

# iOS: Saving, Loading, and Application States

BNRG CHAPTER 16

# Topics

Persistent data

- ▶ encoding and decoding
- ▶ saving raw data

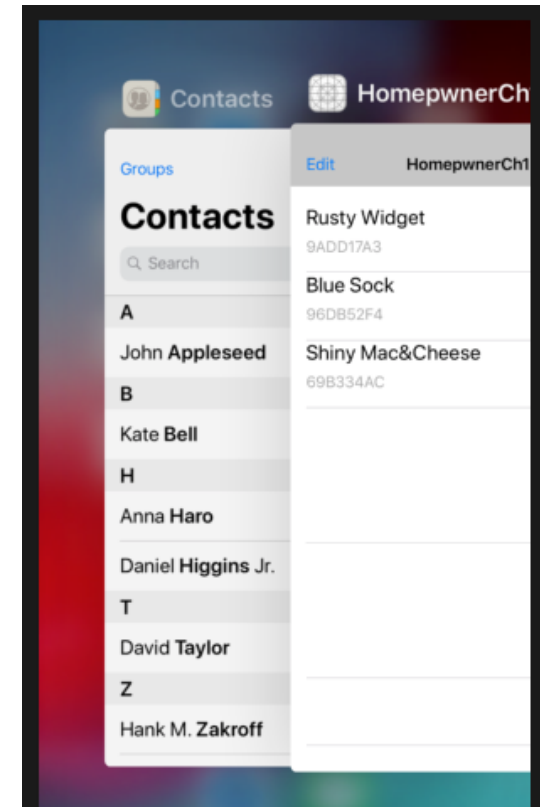
Device filesystem

Application states and transitions

# Persistent Data

Most apps need to save data between runs of the app

This data must go out to the flash storage of the device



# Archiving

Most applications essentially just provide an interface for the user to manipulate data

And so applications need to retain their "model" data between separate runs of the application

- ▶ for Homepwner, the model data is instances of `Item`
- ▶ for Homepwner to be useful, we want to save the `Item` instances after the application quits
- ▶ and if there is an image associated with an `Item`, then we want to save the image as well
- ▶ think of the Contacts on the phone: Contacts would not be very useful if we had to add back contacts every time we launched the app

# Archiving

A common way of persisting model objects in iOS is through the *archiving* mechanism

- ▶ archiving an object consists of recording the value of all of its properties and then saving them to the filesystem

Unarchiving does the opposite: reads data from the filesystem and restores that data into values for the properties of the object

- ▶ In particular, the storyboard itself is unarchived and archived when an app runs/ends

This is analogous to serialization in Java or "pickling" in Python

- ▶ turn an instance of an object into a stream of bytes that can be written and then read back in

# Archiving

Classes whose instances need to be archived and unarchived must conform to the `NSCoding` protocol and implement two methods:

- ▶ `encode(with:)`
- ▶ `init?(coder:)`

Here are the interfaces:

```
protocol NSCoding {  
    func encode(with aCoder: NSCoder)  
    init?(coder aDecoder: NSCoder)  
}
```

**NSCoder:** a protocol that provides an interface for saving data from memory to storage; and vice-versa

# Archiving

`UIView` and `UIViewController` both confirm to the `NSCoding` protocol

- ▶ But the user class `Item` does not (not yet)
- ▶ to do so, we must first implement `encode(with aCoder: NSCoder)`

Basic idea: explicitly encode each property, using a `String` key

- ▶ by convention, the key is the name of the property

# Archiving

So for `Item`, it looks like this:

```
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")  
}
```

This will encode an instance of an item when we trigger the "save"



# Encoding

Encoding is creating a stream of data that consists of key-value pairs

- ▶ encoding is recursive: if you encode an instance, that instance is sent `encode(with:)`
- ▶ and that instance must encode all of its properties using `encode(_: forKey:)`

```
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")  
}
```

# Unarchiving

Unarchiving is the reverse process:

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

This will build an instance of Item when we trigger the "load"

- ▶ but this is separate from the normal initialization mechanism for Item (the designated initializer)
- ▶ the keyword **required** means that any subclass of this class must implement this initializer
- ▶ next thing will be to specify a place in the filesystem for saving and loading

# Application Sandbox

Every iOS application has its own application sandbox

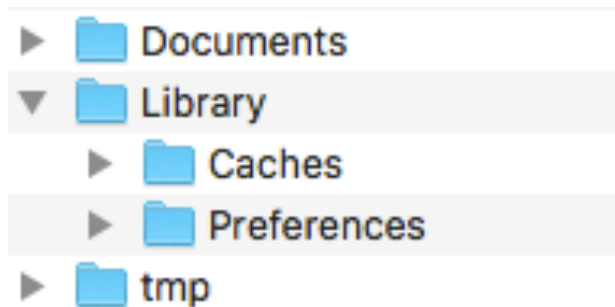
- ▶ this is a directory (folder) in the filesystem that is walled off from the rest of the filesystem

Each application must stay in its own sandbox

And an application cannot access the sandbox of any other application

# Application Sandbox

Here's an application's sandbox:



Viewing the Simulator's filesystem:

<https://stackoverflow.com/questions/6480607/is-there-any-way-to-see-the-file-system-on-the-ios-simulator>

# Application Sandbox

## Documents/

- ▶ this is where the app will write data during runtime, including data that we want to persist between runs of the app
- ▶ data in this directory is backed up when the device is synchronized with iCloud or iTunes
- ▶ if something goes wrong with the device, files in this directory can be restored from iTunes or iCloud
- ▶ for Homepwner, we'll store Item data here

# Application Sandbox

## Library/Caches/

- ▶ this is where the app will write data during runtime, including data that we want to persist between runs of the app
- ▶ this directory is not sync'ed and backed up
- ▶ this directory is meant for large, semi-transient data from an app that does not need to be backed up
- ▶ it's most effective for saving data in the app that is also saved somewhere else (such as on a web server)

# Application Sandbox

Library/Preferences/

- ▶ this is where app preferences are stored
- ▶ and this is where Settings looks for application preferences
- ▶ data in this directory is handled automatically by the class `NSUserDefaults`
- ▶ and it is backed up with the device is sync'd to iTunes or iCloud

# Application Sandbox

`tmp/`

- ▶ this directory is for true scratch data that the app loads and saves during a run
- ▶ the OS may purge files in this directory when your app is not running
- ▶ data in this directory is not backed up
- ▶ it is good practice for an app to clean up its data in this directory



# Constructing a File URL

In order to access a specific file in the filesystem of the device:

- ▶ we have to construct a URL that specifies it

URL: uniform resource locator

- ▶ technically, this is a URI (uniform resource identifier)
- ▶ URL is a special case of URI
- ▶ but the Foundation framework uses the term URL to refer to URI

# Constructing a File URL

Using a closure to define a URL has these benefits:

- ▶ allows us to set the value for a variable or constant that requires multiple lines of code
- ▶ keeps the property and the code needed to generate the property together

It looks like this:

```
let itemArchiveURL: URL = {  
    let documentDirectories = FileManager.default.urls(for: .documentDirectory,  
                                                         in: .userDomainMask)  
    let documentDirectory = documentDirectories.first!  
    return documentDirectory.appendingPathComponent("items.archive")  
}()
```

# Using a Closure to Set Up a Class/Struct Instance

Another example:

```
struct Thing {  
    var name: String  
    var size: Int  
    var owner: String  
}  
  
var t: Thing = {  
    let n = "cheese"  
    let sz = 45  
    let o = "jason"  
    return Thing(name: n, size: sz, owner: o)  
}()
```

# Saving Data

Now we have a mechanism for saving Item instances and a place to save them

- ▶ a URI and the archiving mechanism

Now we actually have to trigger the save

We will do this when the application exits

- ▶ and so we'll have to do this when the OS notifies the app that it's going to exit
- ▶ during a so-called lifecycle event

# Saving Data

Here's the call that will save the data:

```
NSKeyedArchiver.archiveRootObject(allItems, toFile: itemArchiveURL.path)
```

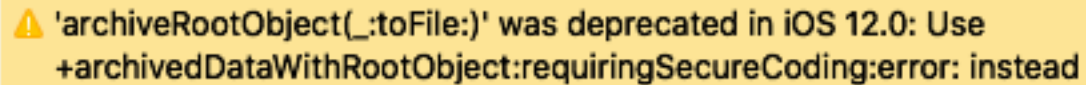
This will call `encode(with:)` on the array `allItems`

- ▶ the array will call `encode(with:)` on each of its elements
- ▶ and so this will call the `encode(with:)` that we implemented in `Item`

What a simple and elegant design! Like all Apple development.

# Compiler Warning

You might see this warning when you build:

A yellow warning banner from Xcode. On the left is a yellow triangle with an exclamation mark icon. The text reads: "'archiveRootObject(\_:toFile:)' was deprecated in iOS 12.0: Use +archivedDataWithRootObject:requiringSecureCoding:error: instead". On the right is a grey circle with a white 'x' icon for closing the warning.

⚠️ 'archiveRootObject(\_:toFile:)' was deprecated in iOS 12.0: Use +archivedDataWithRootObject:requiringSecureCoding:error: instead

If you want, you can use this instead:

```
func saveChanges() -> Bool {  
    do {  
        let data = try NSKeyedArchiver.archivedData(withRootObject: allItems, requiringSecureCoding: false)  
        try data.write(to: itemArchiveURL)  
        return true  
    } catch {  
        return false  
    }  
}
```

# Loading Files

We can put a hook into the initializer of `ItemStore` to load previously saved items from storage

- ▶ the key call is `NSKeyedUnarchiver.unarchiveObject(withFile:)`
- ▶ the code from the book will produce a compiler warning saying that this is a deprecated interface in iOS 13.0
- ▶ Here's the newer call:

```
init() {  
    do {  
        let data = try Data(contentsOf: itemArchiveURL)  
        if let archivedData = try NSKeyedUnarchiver.unarchiveTopLevelObjectWithData(data) as? [Item] {  
            allItems = archivedData  
        }  
    } catch {  
        allItems = []  
    }  
}
```

# Lifecycle Callbacks

Prior to iOS 13.0, the Application Delegate (in `AppDelegate.swift`) handled messages from the OS about changes in the app's status

- ▶ for example, `applicationDidEnterBackground(_:)`

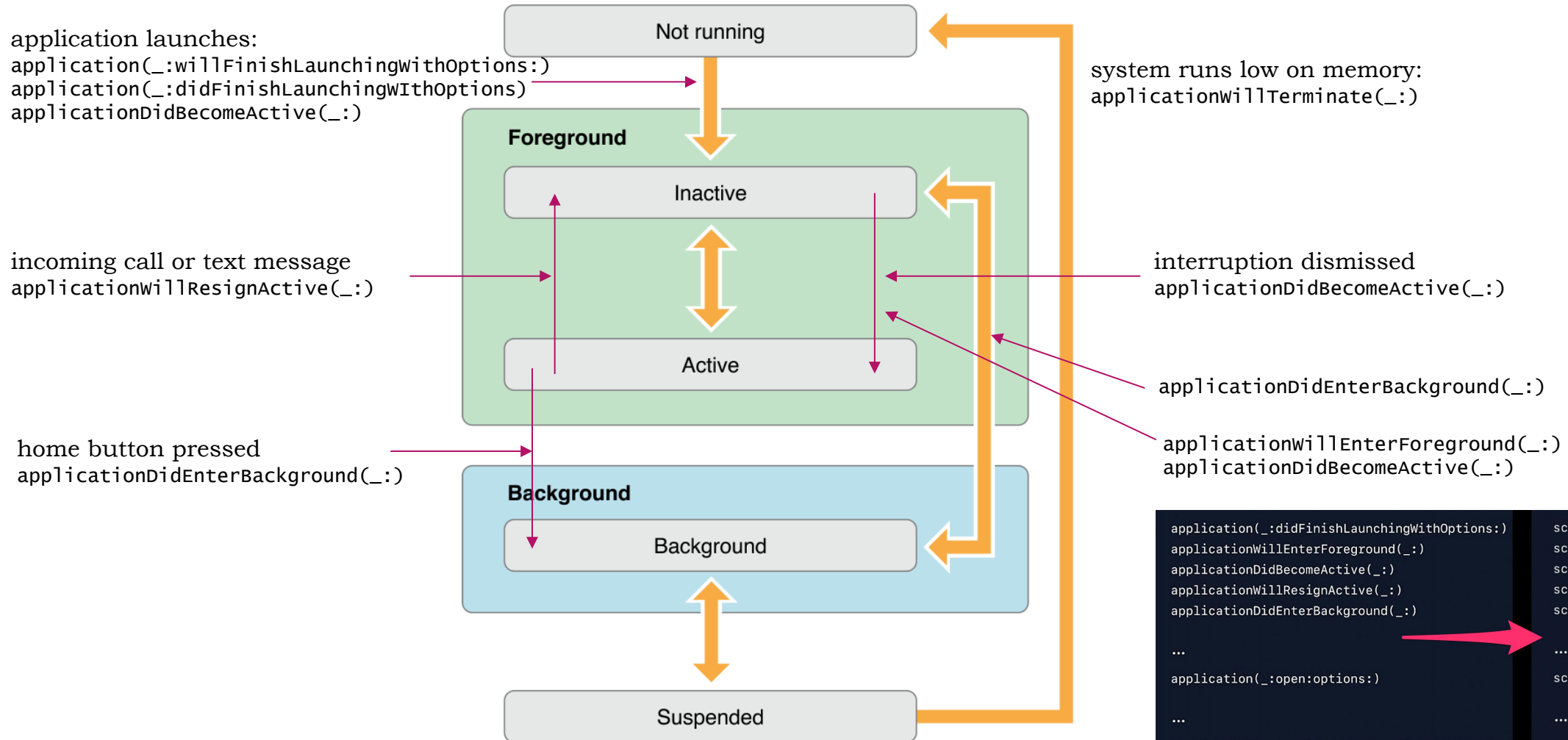
In iOS 13.0, there is now a "Scene Delegate", in `SceneDelegate.swift`

- ▶ `sceneDidEnterBackground(_:)`
- ▶ `sceneWillResignActive(_:)`

But note: stopping the app from Xcode prevents these methods from being called



# Application States and Transitions



# Application States

When an application is active, its interface is on the screen, its is accepting events, and its code is handling those events

An app in the active state can be interrupted by the OS:

- ▶ if a text message arrives, or a push notification, phone call, or alarm
- ▶ in this case, an overlay appears
- ▶ and the app moves to the inactive state

# Application States

When an app is in the inactive state

- ▶ the application is visible behind the overlay
- ▶ but it will not receive events
- ▶ the "lock" button will also force an active application into the inactive state

# Application States

When the user presses the home button, an application in the active state moves into the background state (it actually moves into the inactive state first)

In the background state, an app's interface is not visible, and it doesn't receive events

- ▶ but it can still execute code

After about 10 seconds, an app in the background state moves into the suspended state

# Application States

- ▶ In the suspended state, an app is not visible and cannot execute code
- ▶ An app in suspended state will remain there as long as the system has adequate memory to keep it there
- ▶ The OS can dump a suspended app when it wants to
- ▶ The task switcher shows apps that are in background or suspended state
- ▶ In the simulator, do shift-command-H twice to see the task switcher

# Application States

State	Visible	Receives Events	Executes Code
Not Running	no	no	no
Active	yes	yes	yes
Inactive	mostly	no	yes
Background	no	no	yes
Suspended	no	no	no

# Writing Data to the Filesystem

In Swift, the `Data` class is the representation of binary data

To save an image from the Homepwner app, turn the JPEG representation of the image into an instance of `Data`

► we can then store those instances of `Data` in the filesystem

# Writing Data to the Filesystem

1. Create a URI for an image

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

    let documentDirectory = documentsDirectories.first!
    return documentDirectory.appendingPathComponent(key)
}
```



# Writing Data to the Filesystem

## 2. Save the image

```
// in ImageStore
func setImage(_ image: UIImage, forKey key: String) {
    cache.setObject(image, forKey: key as NSString)
    // create a URI for the image
    let url = imageURL(forKey: key)
    // turn the image into jpeg data
    if let data = image.jpegData(compressionQuality: 0.5) {
        // write it to the uri
        let _ = try? data.write(to: url, options: [.atomic])
    }
}
```

- this is not archiving—this just writes raw data to the filesystem
- `.atomic` means "write the whole file to a temporary place and then rename it after the writing is complete"

note: the call in the book, `UIImageJPEGRepresentation()`, has been replaced by `jpegData(compressionQuality:)`

# Reading Data from the Filesystem

```
// in ImageStore
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
}
```

# guard

The `guard` statement in Swift:

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

`guard` is a kind of special `if` check

- ▶ if the statement after the `guard` fails, then the `else` block is executed
- ▶ furthermore, you must return in the else of a `guard` (otherwise, it's a compiler error)

# Deleting Data from the Filesystem

```
// in ImageStore
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)
    let url = imageURL(forKey: key)
    FileManager.default.removeItem(at: url)
}
```



Call can throw, but it is not marked with 'try' and the error is not handled

# Catching Thrown Errors

Some calls can throw errors

- ▶ such as `removeItem(at:)`

This is different from the concept of an optional: an optional either fails or it doesn't

To get more info about why something fails, Swift provides error handling

- ▶ In this case, we must wrap the call in a do-try-catch

# Deleting Data from the Filesystem

```
// in ImageStore
func deleteImage(forKey key: String) {
    cache.removeObject(forKey: key as NSString)
    let url = imageURL(forKey: key)
    FileManager.default.removeItem(at: url)
}
```

❗ Call can throw, but it is not marked with 'try' and the error is not handled

```
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: \(deleteError)")
    }
}
```

# A Few More Things

- ▶ p. 301 has a nice experiment to show activity transitions
  - `#function` is a compiler directive: the compiler fills it in with the name of the function in which it appears
- ▶ Exceptions vs. error handling in Swift
  - rule of thumb: exceptions are for programmer errors (such as indexing beyond the end of an array)
  - error handling is for things you can't control (such as an attempt to read a file that doesn't exist)
- ▶ Discussion of xml property lists
  - portable format for data storage
- ▶ The application bundle
  - what actually gets built into an app and put on a device

# The Application Bundle

When we build an app, Xcode creates an application bundle, which contains:

- ▶ the application's executable program
- ▶ resources: storyboard files, images, audio
- ▶ any other files that are part of resources (could also be a csv, for example, for the initial entries in a list or in a database)
- ▶ the files in the bundle are read-only



# Loading a File from an App's Bundle

First: get a reference to the application bundle (in your code):

```
let applicationBundle = Bundle.main
```

Then, for example, to load "dogs.png" from the resource of the app:

```
if let url = applicationBundle.url(forResource: "dogs", ofType: "png") {  
    // do something with url  
}
```