26

Introduction to Core Data

The Core Data framework provides solutions to tasks commonly associated with managing the lifecycle of objects in your application, including object serialization. Prior to Core Data, programmers relied on SQLite to store their application data; Core Data can be viewed as an object-oriented wrapper around a SQLite database. It provides you with a convenient mechanism to create, update, and delete entities in the database without having to write a single line of SQL. In this lesson, you learn to use Core Data to implement simple object persistence in your applications.

BASIC CONCEPTS

Core Data is based on the Model-View-Controller pattern and essentially fits in at the model stage. It forces you to think of your applications data in terms of objects. Core Data introduces quite a few new concepts and terminology. These are discussed briefly in this section. Figure 26-1 provides an overview of the key classes introduced by Core Data.

Managed Object

A managed object is a representation of the object that you want to save to the data store. This is conceptually similar to a record in a relational database table and typically contains fields that correspond to properties in the object you want to save. The lifecycle of managed objects is managed by Core Data; you should not hold strong references in your code to managed objects. Managed objects are subclasses of `NSManagedObject` and not `NSObject`.

Managed Object Context

The managed object context is akin to a buffer between your application and the data store. It contains all your managed objects before they are written to the data store and manages their lifecycles. Inside this context you can add, delete, or modify managed objects. When you load data from the underlying data store, managed objects that are created as a result will live within the managed object context. When you need to read, insert, or delete objects, you will call methods on the managed object context. A managed object context is represented by an instance of the `NSManagedObjectContext` class. Managed object contexts should only be accessed from the thread in which they were created. An application can have multiple managed object contexts all of which can be connected to a single persistent store coordinator, and in effect are talking to the same database.

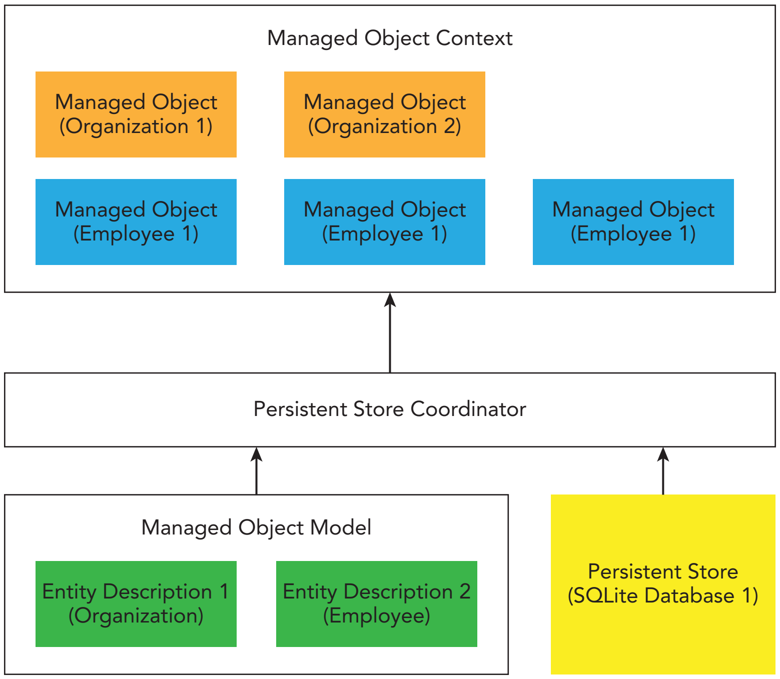

Managed Object Context
Managed Object (Organization 1)
Managed Object (Organization 2)
Managed Object (Employee 1)
Managed Object (Employee 1)
Managed Object (Employee 1)
Persistent Store Coordinator
Managed Object Model
Entity Description 1 (Organization)
Entity Description 2 (Employee)
Persistent Store (SQLite Database 1)

FIGURE 26-1

Persistent Store Coordinator

The persistent store coordinator is an instance of `NSPersistentStoreCoordinator` and represents the connection to the data store. It contains low-level information, such as the name, location, and type of the data store to be used, as well as handling the task of communicating with the store. Your application will have one instance of the persistent store coordinator for each database that it needs to interact with.

The persistent store coordinator is used by the managed object context, and in most cases, you will not need to deal with it directly. Multiple managed object contexts can share the same instance of the persistent store coordinator, and Core Data handles the synchronization of data across all these contexts.

You may have more than one managed object context sharing the persistent store coordinator if you want to access your managed objects from different threads. In such cases you will have one managed object context per thread.

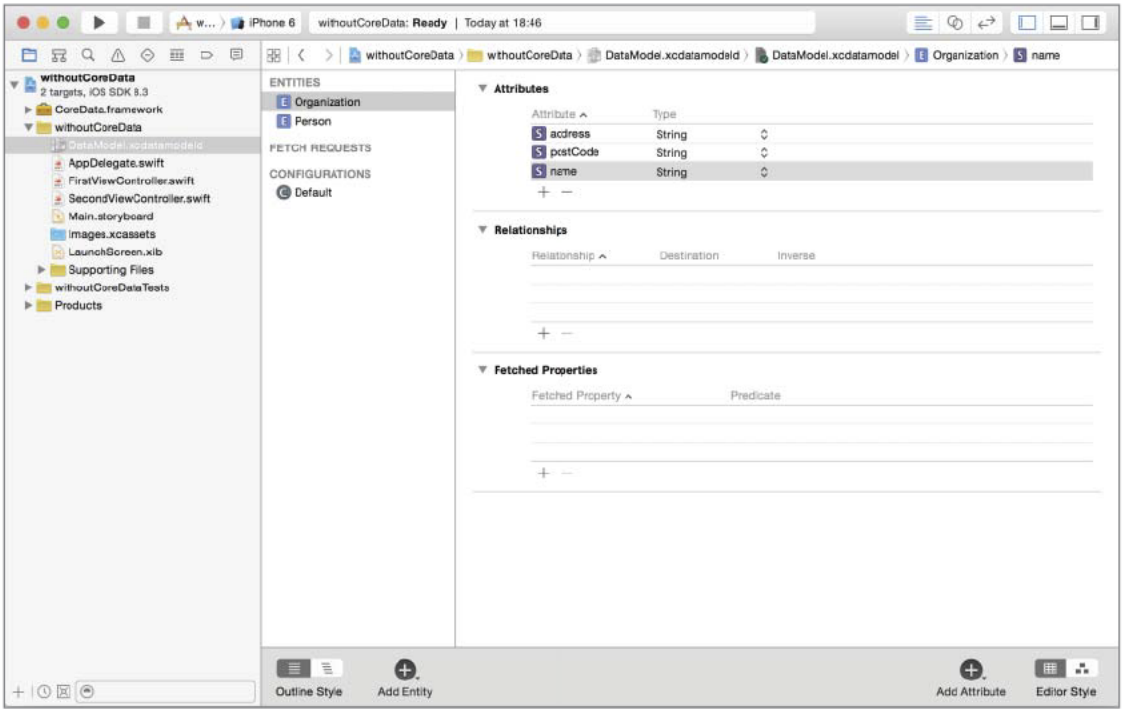
Entity Description

An entity description is an instance of `NSEntityDescription` and essentially describes a table within the database. In Core Data terms, database tables are called entities. It is rare for a programmer to create entity descriptions programmatically, but it can be done. The most common method to create entity descriptions is to use the graphical Core Data editor included within XCode.

It is worth nothing that an entity description is similar to the schema for a database table. It does not contain the actual data; it is used internally by Core Data to create tables in the underlying database.

Managed Object Model

The managed object model is an instance of `NSManagedObjectModel` and is a collection of entity descriptions. When Core Data is used in a project, the project contains a file that ends with the extension `.xcdatamodeld`. This file is used by XCode to build a graphical editor for the managed object model (see Figure 26-2).



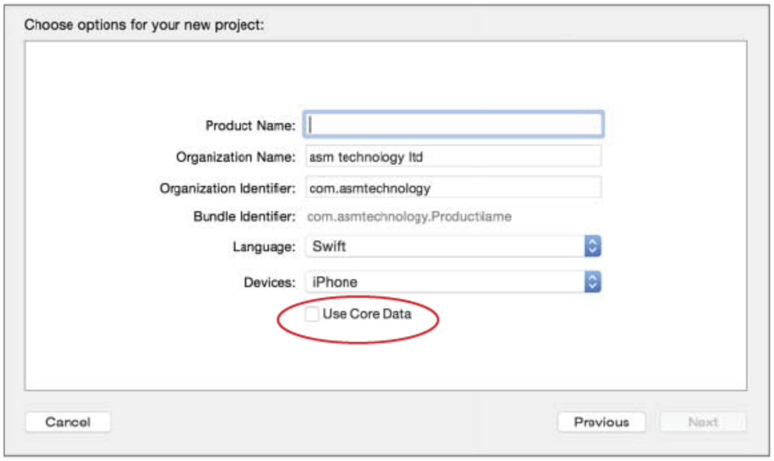
Attribute	Type
address	String
postCode	String
name	String

FIGURE 26-2

When your project is compiled into an executable, this file is compiled into a `.mom` file, which is the managed object model in binary format. For most practical purposes, the `.xcdatamodel` file can be considered to be the managed object model. However, it is important to keep in mind that this file will be compiled to produce the managed object model.

ADDING CORE DATA TO A PROJECT

When you create a new project based on the Master Detail or Single View Application templates, you have the option to include Core Data in the project in the project options dialog box (see Figure 26-3).


Choose options for your new project:

Product Name:
Organization Name: asm technology ltd
Organization Identifier: com.asmtechnology
Bundle Identifier: com.asmtechnology.ProductName
Language: Swift
Devices: iPhone
Use Core Data

FIGURE 26-3

However, for other application types, this checkbox does not exist. This section walks you through what you need to do to add Core Data into a project manually.

To add Core Data to your project, you first need to add a reference to the framework. You can do this from the Project Settings page in Xcode. Select the project node in the project navigator to display the settings page. On the settings page, select the appropriate build target and then switch to the Build Phases tab. Click the + button under the Link Binary With Libraries category. Select CoreData.framework from the list of available frameworks (see Figure 26-4).

The next step is to create a managed object model for the project. To create an empty model file (into which you will later add entities), right-click the project group in the project navigator and select New File from the context menu. Select the Data Model template from the Core Data section and create the new file (see Figure 26-5).

To open the model in the Xcode editor, simply click the file in the project navigator (the model file has the `.xcdatamodeld` extension). The new model file is initially empty (see Figure 26-6), and as such is not much use to you in this state.

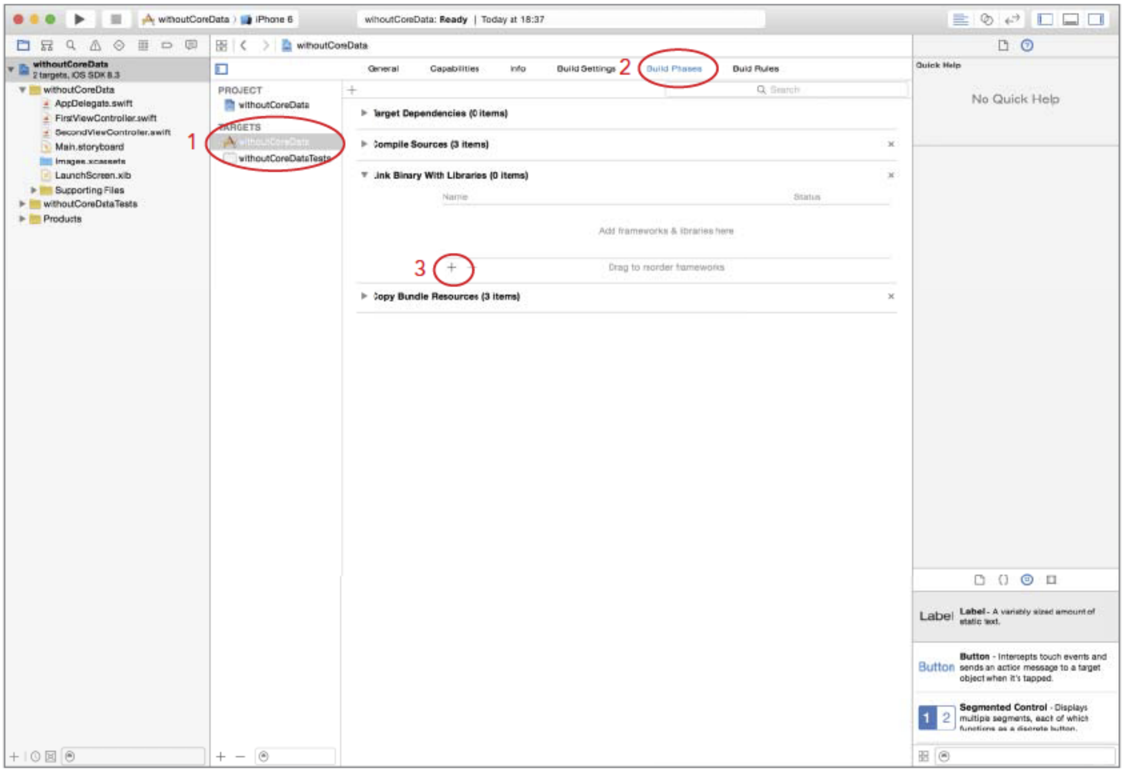
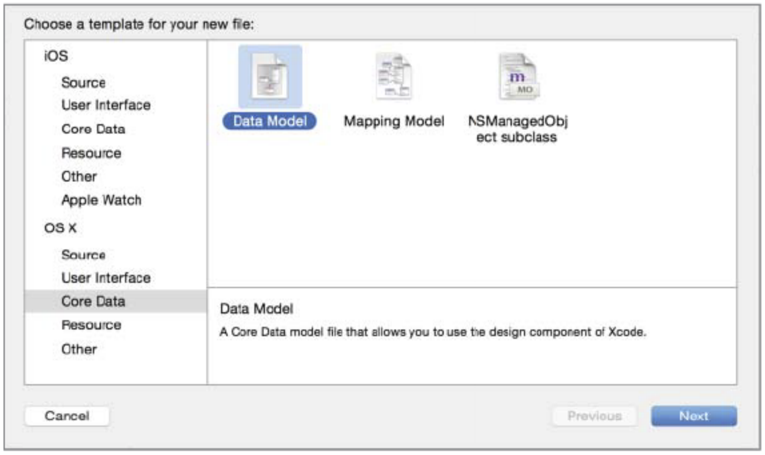

withoutCoreData
iPhone 6
withoutCoreData: Ready | Today at 18:37
withoutCoreData
2 targets, iOS SDK 8.3
AppDelegate.swift
FirstViewController.swift
SecondViewController.swift
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
withoutCoreDataTests
Products
PROJECT
TARGETS
withoutCoreDataTests
1
General
Capabilities
Info
Build Settings
2
Build Phases
Build Rules
Target Dependencies (0 items)
Compile Sources (3 items)
Link Binary With Libraries (0 items)
Name
Status
Add frameworks & libraries here
3
Drag to reorder frameworks
Copy Bundle Resources (3 items)
Quick Help
No Quick Help
Label - A variably sized amount of static text.
Button - Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control - Displays multiple segments, each of which functions as a discrete button.

FIGURE 26-4



Data Model

A Core Data model file that allows you to use the design component of Xcode.

FIGURE 26-5

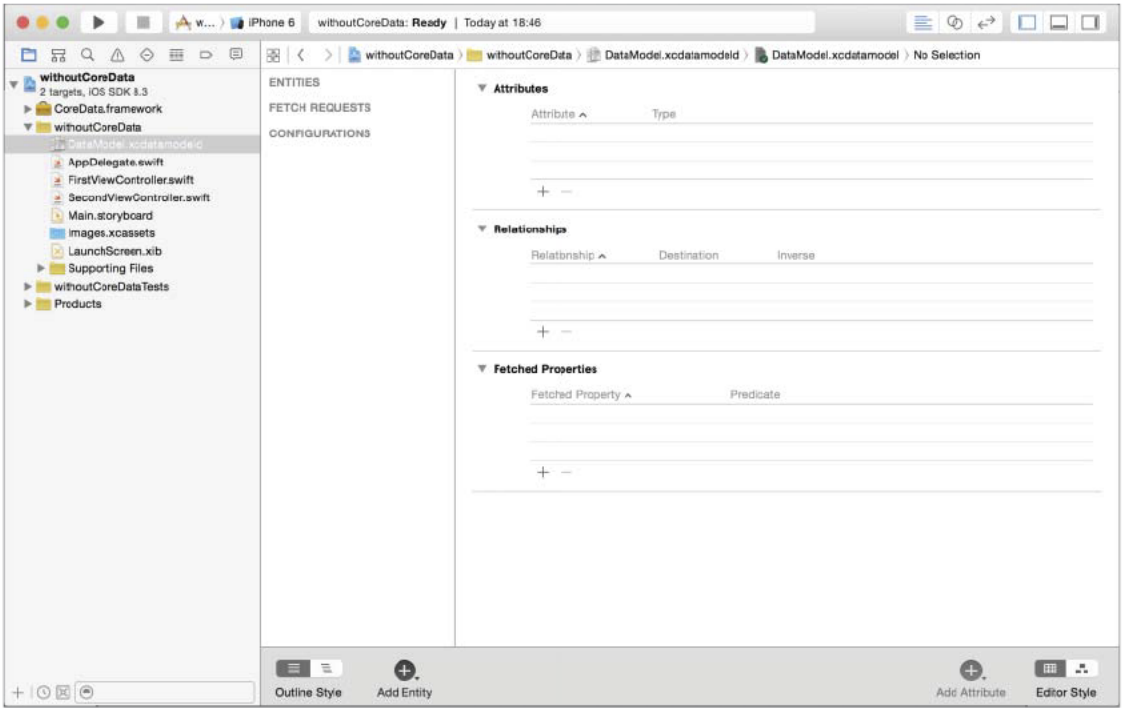


FIGURE 26-6

To persist objects into the underlying data store, you first need to define an entity in the data model for each object that you want to persist. Defining entities is trivial with the Xcode editor: To add a new entity called `ContactData`, select Editor ➪ Add Entity and name the new entity appropriately. You will see the new entity listed under the Entities section of the Xcode editor (see Figure 26-7).

After you have defined an entity, you need to add attributes to it. Attributes represent the actual data fields in the entities themselves. Assuming the `ContactData` entity represents customer contact information, some of its attributes may be:

- Customer Name
- Phone Number
- Postcode

To add an attribute to the currently selected entity, select Editor ➪ Add Attribute. This adds a new row to the Attributes section of the Xcode model editor (see Figure 26-8).

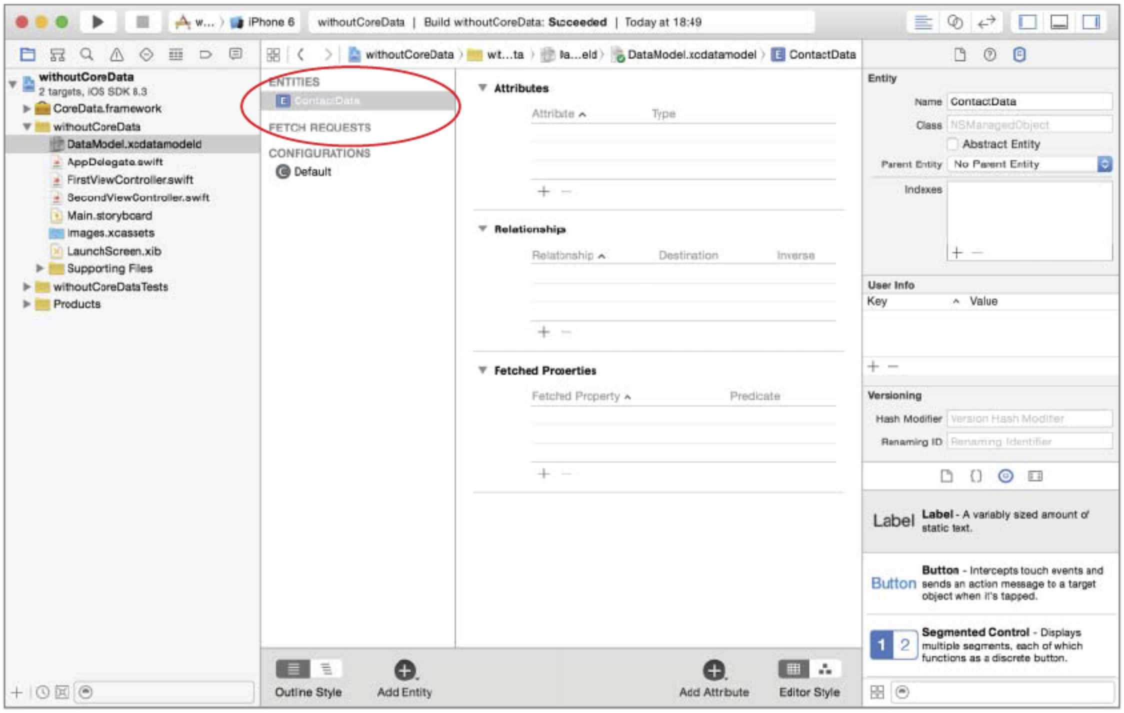


FIGURE 26-7

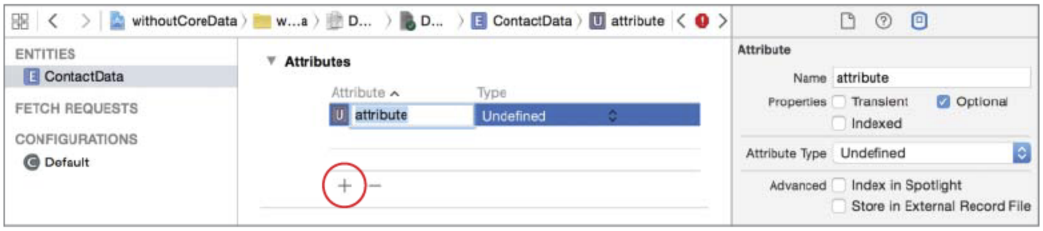


FIGURE 26-8

Type in an appropriate name for the attribute and specify the attribute type. Attribute names must begin with a lowercase letter and cannot contain whitespace. The attribute type is similar to the data type of a variable, and determines what type of data the attribute contains. Core Data provides several data types that can be selected from a drop-down list (see Figure 26-9). The type for each attribute of the `ContactData` entity can be `String`.

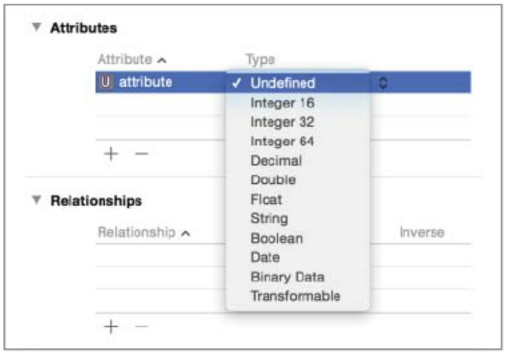


FIGURE 26-9

At this stage, you have created a new data model and added an entity to it. Now you need an actual Swift class that maps to the entity defined in the model. To do this, select Editor ➪ Create NSManagedObject Subclass. This presents a dialog box asking you where to save the `file` for the new class. In this dialog box, ensure the language is set to Swift (see Figure 26-10).

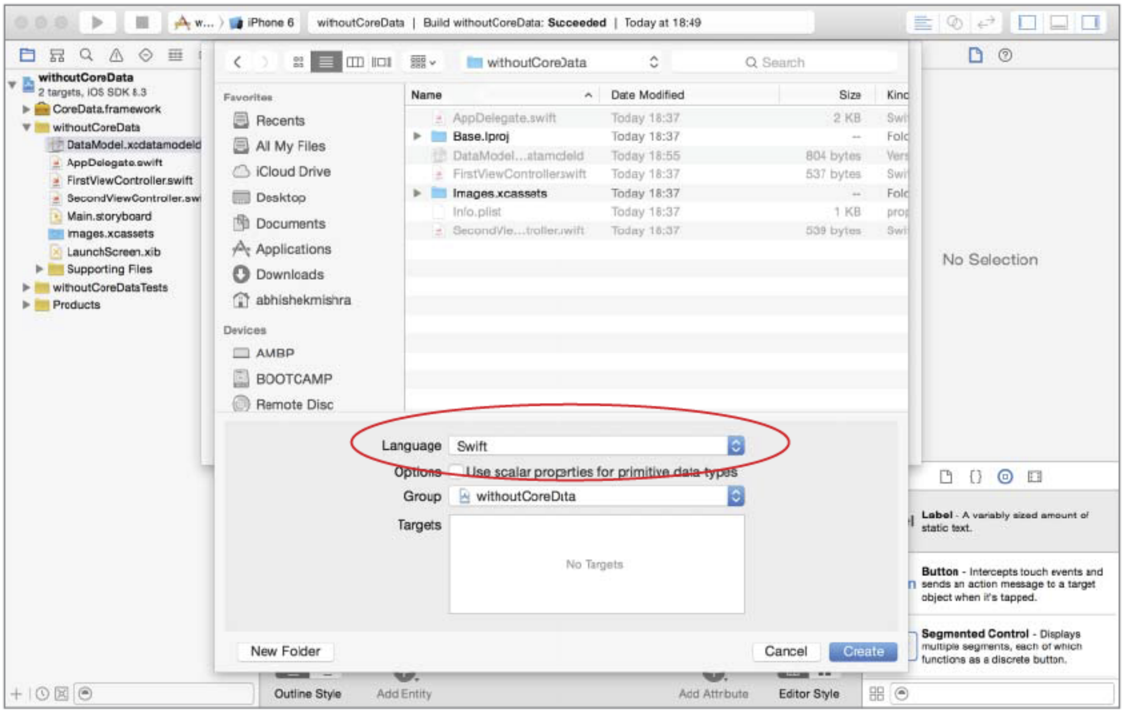


FIGURE 26-10

The name of the class will be the same as the name of the entity. The `ContactData` class that is created for you by Xcode is a subclass of `NSManagedObject` and maps to the entity with the same name. Its interface is listed here:

```
import Foundation
import CoreData

class ContactData: NSManagedObject {

    @NSManaged var customerName: String
    @NSManaged var phoneNumber: String
    @NSManaged var postCode: String

}
```

You need to ensure that the name of the class that corresponds to the entity in the data model is set up correctly. To do this, select the Entity in the `xcdatamodeld` file and switch to the Data Model Inspector by selecting View ➪ Utilities ➪ Show Data Model Inspector. Examine the value of the Class property; it should be set to `<your project name>.ContactData`.

It is worth mentioning that if you now decided to make changes to the entity in the `.xcdatamodel` file, the managed object class will not automatically update. You will need to regenerate the managed object class and this process will overwrite the contents of the previous managed object class header and implementation files. If you need to add code to the Core Data–generated class files, it is best to do so in a subclass of the class generated by Core Data.

INSTANTIATING CORE DATA OBJECTS

Before you can read or write model objects to the underlying data store, you will need to instantiate the managed object model, the managed object context, and the persistent store coordinator.

The managed object model is represented by an instance of the `NSManagedObjectModel` class, and you instantiate a single instance for the `.xcdatamodeld` file in your project using the following snippet.

```
let modelURL = NSBundle.mainBundle().URLForResource("withCoreData",
    withExtension: "momd")!

var managedObjectModel: NSManagedObjectModel =
    NSManagedObjectModel(contentsOfURL: modelURL)!
```

Once you have an `NSManagedObjectModel` instance, you can create an instance of the `NSPersistentStoreCoordinator` class, which represents the persistent store coordinator. Recall that the persistent store coordinator handles the low-level connection with underlying data stores. Individual databases are referred to as *persistent stores*.

To create an NSPersistentStoreCoordinator instance, use the following snippet:

```
var coordinator: NSPersistentStoreCoordinator? =
NSPersistentStoreCoordinator(managedObjectModel:
self.managedObjectModel)
```

Once you have the store coordinator, you need to give it a data store to manage. You do this by calling the addPersistentStoreWithType(storeType, configuration, URL, options) method on the store coordinator object. For instance, the following code snippet sets up a SQLite database as the data store:

```
let urls =
NSFileManager.defaultManager().URLsForDirectory(.DocumentDirectory,
inDomains: .UserDomainMask)

var applicationDocumentsDirectory:NSURL = urls[urls.count-1] as! NSURL

let url = applicationDocumentsDirectory.URLByAppendingPathComponent("data.sqlite")

do {
   try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
       configuration: nil, URL: url, options: nil)
   } catch {
      // Report any error.
 }
```

Finally, with the store coordinator object in place, it is time to instantiate a managed object context. Recall that a managed object context is like a buffer where you place your managed objects before writing to (or reading from) the database. The managed object context is represented by an instance of the NSManagedObjectContext class and can be created as follows:

```
var managedObjectContext = NSManagedObjectContext(
    concurrencyType: .MainQueueConcurrencyType)
managedObjectContext.persistentStoreCoordinator = coordinator
```

WRITING MANAGED OBJECTS

Instantiating a managed object is slightly different from the usual process. With managed objects, you allow Core Data to instantiate them within a managed object context. Once the object has been instantiated, you can use it as you would any other object. To instantiate a ContactData object, use the following code:

```
let newContact =
NSEntityDescription.insertNewObjectForEntityForName("ContactData",
inManagedObjectContext:managedObjectContext) as! ContactData
```

Now that you have instantiated a ContactData object, you can set up its attributes just as you would for any object:

```
newContact.customerName = "John Smith";
newContact.phoneNumber = "+44 78901 78192";
newContact.postcode = "PB2 7YK";
```

To write managed objects to the data store, simply call the `save` method of the managed object context. Doing so saves any new objects to the underlying data store (by using the persistent store coordinator). The `save` method returns a Boolean value indicating success or failure.

```
do {
    try managedObjectContext.save()
} catch {
    // handle error.
}
```

READING MANAGED OBJECTS

Reading objects from a data store with Core Data is quite straightforward. You simply create an appropriate fetch request and ask the managed object context to execute the request. The managed object context will then return an array of objects read from the data store.

A fetch request is an instance of the `NSFetchRequest` class, and is similar to a SELECT statement in SQL. When creating a fetch request, you need to specify the entity that you want to fetch. The entity has to be one that exists in the data model. To create a fetch request that retrieves all `ContactData` entities from the data store, use the following code:

```
let fetchRequest = NSFetchRequest(entityName: "ContactData")
```

To retrieve an array of managed objects from the data store, you need to ask the managed object context to execute the fetch request, as shown in the following snippet:

```
do {
    if let fetchResults = try
    appDelegate.manaedObjectContext!.executeFetchRequest(fetchRequest)
    as? [ContactData] {
       // fetchReults is now an array of ContactData objects.
    }
} catch {
    // handle errors here.
}
```

TRY IT

In this Try It, you build an iPhone application based on the Single View Application template called `CoreDataTest` that can serialize/de-serialize object data to an SQLite database using Core Data.

Lesson Requirements

- Launch Xcode.
- Create a new project based on the Single View Application template.
- Add an entity to the data model.
- Create an `NSManagedObject` subclass.
- Create a simple user interface with a storyboard.

- Initialize Core Data objects.
- Save managed objects to the database with Core Data.
- Read managed objects from the database with Core Data.

> **REFERENCE** *The code for this Try It is available at* `www.wrox.com/go/swiftios`.

Hints

- When creating a new project, you can use your website's domain name as the Company Identifier in the Project Options dialog box.
- To show the Object library, select View ➪ Utilities ➪ Show Object Library.
- To show the assistant editor, select View ➪ Assistant Editor ➪ Show Assistant Editor.
- Ensure the Use Core Data option is selecting when creating the project.

Step-by-Step

- Create a Single View Application in Xcode called `CoreDataTest`.
  1. Launch Xcode and create a new application by selecting File ➪ New ➪ Project.
  2. Select the Single View Application template from the list of iOS project templates.
  3. In the project options screen, use the following values:
     - Product Name: CoreDataTest
     - Organization Name: your company
     - Organization Identifier: com.yourcompany
     - Language: Swift
     - Devices: iPhone
     - Use Core Data: Checked
  4. Save the project onto your hard disk.
- Edit the data model file.
  1. Select the `CoreDataTest.xcdatamodeld` file in the project navigator to open it in the Xcode editor.
  2. Add an Entity to the data model to represent contact data instances.

     Select Editor ➪ Add Entity and name the new entity `ContactData`.
  3. Add attributes to the `ContactData` entity.

- Select Editor ➪ Add Attribute to create a new attribute. Name it `customer-Name` and set its type to `String`.
- Add two more `String` attributes, `phoneNumber` and `postCode`, to the entity.

- Create an `NSManagedObject` subclass to represent the `ContactData` entity.

1. Select Editor ➪ Create NSManagedObject Subclass. You will be aksed to select the entities for which you wish to create `NSManagedObject` subclasses. Ensure the check box beside the ContactData entity is selected.
2. Accept the default file location, but ensure the language is set to Swift. Click Save to create a new class called `ContactData` in your project.
3. Select the Entity in the `CoreDataTest.xcdatamodeld` file and switch to the Data Model Inspector by selecting View ➪ Utilities ➪ Show Data Model Inspector.
4. Ensure the value of the Class field is `CoreDataTest.ContactData`.

- Create a simple user interface using a storyboard.

1. Open the `Main.storyboard` file in Interface Builder.
2. From the Object library, drag and drop five Label objects, three Text Field objects, one Button object, and one Table View object onto the scene.
3. Arrange these objects to resemble Figure 26-11.

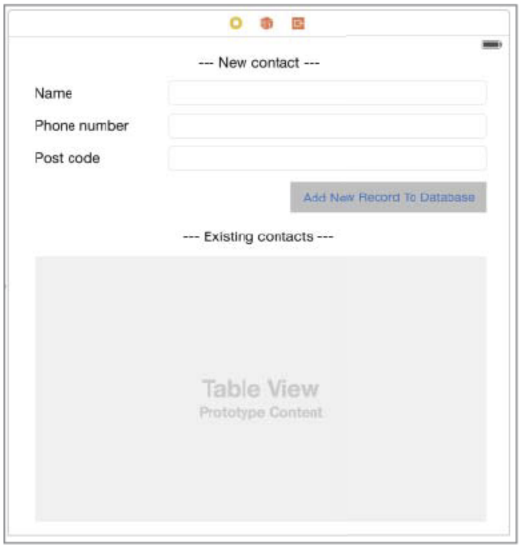


FIGURE 26-11

4. Create three outlets in the view controller class corresponding to the three Text Field objects in the scene. Name the outlets `nameField`, `phoneNumberField`, and `postcode-Field`, respectively.

5. Create an action method called `onAdd` in the view controller class and connect it to the Touch Up Inside event of the Add New Record To Database button.
6. Create an outlet in the view controller class corresponding to the Table View object in the scene. Name the outlet `tableOfContacts`.
7. Select the table view in the scene. Use the Assistant Editor to set its content type to Dynamic Prototypes and the number of prototype cells for the table view to 1.
8. Select the prototype cell within the table view. Use the Assistant Editor to set the table view cell style to Basic and the Identifier to `ContactDataTableViewCellIdentifier`.

- Setup constraints in the default scene.
  1. Select the New Contact label.
     - Select Editor ➪ Size To Fit Contents. This will ensure the size of the label is precisely what is needed to show all its contents.
     - Select Editor ➪ Align ➪ Horizontal Center in Container to center this label horizontally in the scene.
     - Ensure the label is selected and bring up the Pin Constraints dialog box.
     - Ensure the Constrain to Margins option is unchecked.
     - Pin the distance from the top of the label to the view to 20.
     - Pin the width of the label.
     - Pin the height of the label.
  2. Select the Name label and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
     - **Left:** 31
     - **Top:** 37
     - **Width:** 46
     - **Height:** 21
  3. Select the Phone number label and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
     - **Left:** 31
     - **Top:** 18
     - **Width:** 113
     - **Height:** 21
  4. Select the Postcode label and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
     - **Left:** 31
     - **Top:** 18

- **Width:** 79
- **Height:** 21

5. Select the Name text field and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
   - **Left:** 114
   - **Top:** 12
   - **Right:** 26
   - **Height:** 30
6. Select the Phone number text field and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
   - **Left:** 47
   - **Top:** 10
   - **Right:** 26
   - **Height:** 30
7. Select the Postcode text field and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
   - **Left:** 81
   - **Top:** 8
   - **Right:** 26
   - **Height:** 30
8. Select the Add New Record To Database button and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
   - **Top:** 13
   - **Right:** 26
   - **Width:** 236
   - **Height:** 37
9. Select the Existing Contacts label.
   - Select Editor ➪ Size To Fit Contents. This will ensure the size of the label is precisely what is needed to show all its contents.
   - Select Editor ➪ Align ➪ Horizontal Center in Container to center this label horizontally in the scene.
   - Ensure the label is selected and bring up the Pin Constraints dialog box.
   - Ensure the Constrain to Margins option is unchecked.

- Pin the distance from the top of the label to the view to 18.
- Pin the width of the label.
- Pin the height of the label.

10. Select the table view and use the Pin Constraints dialog box to set up the following constraints while ensuring the Constrain to Margins option is unchecked.
    - **Left:** 31
    - **Top:** 13
    - **Right:** 26
    - **Bottom:** 15
11. Update the frames to match the constraints you have set.
    - Click on the View controller item in the dock above the storyboard scene. This is the first of the three icons located directly above the selected storyboard scene.
    - Select Editor ➪ Resolve Auto Layout Issues ➪ Update Frames.

- Create a managed object in the data store when the Add New Record To Database button is tapped.

1. Import the `CoreData` header files at the top of the `ViewController.swift` file by adding this line:

```
import CoreData
```

2. Update the implementation of the `onAdd` method to the following:

```
@IBAction func onAdd(sender: AnyObject) {

        nameField.resignFirstResponder()
        phoneNumberField.resignFirstResponder()
        postCodeField.resignFirstResponder()

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

        let newCustomerName:String! = nameField.text
        let newCustomerPhoneNumber:String! = phoneNumberField.text
        let newCustomerPostcode:String! = postCodeField.text

        if newCustomerName.isEmpty &&
           newCustomerPhoneNumber.isEmpty &&
           newCustomerPostcode.isEmpty
        {
            return
        }

        let newItem =
         NSEntityDescription.insertNewObjectForEntityForName(
        "ContactData",
        inManagedObjectContext: appDelegate.managedObjectContext)
        as! ContactData

         newItem.customerName = newCustomerName
         newItem.phoneNumber = newCustomerPhoneNumber
         newItem.postCode = newCustomerPostcode

         var error:NSError? = nil
         appDelegate.managedObjectContext!.save(&error)

         fetchExistingContacts()
         tableOfContacts.reloadData()

    }
```

➤ Read managed objects from the database and display them in a table view.

1. Ensure the `ViewController` class implements the `UITableViewDataSource` and `UITableViewDelegate` protocols by changing its declaration to the following:

```
class ViewController: UIViewController,
                      UITableViewDataSource,
                      UITableViewDelegate
```

2. Add the following variable declaration to the `ViewController.swift` file:

```
var listOfContacts:Array<ContactData>? = nil
```

3. Create a new method in the `ViewController.swift` file called `fetchExisting ContactData` as follows:

```
 func fetchExistingContacts()
     {
     let fetchRequest = NSFetchRequest(entityName: "ContactData")

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

     do {
          self.listOfContacts = try
          appDelegate.managedObjectContext.executeFetchRequest
          (fetchRequest) as? [ContactData]
         } catch {
           // handle errors here.
         }
     }
```

4. Add the following lines of code to the end of the `viewDidLoad` method. These lines set up the `datasource` and `delegate` properties of the table view object and call the `fetchExistingContactData` method.

```
       fetchExistingContacts()
       tableOfContacts.dataSource = self
       tableOfContacts.delegate = self
```

5. Implement `UITableViewDataSource` and `UITableViewDelegate` methods in the `ViewController.swift` file as follows:

```
func tableView(tableView: UITableView,
               numberOfRowsInSection section: Int)
               -> Int
    {
        return listOfContacts!.count;
    }

    func tableView(tableView: UITableView,
         cellForRowAtIndexPath indexPath: NSIndexPath)
         -> UITableViewCell
    {
       let cell = tableView.dequeueReusableCellWithIdentifier(
       "ContactDataTableViewCellIdentifier",
        forIndexPath: indexPath)

        var someContactData:ContactData! =
            listOfContacts![indexPath.row]

        cell.textLabel?.text = someContactData.customerName

        return cell
    }
```

➤ Test your app in the iOS Simulator.

Click the Run button in the Xcode toolbar. Alternatively, you can select Project ➪ Run.

REFERENCE *To see some of the examples from this lesson, watch the Lesson 26 video online at* `www.wrox.com/go/swiftiosvid`.