Saving and Loading Application States-2

There are many ways to save and load data in an iOS application. In this assignment we will incorporate some of the most common mechanisms by savings some data from our Home Owner app to the file system so we can reload it when the app is ran again.

1. Create a new repository on GitHub, add your instructors as collaborators, then open your Home Owner project.

2. In Xcode, open the source control navigator, expand the remotes section, then remove your old repository remote and add your new repository remote. Perform a push to ensure all your files are pushed to your new remote.

3. Your Item class does not currently conform to NSCoding and we need it to so we can save the data stored in our class to the file system. Add this protocol declaration in Item.swift.

   class Item: NSObject**, NSCoding** {

4. The next step is to implement the required methods. Let's start with encode(with:). When encode(with:) is called on an Item, it will encode all of its properties into the NSCoder object that is passed as an argument. While saving, you will use NSCoder to write out a stream of data. That stream will be organized as key-value pairs and stored on the filesystem. In Item.swift, implement encode(with:) to add the names and values of the item's properties to the stream.

   ```
   func encode(with aCoder: NSCoder) {
      aCoder.encode(name, forKey: "name")
      aCoder.encode(dateCreated, forKey: "dateCreated")
      aCoder.encode(itemKey, forKey: "itemKey")
      aCoder.encode(serialNumber, forKey: "serialNumber")
      aCoder.encode(valueInDollars, forKey: "valueInDollars")
   }
   ```

5. The purpose of the key is to retrieve the encoded value when this Item is loaded from the file system later. Objects being loaded from an archive are sent the message init(coder:). This method should grab all of the objects that were encoded in encode(with:) and assign them to the appropriate property. In Item.swift, implement init(coder:).

   ```
   required init(coder aDecoder: NSCoder) {
      name = aDecoder.decodeObject(forKey: "name") as! String
      dateCreated = aDecoder.decodeObject(forKey: "dateCreated") as! Date
      serialNumber = aDecoder.decodeObject(forKey: "serialNumber") as! String?
      itemKey = aDecoder.decodeObject(forKey: "itemKey") as! String
      valueInDollars = aDecoder.decodeInteger(forKey: "valueInDollars")

      super.init()
   }
   ```

6. The instances of Item will be saved to a single file in the Documents directory. The ItemStore will handle writing to and reading from that file. To do this, the ItemStore needs to construct a URL to this file. Implement a new property in ItemStore.swift to store this URL.

```
var allItems = [Item]()
let itemArchiveURL: URL = {
  let documentsDirectories =
    FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
  let documentDirectory = documentsDirectories.first!
  return documentDirectory.appendingPathComponent("items.archive")
}()
```

7. You now have a place to save data on the filesystem and a model object that can be saved to the filesystem. The final two questions are: How do you kick off the saving and loading processes, and when do you do it? To save instances of Item, you will use the class NSKeyedArchiver when the application "exits." In ItemStore.swift, implement a new method that calls archiveRootObject(_:toFile:) on the NSKeyedArchiver class.

```
func saveChanges() -> Bool {
  print("Saving items to: \(itemArchiveURL.path)")
  return NSKeyedArchiver.archiveRootObject(allItems, toFile: itemArchiveURL.path)
}
```

8. When the user presses the Home button on the device, the message applicationDidEnterBackground(_:) is sent to the AppDelegate. That is when you want to send saveChanges to the ItemStore. Open AppDelegate.swift and add a property to the class to store the ItemStore instance. You will need a property to reference the instance in applicationDidEnterBackground(_:).

```
class AppDelegate: UIResponder, UIApplicationDelegate {

  var window: UIWindow?
  let itemStore = ItemStore()
```

9. Then update application(_:didFinishLaunchingWithOptions:) to use this property instead of the local constant (because the property we created above and the local constant shown below are named the same, you only needed to remove the code that created the local constant as seen below).

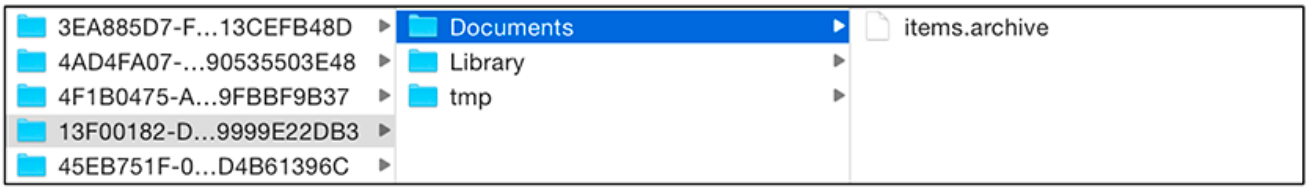
```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
      launchOptions: [UIApplicationLaunchOptionsKey : Any]?) -> Bool {
   // Override point for customization after application launch.

   // Create an ItemStore
   let itemStore = ItemStore()

   // Access the ItemsViewController and set its item store and image store
   let navController = window!.rootViewController as! UINavigationController
   let itemsController = navController.topViewController as! ItemsViewController
   itemsController.itemStore = itemStore

   return true
}
```

10. Now, still in AppDelegate.swift, implement applicationDidEnterBackground(_:) to kick off saving the Item instances.

```
func applicationDidEnterBackground(_ application: UIApplication) {
   let success = itemStore.saveChanges()
   if (success) {
      print("Saved all of the Items")
   } else {
      print("Could not save any of the Items")
   }
}
```

11. Build and run the application on the simulator. Create a few items, then press the Home button to leave the application. Check the console and you should see a log statement indicating that the items were saved. While you cannot yet load these instances of Item back into the application, you can still verify that something was saved. In the console's log statements, find one that logs out the itemArchiveURL location and another that indicates whether saving was successful. If saving was not successful, confirm that your itemArchiveURL is being created correctly. If the items were saved successfully, copy the path that is printed to the console.

12. Open Finder and press Command-Shift-G. Paste the file path that you copied from the console and press Return. You will be taken to the directory that contains the items.archive file. Press Command-Up to navigate to the parent directory of items.archive. This is the application's sandbox directory. Here, you can see the Documents, Library, and tmp directories alongside the application itself (Figure 16.5).



13. Note: the location of the sandbox directory can change between runs of the application; however, the contents of the sandbox will remain unchanged. Due to this, you may need to copy and paste the directory into Finder frequently while working on an application.

14. Now let's turn to loading these files. To load instances of Item when the application launches, you will use the class NSKeyedUnarchiver when the ItemStore is created. In ItemStore.swift, override init() to add the following code.

```
init() {
    if let archivedItems =
        NSKeyedUnarchiver.unarchiveObject(withFile: itemArchiveURL.path) as? [Item] {
        allItems = archivedItems
    }
}
```

15. Build and run the application. Your items will be available until you explicitly delete them. One thing to note about testing your saving and loading code: If you kill Homepwner from Xcode, the method applicationDidEnterBackground(_:) will not get a chance to be called and the item array will not be saved. You must press the Home button first and then kill it from Xcode by clicking the Stop button.

16. Image data is not part of our Item class. Image data should be stored in the Documents directory. You can use a file name to name the image in the filesystem. Implement a new method in ImageStore.swift named imageURL() to create a URL in the documents directory.

```
func imageURL(forKey key: String) -> URL {
    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
    let documentDirectory = documentsDirectories.first!

    return documentDirectory.appendingPathComponent(key)
}
```

17. To save and load an image, you are going to copy the JPEG representation of the image into a buffer in memory. Instead of just creating a buffer, Swift programmers have a handy class to create, maintain, and destroy these sorts of buffers – Data. A Data instance holds some number of bytes of binary data, and you will use Data to store image data. In ImageStore.swift, add a setImage() to get a URL and save the image.

```
func setImage(_ image: UIImage, forKey key: String) {
    cache.setObject(image, forKey: key as NSString)

    // Create full URL for image
    let url = imageURL(forKey: key)

    // Turn image into JPEG data
    if let data = UIImageJPEGRepresentation(image, 0.5) {
        // Write it to full URL
        let _ = try? data.write(to: url, options: [.atomic])
    }
}
```

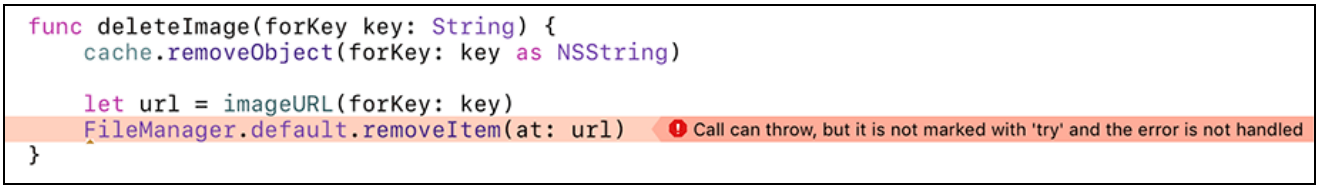
18. Now that the image is stored in the filesystem, the ImageStore will need to load that image when it is requested. The initializer init(contentsOfFile:) of UIImage will read in an image from a file, given a URL. In ImageStore.swift, add a method image() so that the ImageStore will load the image from the filesystem.

```
func image(forKey key: String) -> UIImage? {
    return cache.object(forKey: key as NSString)

    if let existingImage = cache.object(forKey: key as NSString) {
        return existingImage
    }

    let url = imageURL(forKey: key)
    guard let imageFromDisk = UIImage(contentsOfFile: url.path) else {
        return nil
    }

    cache.setObject(imageFromDisk, forKey: key as NSString)
    return imageFromDisk
}
```

19. In ImageStore.swift, add a method to delete an image from the filesystem. (You will see an error when you type in this code, which we will discuss next.)

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

    let url = imageURL(forKey: key)
    FileManager.default.removeItem(at: url)
}
```

20. Let's take a look at the error message that this code generated.



21. This error message is letting you know that the method removeItem(at:) can fail, but you are not handling the error. Let's fix this. In ImageStore.swift, update deleteImage() to call removeItem(at:) using a do-catch statement.

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

    let url = imageURL(forKey: key)
    ~~FileManager.default.removeItem(at: url)~~
    do {
        try FileManager.default.removeItem(at: url)
    } catch {
    }
}
```

22. If a method does throw an error, then the program immediately exits the do block; no further code in the do block is executed. At that point, the error is passed to the catch block for it to be handled in some way. Now, update deleteImage() to print out the error to the console.

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)
    let url = imageURL(forKey: key)
    do {
        try FileManager.default.removeItem(at: url)
    } catch {
        print("Error removing the image from disk: \(error)")
    }
}
```

23. Within the catch block, there is an implicit error constant that contains information describing the error. You can optionally give this constant an explicit name. Update deleteImage() again to use an explicit name for the error being caught.

```
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)

    let url = imageURL(forKey: key)
    do {
        try FileManager.default.removeItem(at: url)
    } catch let deleteError {
        print("Error removing the image from disk: \(~~error~~deleteError)")
    }
}
```

24. There is a lot more that you can do with error handling, but this is the basic knowledge that you need for now. We will cover more details as you progress through this course. Build and run the application now that the ImageStore is complete. Open your photo screen and take a photo. Exit the application to the Home screen (on the simulator, select Hardware → Home or press Shift-Command-H; on a hardware device simply press the Home button). Launch the application again and open the photo screen, you can now see your photo saved/loaded properly