Persist Data

This lesson is focused on saving a meal list across FoodTracker app sessions. Data persistence is one of the most important and common problems in iOS app development. iOS has many persistent data storage solutions; in this lesson, you'll use `NSCoding` as the data persistence mechanism in the FoodTracker app. `NSCoding` is a protocol that enables a lightweight solution for archiving objects and other structures. Archived objects can be stored on disk and retrieved at a later time.

Learning Objectives

At the end of the lesson, you'll be able to:

- Create a structure
- Understand the difference between static properties and instance properties
- Use the `NSCoding` protocol to read and write data

Save and Load the Meal

In this step you'll implement the behavior in the `Meal` class to save and load the meal. Using the `NSCoding` approach, the `Meal` class is in charge of storing and loading each of its properties. It needs to save its data by assigning the value of each property to a particular key, and load the data by looking up the information associated with that key.

A key is simply a string value. You choose your own keys based on what makes the most sense in your app. For example, you might use the key "name" to store the value of the `name` property.

To make it clear which coding key corresponds to each piece of data, create a structure to store the key strings. This way, when you need to use the keys in multiple places throughout your code, you can use the constants instead of retyping the strings (which increases the likelihood of mistakes).

To implement a coding key structure

1. Open `Meal.swift`.
2. In `Meal.swift`, below the `// MARK: Properties` section, add this structure:

    ```
    // MARK: Types

    struct PropertyKey {
    }
    ```

3. In the `PropertyKey` structure, add these properties:

    ```
    static let nameKey = "name"
    static let photoKey = "photo"
    static let ratingKey = "rating"
    ```

    Each constant corresponds to one of the three properties on `Meal`. The `static` keyword indicates that this constant applies to the structure itself, not an instance of the structure. These values will never change.

Your `PropertyKey` structure should look like this:

```
struct PropertyKey {
    static let nameKey = "name"
    static let photoKey = "photo"
    static let ratingKey = "rating"
}
```

To be able to encode and decode itself and its properties, the `Meal` class needs to conform to to the `NSCoding` protocol. To conform to `NSCoding`, the `Meal` needs to subclass `NSObject`. `NSObject` is considered a base class that defines a basic interface to the runtime system.

To subclass NSObject and conform to NSCoding

1. In `Meal.swift`, find the `class` line:

   ```
   class Meal {
   ```

2. After `Meal`, add a colon (`:`) and `NSObject` to subclass from the `NSObject` class:

   ```
   class Meal: NSObject {
   ```

3. After `NSObject`, add a comma (`,`) and `NSCoding` to adopt the `NSCoding` protocol:

   ```
   class Meal: NSObject, NSCoding {
   ```

The `NSCoding` protocol declares two methods that any class that adopts to it must implement so that instances of that class can be encoded and decoded:

```
func encodeWithCoder(aCoder: NSCoder)
init(coder aDecoder: NSCoder)
```

The `encodeWithCoder(_:)` method prepares the class's information to be archived, and the initializer unarchives the data when the class is created. You need to implement both the `encodeWithCoder(_:)` method and the initializer for the data to save and load properly.

To implement the encodeWithCoder NSCoding method

1. In `Meal.swift`, before the last curly brace (`}`), add the following:

   ```
   // MARK: NSCoding
   ```

   This is a comment to help you (and anybody else who reads your code) know that the code in this section is related to data persistence.
2. Below the comment, add this method:

   ```
   func encodeWithCoder(aCoder: NSCoder) {
   }
   ```

3. In the `encodeWithCoder(_:)` method, add the following code:

   ```
   aCoder.encodeObject(name, forKey: PropertyKey.nameKey)
   aCoder.encodeObject(photo, forKey: PropertyKey.photoKey)
   aCoder.encodeInteger(rating, forKey: PropertyKey.ratingKey)
   ```

   The `encodeObject(_:forKey:)` method encodes any type of object, while the `encodeInteger(_:forKey:)` method encodes an integer. These lines of code encode the value of each property on the `Meal` class and store them with their corresponding key.

The `encodeWithCoder(_:)` method should look like this:

```
func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(name, forKey: PropertyKey.nameKey)
    aCoder.encodeObject(photo, forKey: PropertyKey.photoKey)
    aCoder.encodeInteger(rating, forKey: PropertyKey.ratingKey)
}
```

With the encoding method written, implement the initializer to decode the encoded data.

To implement the initializer to load the meal

1. Below the `encodeWithCoder(_:)` method, add the following initializer:

    ```
    required convenience init?(coder aDecoder: NSCoder) {
    }
    ```

    The `required` keyword means this initializer must be implemented on every subclass of the class that defines this initializer.

    The `convenience` keyword denotes this initializer as a convenience initializer. Convenience initializers are secondary, supporting initializers that need to call one of their class's designated initializers. Designated initializers are the primary initializers for a class. They fully initialize all properties introduced by that class and call a superclass initializer to continue the initialization process up the superclass chain. Here, you're declaring this initializer as a convenience initializer because it only applies when there's saved data to be loaded.

    The question mark (`?`) means that this is a failable initializer that might return nil.

2. Add the following line of code:

    ```
    let name = aDecoder.decodeObjectForKey(PropertyKey.nameKey) as! String
    ```

    The `decodeObjectForKey(_:)` method unarchives the stored information stored about an object.

    The return value of `decodeObjectForKey(_:)` is `AnyObject`, which you downcast in the code above as a `String` to assign it to a `name` constant. You downcast the return value using the forced type cast operator (`as!`) because if the object can't be cast as a `String`, or if it's `nil`, something has gone wrong and the error should cause a crash at runtime.

3. Below the previous line, add the following code:

    ```
    // Because photo is an optional property of Meal, use conditional cast.
    let photo = aDecoder.decodeObjectForKey(PropertyKey.photoKey) as? UIImage
    ```

    You downcast this return value of `decodeObjectForKey(_:)` as a `UIImage` to be assigned to a `photo` constant. In this case, you downcast using the optional type cast operator (`as?`), because the photo property is an optional, so the value might be a `UIImage`, or it might be `nil`. You need to account for both cases.

4. Below the previous line, add the following line of code:

    ```
    let rating = aDecoder.decodeIntegerForKey(PropertyKey.ratingKey)
    ```

    The `decodeIntegerForKey(_:)` method unarchives an integer. Because the return value of `decodeIntegerForKey` is `Int`, there's no need to downcast the decoded value.

5. Add the following code at the end of the implementation:

    ```
    // Must call designated initializer.
    self.init(name: name, photo: photo, rating: rating)
    ```

    As a convenience initializer, this initializer is required to call one of its class's designated initializers before completing. As the initializer's arguments, you pass in the values of the constants you created while archiving the saved data.

The new `init?(coder:)` initializer should look like this:

```
required convenience init?(coder aDecoder: NSCoder) {
    let name = aDecoder.decodeObjectForKey(PropertyKey.nameKey) as! String

    // Because photo is an optional property of Meal, use conditional cast.
    let photo = aDecoder.decodeObjectForKey(PropertyKey.photoKey) as? UIImage

    let rating = aDecoder.decodeIntegerForKey(PropertyKey.ratingKey)

    // Must call designated initializer.
    self.init(name: name, photo: photo, rating: rating)
}
```

Because the other initializer you defined on the `Meal` class, `init?(name:photo:rating:)`, is a designated initializer, its implementation needs to call to its superclass's initializer.

To update the initializer implementation to call its superclass initializer

1. Find the initializer that looks like this:

```
init?(name: String, photo: UIImage?, rating: Int) {
    // Initialize stored properties.
    self.name = name
    self.photo = photo
    self.rating = rating

    // Initialization should fail if there is no name or if the rating is
  negative.
    if name.isEmpty || rating < 0 {
        return nil
    }
}
```

2. Below the `self.rating = rating` line, add a call to the superclass's initializer.

```
super.init()
```

The `init?(name:photo:rating:)` initializer should look like this:

```
init?(name: String, photo: UIImage?, rating: Int) {
    // Initialize stored properties.
    self.name = name
    self.photo = photo
    self.rating = rating

    super.init()

    // Initialization should fail if there is no name or if the rating is negative.
    if name.isEmpty || rating < 0 {
        return nil
    }
}
```

Next, you need a persistent path on the file system where data will be saved and loaded, so you know where to look for it.

To create a file path to data

- In `Meal.swift`, below the `// MARK: Properties` section, add this code:

```
// MARK: Archiving Paths

static let DocumentsDirectory =
  NSFileManager().URLsForDirectory(.DocumentDirectory, inDomains:
  .UserDomainMask).first!
static let ArchiveURL =
  DocumentsDirectory.URLByAppendingPathComponent("meals")
```

  You mark these constants with the `static` keyword, which means they apply to the class instead of an instance of the class. Outside of the `Meal` class, you'll access the path using the syntax `Meal.ArchiveURL.path!`.

Checkpoint: Build your app using Command-B. It should build without issues.

Save and Load the Meal List

Now that you can save and load an individual meal, you need to save and load the meal list whenever a user adds, edits, or removes a meal.

To implement the method to save the meal list

1. Open `MealTableViewController.swift`.
2. In `MealTableViewController.swift`, before the last curly brace (}), add the following:

   ```
   // MARK: NSCoding
   ```

   This is a comment to help you (and anybody else who reads your code) know that the code in this section is related to data persistence.
3. Below the comment, add the following method:

   ```
   func saveMeals() {
   }
   ```

4. In the `saveMeals()` method, add the following line of code:

   ```
   let isSuccessfulSave = NSKeyedArchiver.archiveRootObject(meals, toFile:
     Meal.ArchiveURL.path!)
   ```

   This method attempts to archive the `meals` array to a specific location, and returns `true` if it's successful. It uses the constant `Meal.ArchiveURL` that you defined in the `Meal` class to identify where to save the information.

   But how do you quickly test whether the data saved successfully? Use `print` to print a message to the console. For example, print a failure message if the meals fail to save successfully.
5. Below the previous line, add the following `if` statement:

   ```
   if !isSuccessfulSave {
       print("Failed to save meals...")
   }
   ```

   Now, if a save fails, you'll see a message printed in the console.

Your `saveMeals()` method should look like this:

```
func saveMeals() {
    let isSuccessfulSave = NSKeyedArchiver.archiveRootObject(meals, toFile:
  Meal.ArchiveURL.path!)
    if !isSuccessfulSave {
        print("Failed to save meals...")
    }
}
```

Now, implement a method to load saved meals.

To implement the method to load the meal list

1. In `MealTableViewController.swift`, before the last curly brace (}), add the following method:

   ```
   func loadMeals() -> [Meal]? {
   }
   ```

   This method has a return type of an optional array of Meal objects, meaning that it might return an array of Meal objects or might return nothing (`nil`).
2. In the `loadMeals()` method, add the following line of code:

   ```
   return NSKeyedUnarchiver.unarchiveObjectWithFile(Meal.ArchiveURL.path!) as?
   ```

```
[Meal]
```

This method attempts to unarchive the object stored at the path `Meal.ArchiveURL.path!` and downcast that object to an array of `Meal` objects. This code uses the `as?` operator so that it can return `nil` when appropriate. Because the array may or may not have been stored, it's possible that the downcast will fail, in which case the method should return `nil`.

Your `loadMeals()` method should look like this:

```
func loadMeals() -> [Meal]? {
    return NSKeyedUnarchiver.unarchiveObjectWithFile(Meal.ArchiveURL.path!) as?
  [Meal]
}
```

With these methods implemented, you need to add code to save and load the list of meals whenever a user adds, removes, or edits a meal.

To save the meal list when a user adds, removes, or edits a meal

1. In `MealTableViewController.swift`, find the `unwindToMealList(_:)` action method:

```
@IBAction func unwindToMealList(sender: UIStoryboardSegue) {
    if let sourceViewController = sender.sourceViewController as?
  MealViewController, meal = sourceViewController.meal {
        if let selectedIndexPath = tableView.indexPathForSelectedRow {
            // Update an existing meal.
            meals[selectedIndexPath.row] = meal
            tableView.reloadRowsAtIndexPaths([selectedIndexPath],
  withRowAnimation: .None)
        }
        else {
            // Add a new meal.
            let newIndexPath = NSIndexPath(forRow: meals.count, inSection: 0)
            meals.append(meal)
            tableView.insertRowsAtIndexPaths([newIndexPath], withRowAnimation:
  .Bottom)
        }
    }
}
```

2. Right after the `else` clause, add the following code:

```
// Save the meals.
saveMeals()
```

   This code saves the `meals` array whenever a new one is added or an existing one is updated. Make sure this line of code is inside of the outer `if` statement.

3. In `MealTableViewController.swift`, find the `tableView(_:commitEditingStyle:forRowAtIndexPath:)` method:

```
// Override to support editing the table view.
override func tableView(tableView: UITableView, commitEditingStyle
  editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath:
  NSIndexPath) {
    if editingStyle == .Delete {
        // Delete the row from the data source
        meals.removeAtIndex(indexPath.row)
        tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
    } else if editingStyle == .Insert {
        // Create a new instance of the appropriate class, insert it into the
  array, and add a new row to the table view
```

```
    }
}
```

4. After the `meals.removeAtIndex(indexPath.row)` line, add the following line of code:

   ```
   saveMeals()
   ```

   This code saves the `meals` array whenever a meal is deleted.

Your `unwindToMealList(_:)` action method should look like this:

```
@IBAction func unwindToMealList(sender: UIStoryboardSegue) {
    if let sourceViewController = sender.sourceViewController as?
  MealViewController, meal = sourceViewController.meal {
        if let selectedIndexPath = tableView.indexPathForSelectedRow {
            // Update an existing meal.
            meals[selectedIndexPath.row] = meal
            tableView.reloadRowsAtIndexPaths([selectedIndexPath], withRowAnimation:
  .None)
        }
        else {
            // Add a new meal.
            let newIndexPath = NSIndexPath(forRow: meals.count, inSection: 0)
            meals.append(meal)
            tableView.insertRowsAtIndexPaths([newIndexPath], withRowAnimation:
  .Bottom)
        }
        // Save the meals.
        saveMeals()
    }
}
```

And your `tableView(_:commitEditingStyle:forRowAtIndexPath:)` method should look like this:

```
// Override to support editing the table view.
override func tableView(tableView: UITableView, commitEditingStyle editingStyle:
  UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == .Delete {
        // Delete the row from the data source
        meals.removeAtIndex(indexPath.row)
        saveMeals()
        tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
    } else if editingStyle == .Insert {
        // Create a new instance of the appropriate class, insert it into the
  array, and add a new row to the table view
    }
}
```

Now that meals are saved at the appropriate times, you need to make sure that meals get loaded at the appropriate time. This should happen every time the meal list scene loads, which means the appropriate place to load the stored data is in `viewDidLoad`.

To load the meal list at the appropriate time

1. In `MealTableViewController.swift`, find the `viewDidLoad()` method:

   ```
   override func viewDidLoad() {
       super.viewDidLoad()

       // Use the edit button item provided by the table view controller.
       navigationItem.leftBarButtonItem = editButtonItem()
   ```

```swift

    // Load the sample data.
    loadSampleMeals()
}
```

2. After the second line of code (`navigationItem.leftBarButtonItem = editButtonItem()`), add the following `if` statement:

   ```swift
   // Load any saved meals, otherwise load sample data.
   if let savedMeals = loadMeals() {
       meals += savedMeals
   }
   ```

   If `loadMeals()` successfully returns an array of `Meal` objects, this condition is `true` and the `if` statement gets executed. If `loadMeals()` returns `nil`, there were no meals to load and the `if` statement doesn't get executed. This code adds any meals that were successfully loaded to the `meals` array.

3. After the if statement, add an `else` clause and move the call to `loadSampleMeals()` inside of it:

   ```swift
   else {
       // Load the sample data.
       loadSampleMeals()
   }
   ```

   This code adds any meals that were loaded to the `meals` array.

Your `viewDidLoad()` method should look like this:

```swift
override func viewDidLoad() {
    super.viewDidLoad()

    // Use the edit button item provided by the table view controller.
    navigationItem.leftBarButtonItem = editButtonItem()

    // Load any saved meals, otherwise load sample data.
    if let savedMeals = loadMeals() {
        meals += savedMeals
    } else {
        // Load the sample data.
        loadSampleMeals()
    }
}
```

Checkpoint: Run your app. If you add a few new meals and quit the app, the meals you added will be there next time you open the app.

NOTE

To see the completed sample project for this lesson, download the file and view it in Xcode.

Download File