Data Persistence

Lecture 9

CIS 1951

Last time, in CIS 1951…

Networking in iOS

- HTTP requests and response handling (async/await)
- URLSession for network tasks
- Parsing JSON data, Encodable & Decodable
- Error handling and network best practices
- **Questions? Comments? Feedback?**

CIS 1951 as a whole

Lectures 1-6: The Basics

Lectures 7-10: Technologies

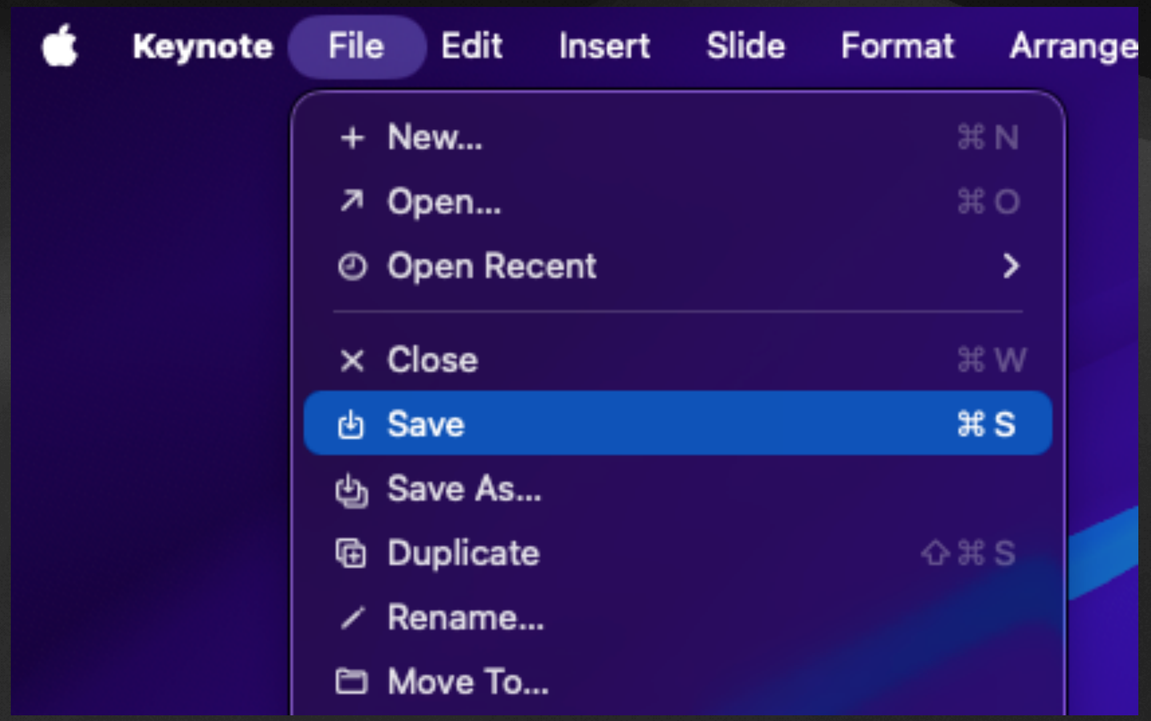
- Sensors
- Networking
- Data Persistence
- UIKit

Lectures 11-13: Beyond Development

What is Data Persistence?

Data Persistence

Save and retrieve data on disk
to access it across launches



Data Persistence

Why?

- Save user preferences and state
- Keep an offline cache of data
- Let users work on local documents
- etc

Data Persistence

TLDR: We want to be able to REMEMBER stuff

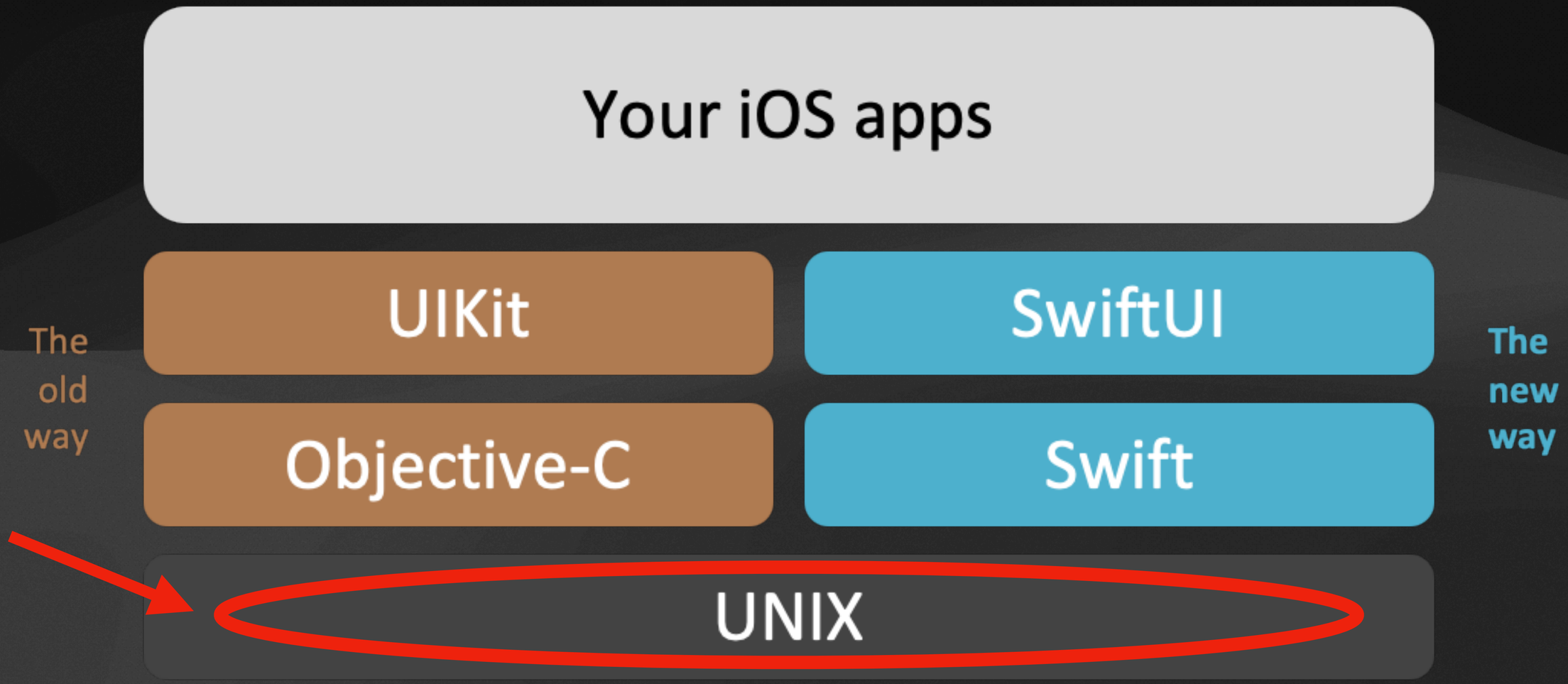
Data Persistence

Options

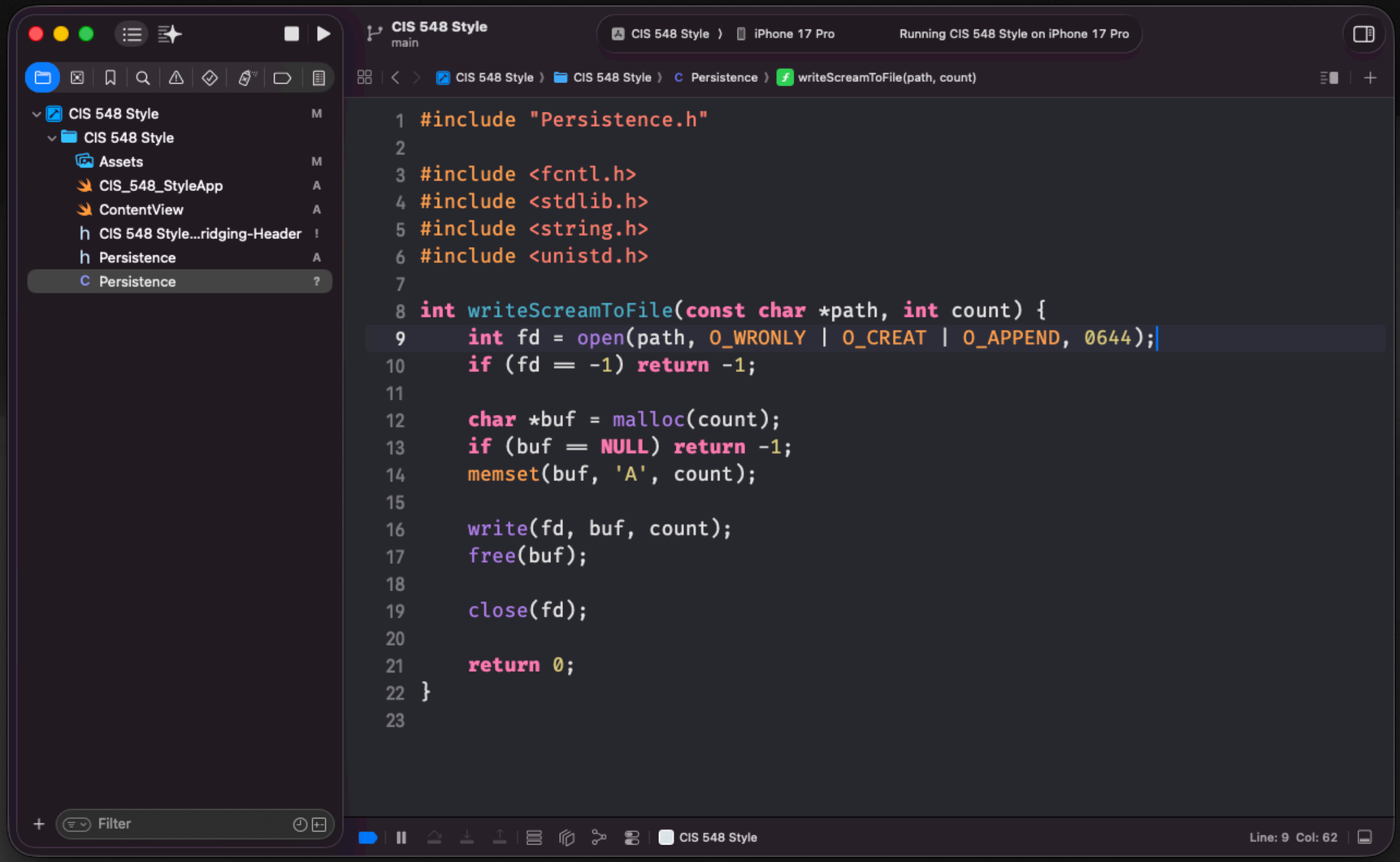
- File Management
- UserDefaults
- Core Data
- SwiftData
- Keychain

File Management

File Management



CIS 548 Style



Swift Style: Writing to a File

```swift
func saveTextToFile(text: String, fileName: String) {
    let baseURL = FileManager.default.url(for: .documentDirectory,
                                          in: .userDomainMask,
                                          appropriateFor: nil,
                                          create: true)
    let fileURL = baseURL.appendingPathComponent(fileName)

    do {
        try text.write(to: fileURL, atomically: true, encoding: .utf8)
    } catch {
        // Handle the error
        print("Error saving file: \(error)")
    }
}
```

Swift Style: Reading from a File

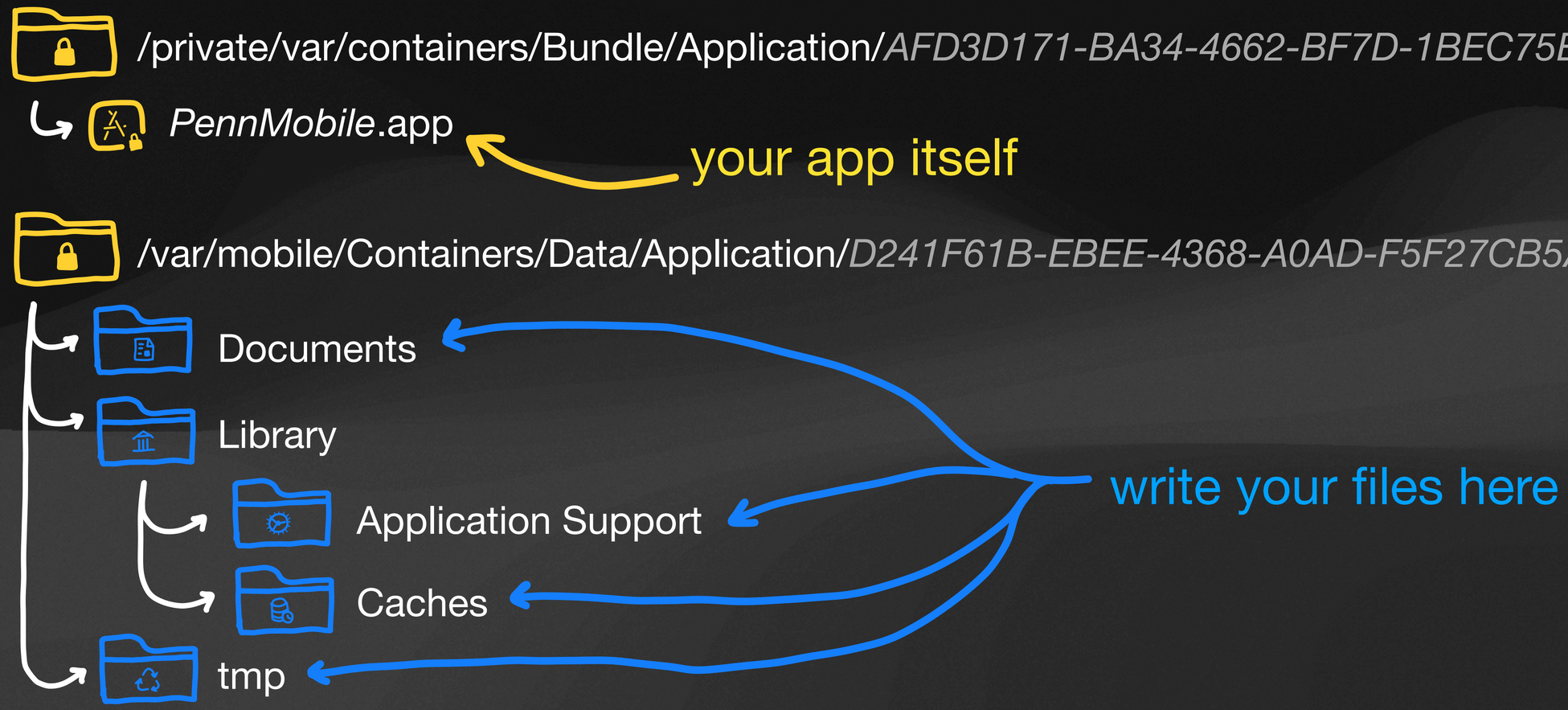
```swift
func readTextFromFile(fileName: String) -> String? {
    let baseURL = FileManager.default.url(for: .documentDirectory,
                                          in: .userDomainMask,
                                          appropriateFor: nil,
                                          create: true)
    let fileURL = baseURL.appendingPathComponent(fileName)

    do {
        let text = try String(contentsOf: fileURL, encoding: .utf8)
        return text
    } catch {
        // Handle the error
        print("Error reading file: \(error)")
        return nil
    }
}
```

Caveat: Sandboxing

- Your app can only read and write in its **own directories**
- Some exceptions apply: App Groups, photos/contacts/calendars with explicit user approval, etc
- iOS will return `EPERM` if you try to read/write a file outside your “sandbox”

Container Directories

🔒 = Read-only

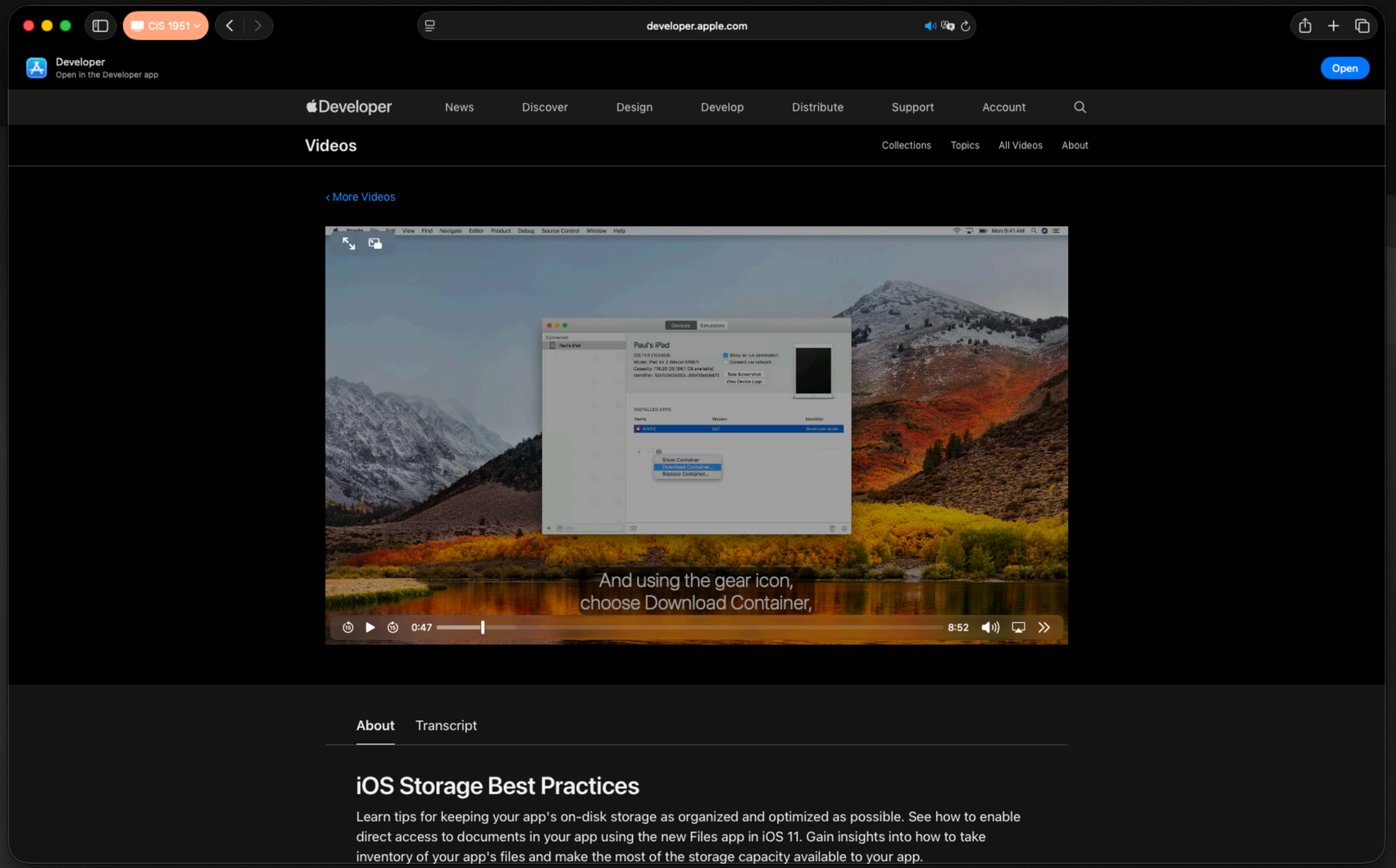


Don’t try to get these paths yourself —
use **FileManager.default.url(for:in:appropriateFor:create:)**

Downloading an app’s data container

iOS Storage Best Practices
2017 Tech Talks

0:30



File Management

Best Practices and Limitations

- Organize files into appropriate directories
- Handle errors and data integrity during read/write operations
- Manual management means higher complexity
- Use when you need maximal control, or when data doesn’t fit neatly into another API

UserDefaults

UserDefaults

Simple, lightweight storage

- Used to store lightweight user preferences and settings
  - Ideal for saving simple configurations (e.g. volume level, display mode)
  - Not intended for sensitive or large quantities of data
- 2 APIs: SwiftUI property wrappers, `UserDefaults` class

UserDefaults

Simple, lightweight storage

- How large is “too large”?
  - Ideally <1 MB
  - Designed for small pieces of data like booleans, integers, strings, or small arrays and dictionaries

1 Using UserDefaults

```
@AppStorage("key") var varName: Type = defaultValue
```

2 Saving Preferences with UserDefaults

```
struct FirstView: View {
    @AppStorage("username") var username: String = ""

    var body: some View {
        // Username is automatically saved
        TextField("Enter your username", text: $username)
            .padding()
    }
}
```

3 Retrieving Preferences with UserDefaults

```
struct SecondView: View {
    // Retrieve the username from UserDefaults
    // or use a default value
    @AppStorage("username") var username: String = "DefaultUser"

    var body: some View {
        Text("Welcome back, \(username)!")
            .padding()
    }
}
```

UserDefaults

Best Practices and Limitations

- Ensure default values are set for a better user experience
- Designed for simple data types and small datasets
- Use more secure storage methods for sensitive information
- May lead to clutter and misuse if overused for complex data

Core Data

Core Data

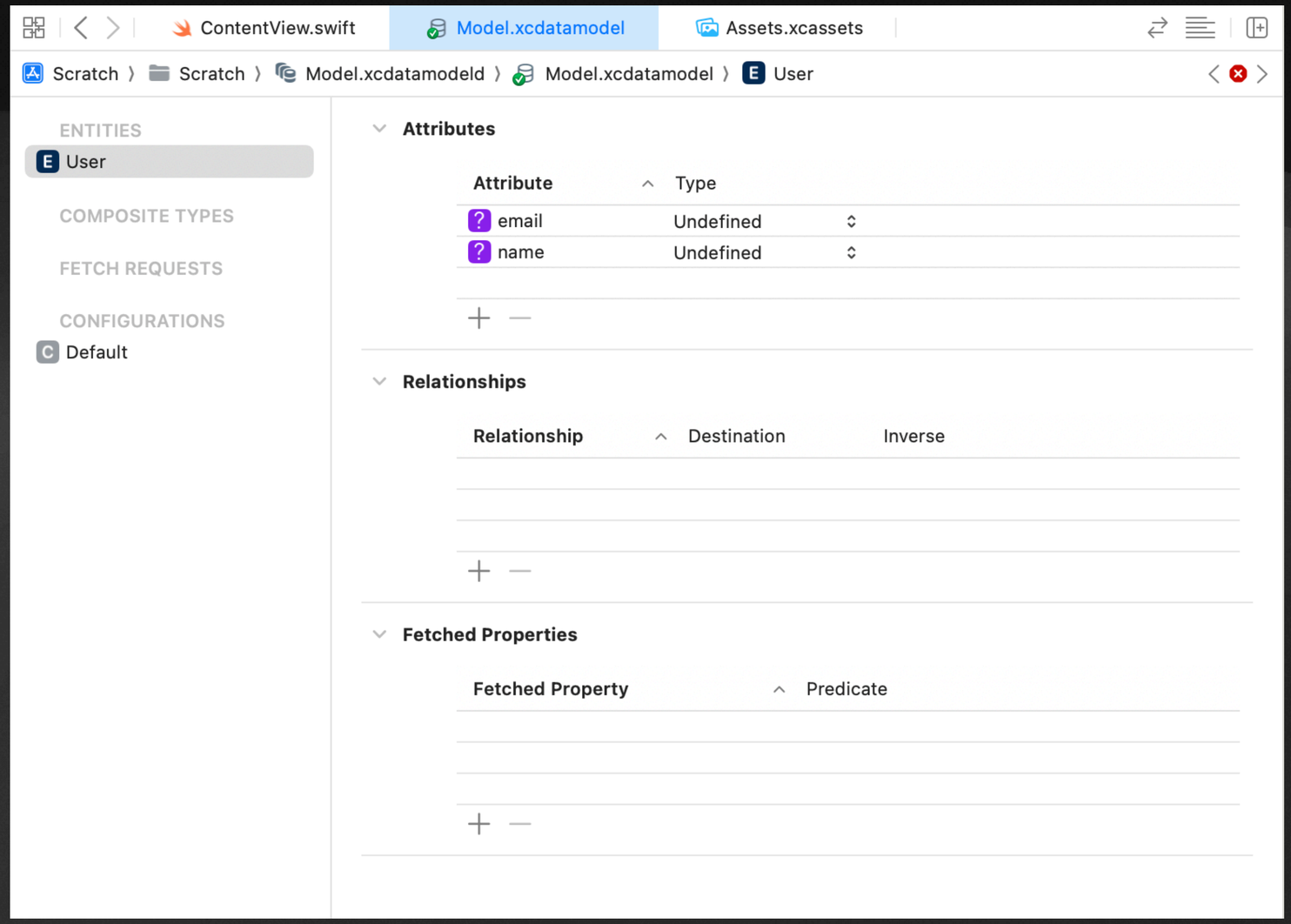
Complex, structured data

- Apple’s native framework for object graph and persistence
- Suitable for complex data models with relationships and extensive data.
- Used in apps requiring data persistence beyond simple preferences

Core Data: Set Up

Step 1: Create the Data Model

- **Xcode > New File > Data Model**
- Define your entities (i.e. objects) and attributes (i.e. properties)



Core Data: Set Up

Step 2: Initialize the Core Data Stack

```
import CoreData

struct PersistenceController {
    static let shared = PersistenceController()

    let container: NSPersistentContainer

    init() {
        container = NSPersistentContainer(name: "User")
        container.loadPersistentStores { (storeDescription, error) in
            if let error = error as NSError? {
                // Error handling...
            }
        }
    }
}
```

Core Data: Set Up

Step 3: Get the Managed Object Context

```
@main
struct MyApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext,
persistenceController.container.viewContext)
        }
    }
}
```

CRUD Operations

What are they?

- **Create:** Adding new records to your database
- **Read:** Fetching existing data
- **Update:** Modifying existing data
- **Delete:** Removing data

CRUD Operations with Core Data

CREATE

```
// Create
let newUser = User(context: managedObjectContext)
newUser.name = "John Doe"
```

CRUD Operations with Core Data

READ

```
// Read
// Traditional way - full control, manual management
let fetchRequest = NSFetchRequest<User>(entityName: "User")
let users = try? managedObjectContext.fetch(fetchRequest)
```

CRUD Operations with Core Data

READ

```
// Read
// SwiftUI way - Less control to fetch request details, but seamless integration
struct UserListView: View {
   @FetchRequest(
        entity: User.entity(),
        sortDescriptors: [NSSortDescriptor(keyPath: \User.name, ascending: true)]
  ) var users: FetchedResults<User>

    var body: some View {
        List(users, id: \.self) { user in
            Text(user.name ?? "Unknown")
        }
    }
}
```

CRUD Operations with Core Data

UPDATE

```
// Update
if let firstUser = users.first {
    firstUser.name = "Jane Doe"
}
```

CRUD Operations with Core Data

DELETE

```
// Delete
if let firstUser = users.first {
    managedObjectContext.delete(firstUser)
}
```

CRUD Operations with Core Data

SAVE - Write to DB!

```
// Save Changes
try? managedObjectContext.save()
```

SwiftData

SwiftData

Flexible for diverse data types

- Similar to Core Data
- Newer and easier to use, but less mature

1 Declaring a Model

```
@Model
class Recipe {
    @Attribute(.unique) var name: String
    var summary: String?
    var ingredients: [Ingredient]

    init(name: String, summary: String? = nil, ingredients: [Ingredient]) {
        self.name = name
        self.summary = summary
        self.ingredients = ingredients
    }
}
```

2 Set up the model container

```
@main
struct SwiftData_DemoApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
        .modelContainer(for: [Recipe.self])
    }
}
```

3 Write some data

```
struct AddRecipeView: View {
    @Environment(\.modelContext) var modelContext

    func addRecipe() {
        let recipe = Recipe(name: "Carrot Cake",
                            ingredients: [.init(emoji: "🥕", name: "Carrot")])
        modelContext.insert(recipe)
    }

    // ...
}
```

4 Query that data in SwiftUI

```
struct RecipeListView: View {
    @Query var recipes: [Recipe]

    var body: some View {
        List(recipes) { recipe in
            NavigationLink(recipe.name, value: recipe)
        }
    }
}
```

5 Update or delete data

```
// Update
recipe.ingredients.append(.init(emoji: "🍰", name: "Cake"))

// Delete
modelContext.delete(recipe)
```

SwiftData

Best Practices and Limitations

- Very new framework - watch for updates & potential bugs in edge cases
- Not as feature-rich or complex as Core Data for managing relationships between data

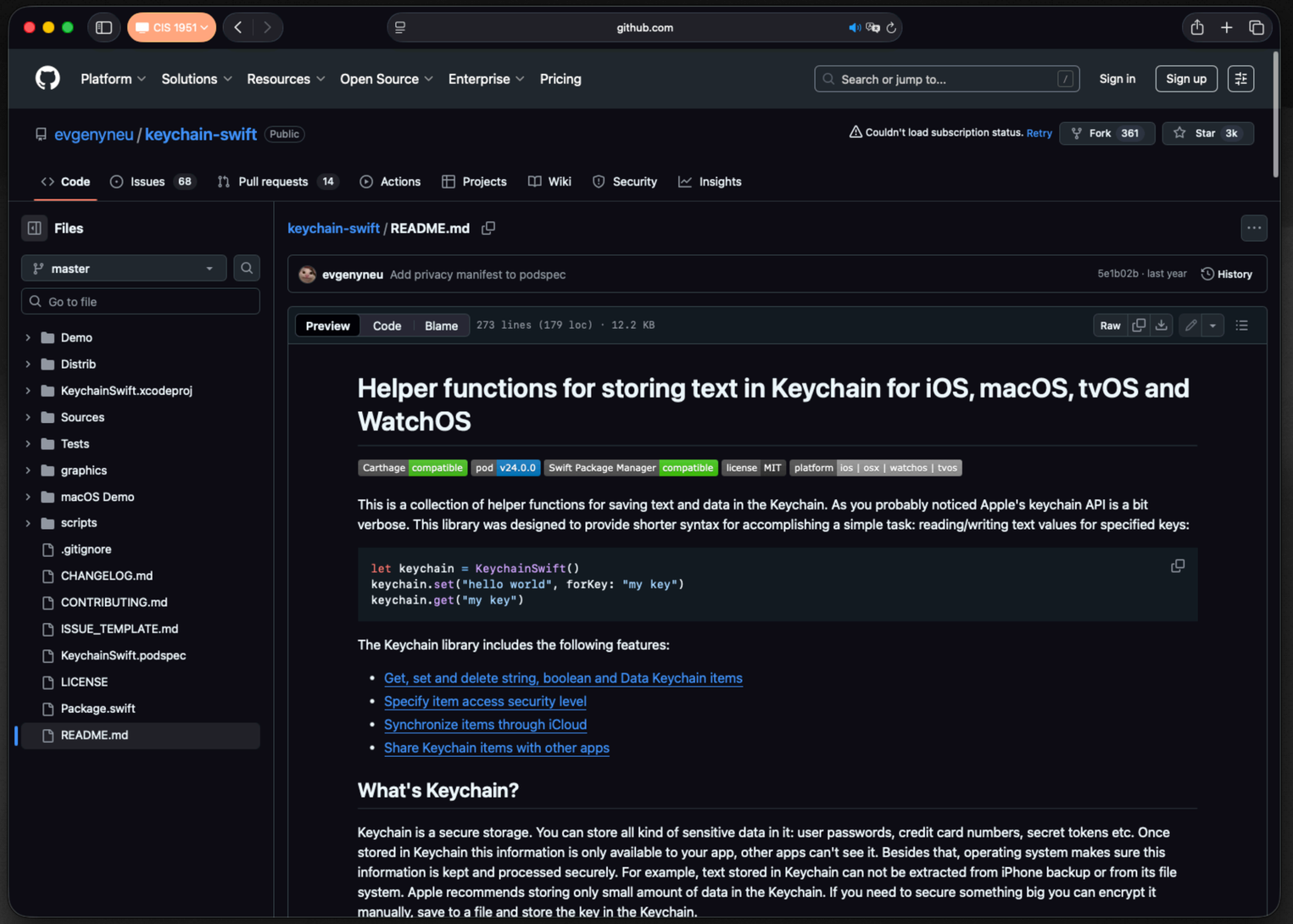
Keychain

Keychain

Secure and sensitive data

- Secure storage for…
  - Sensitive information (e.g. passwords, tokens, and encryption keys)
  - Personal data that must be kept secure
- Built-in API, but we’ll use a third-party wrapper

KeychainSwift



1 Saving to Keychain

```
import KeychainSwift

func saveToKeychain(key: String, value: String) {
    let keychain = KeychainSwift()
    keychain.set(value, forKey: key)
}
```

2 Reading from Keychain

```
import KeychainSwift

func readFromKeychain(key: String) -> String? {
    let keychain = KeychainSwift()
    return keychain.get(key)
}
```

Keychain

Best Practices and Limitations

- Use for small pieces of sensitive data, not large datasets
- Always check for the success or failure of Keychain operations
- Retrieval and storage processes can be slower due to encryption and decryption processes

Coding time!

https://github.com/cis1951/lec9-code