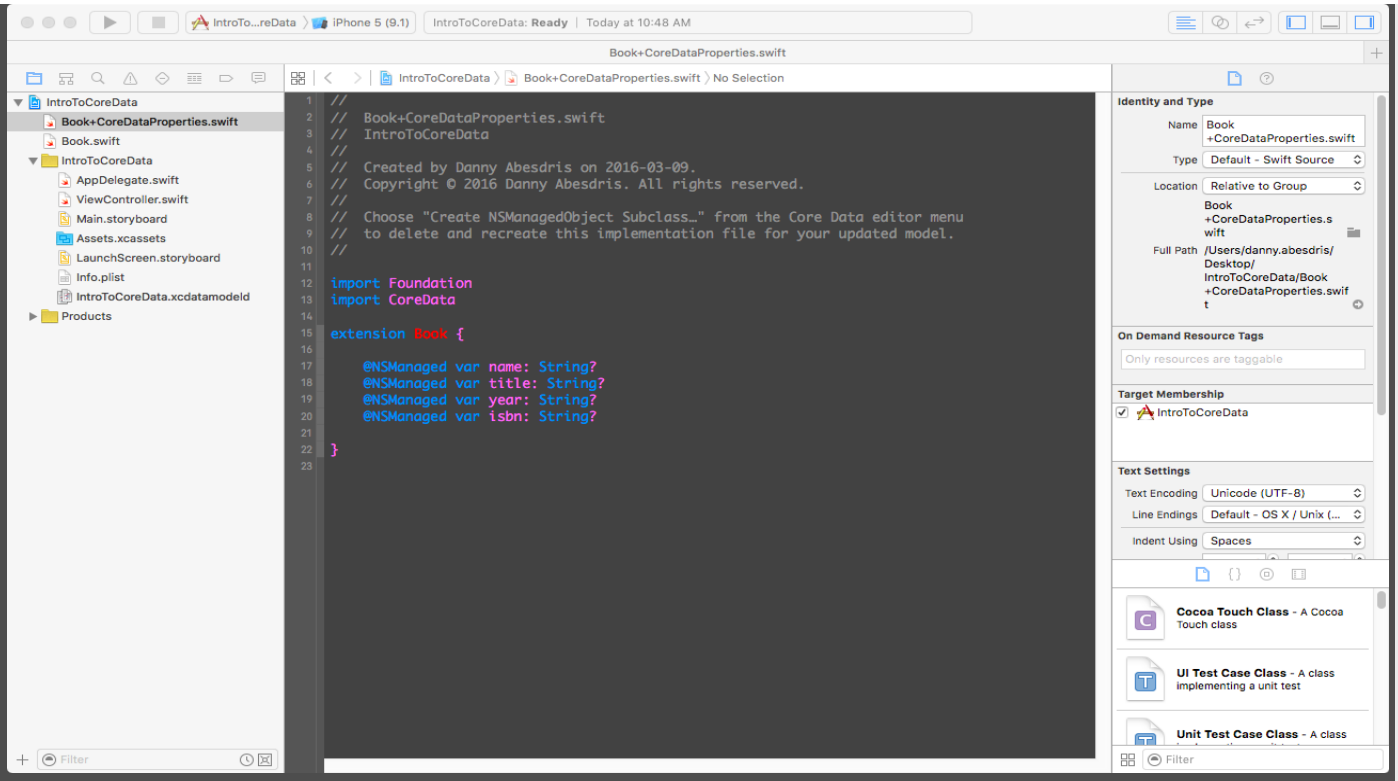
Within the application the data model is going to be represented by a subclass of NSManagedObject. However, rather than manually write the code for this class, Xcode ca be used to automatically generate the class for you. With the Entity editor still displ in Xcode, select the Editor -> Create NSManagedObject Subclass... menu option to displa the dialog shown below and make sure that the IntroToCoreData data model is selected b clicking on the Next button to display the entity selection screen:





Now, verify that the Book entity is selected before clicking on the Next button. Withi the final screen, set the Language menu to Swift and click on the Create button.

A new file named Book.swift will have been added to the project containing the NSManag subclass for the entity which reads as follows:



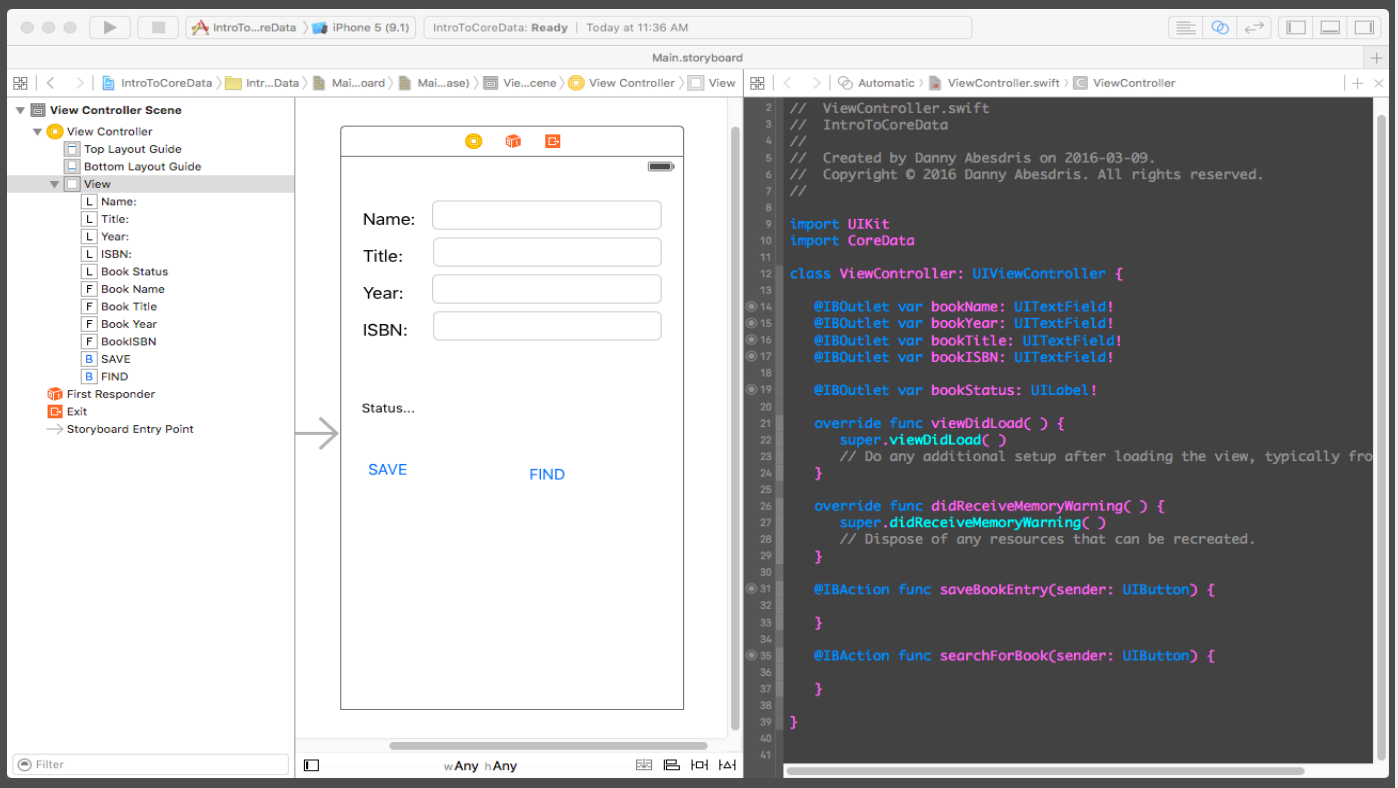
Modifying the Entity Class Name:

When using Core Data within Swift based code it is important to include the application name as part of the entity class name. By default, the new entity has been assigned a name of Book.

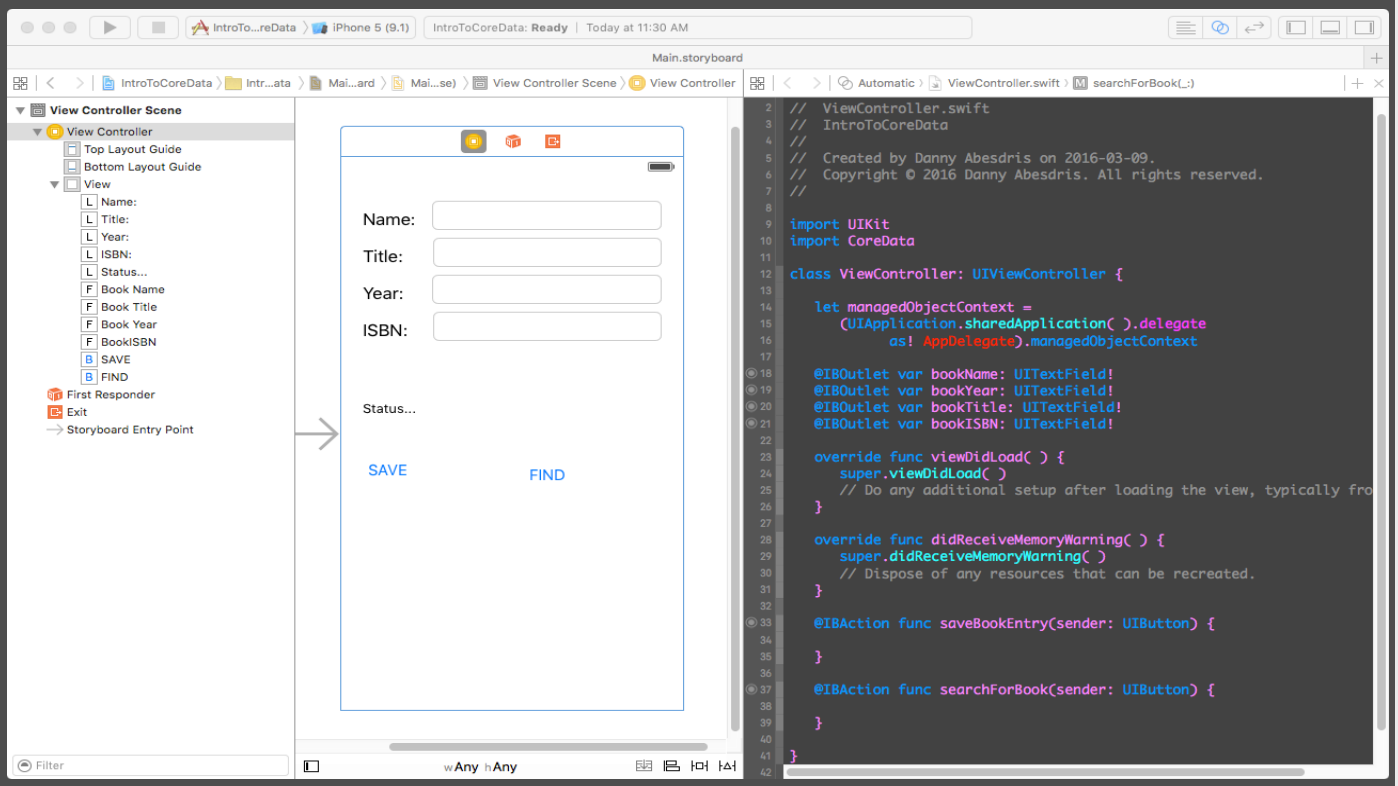
Designing the UI:

With the entity now defined, the user interface can now be created to establish the outlet and action connections. Select Main.storyboard and add the required labels, textfield, Button UI elements as well as the required corresponding connections.

The completed view should appear as outlined below:



NOTE: It is important to include the
import CoreData
In order to store and retrieve data using Core Data. Also, a reference to the applicat
delegates managed object context is required. So, within the ViewController.swift file
import the CoreData Framework and add a variable to store this reference as follows:




```
//  ViewController.swift
//  IntroToCoreData
//
//  Created by Danny Abesdris on 2016-03-09.
//  Copyright © 2016 Danny Abesdris. All rights reserved.
//

import UIKit
import CoreData

class ViewController: UIViewController {

    let managedObjectContext =
        (UIApplication.sharedApplication( ).delegate
            as! AppDelegate).managedObjectContext

    @IBOutlet var bookName: UITextField!
    @IBOutlet var bookYear: UITextField!
    @IBOutlet var bookTitle: UITextField!
    @IBOutlet var bookISBN: UITextField!

    override func viewDidLoad( ) {
        super.viewDidLoad( )
        // Do any additional setup after loading the view, typically fro
    }

    override func didReceiveMemoryWarning( ) {
        super.didReceiveMemoryWarning( )
        // Dispose of any resources that can be recreated.
    }

    @IBAction func saveBookEntry(sender: UIButton) {

    }

    @IBAction func searchForBook(sender: UIButton) {

    }

}
```

Saving Data to the Persistent Store using Core Data:
When the user clicks on the Save button the saveBookEntry method is called. This metho
be used to create and store managed objects containing the data entered by the user.
Therefore, select the ViewController.swift file and add the following code to the save
method:

```
*/
@IBAction func saveBookEntry(sender: UIButton) {
    let entityDescription =
    NSEntityDescription.entityForName("Book",
        inManagedObjectContext: managedObjectContext)

    let book = Book(entity: entityDescription!,
        insertIntoManagedObjectContext: managedObjectContext)

    book.name = bookName.text
    book.title = bookTitle.text
    book.year = bookYear.text
    book.isbn = bookISBN.text

    var userError: NSError? = nil

    do {
        try managedObjectContext.save( )
        bookStatus.text = "Book Data Saved!"
    }
    catch {
        // error as NSError
        userError = error as NSError
        NSLog("Unresolved error \(userError)")
```

```
        print("Error saving Book Entry...")
    }

    if let err = userError {
        bookStatus.text = err.localizedFailureReason
    }
    else {
        bookName.text = ""
        bookTitle.text = ""
        bookYear.text = ""
        bookISBN.text = ""
        bookStatus.text = "Book Data Saved!"
    }
}

/*
The code above uses the managed object context to obtain the Book data and then
uses it to create a new instance of the Book managed object subclass. The book fields
are then set to the current text field values. Then the context is instructed to save
the changes to the persistent store with a call to the contexts save method.
The success or otherwise of the operation is reported on the status label and, in
the case of a successful outcome, the text fields are cleared ready for the next Book
to be entered.

Retrieving Data from the Persistent Store using Core Data:
In order to allow the user to search for a book it is now necessary to implement
the searchForBook action method. As with the save method, this method will need to
identify the entity description for the Book entity and then create a predicate to
ensure that only objects with the name specified by the user are retrieved from the
store. Matching objects are placed in an array from which the attributes for the
first match are retrieved using the valueForKey method and displayed to the user.
A full count of the matches is displayed in the status field. The code is listed below
*/
@IBAction func searchForBook(sender: UIButton) {
    let entityDescription =
    NSEntityDescription.entityForName("Book",
        inManagedObjectContext: managedObjectContext)

    let request = NSFetchRequest( )
    request.entity = entityDescription

    let pred = NSPredicate(format: "(name = %@)", bookName.text!)
    request.predicate = pred

    var userError: NSError? = nil
    var objects : [AnyObject]?

    do {
        objects = try managedObjectContext.executeFetchRequest(request)
        bookStatus.text = "Book Data Saved!"
    }
    catch {
        userError = error as NSError
        NSLog("Unresolved error \(userError)")
        print("Error fetching Book Entry...")
    }

    if let results = objects {
        if results.count > 0 {
            let match = results[0] as! NSManagedObject
```

```
            bookName.text = (match.valueForKey("name") as! String)
            bookTitle.text = (match.valueForKey("title") as! String)
            bookYear.text = (match.valueForKey("year") as! String)
            bookISBN.text = (match.valueForKey("isbn") as! String)

            bookStatus.text = "Matches found: \(results.count)"
        }
        else {
            bookStatus.text = "No Match"
        }
    }
}
/*
The Core Data Framework provides an abstract, object oriented interface to database
storage within iOS applications. As demonstrated in the sample application (above),
Core Data does not require any knowledge of the underlying database system and, combine
with the visual entity creation features of Xcode, allows database storage to be
implemented with little effort!
*/
```