Mobile Programming

iOS Native Development

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

Swift | SwiftUI | UIKit | Platform Channels



Table of Contents

1. Why Learn iOS Native?
2. Swift Language Essentials
3. iOS App Architecture
4. SwiftUI — Declarative UI
5. UIKit — Imperative UI
6. View Controller Lifecycle
7. Background Modes & Tasks
8. iOS Permissions & Privacy
9. Platform Channels (Flutter w/ iOS)
10. Mini Lab: Native iOS Bridge
11. Summary & Key Takeaways

Why Learn iOS Native?

Understanding iOS native development is essential for Flutter engineers.

When you need native iOS:

- **Platform Channels:** Access iOS-only APIs (HealthKit, ARKit, PassKit).
- **Background Modes:** Audio, location, background fetch.
- **App Extensions:** Widgets, Share Extensions, Notification Service.
- **Code Signing & Provisioning:** Understand the deployment pipeline.
- **Debugging:** Read native crash logs, debug memory issues.
- **Security:** iOS sandbox, Keychain, Data Protection.

> **Key Insight:** iOS is more restrictive than Android. Understanding its constraints helps you design better cross-platform apps.

Swift Language Essentials

Swift is Apple's modern, safe, and fast programming language.

> https://google.github.io/swift/#line-wrapping

```swift
// Optionals (like Dart's nullable types)
var name: String = "Ali"        // Non-optional
var nickname: String? = nil     // Optional
let length = nickname?.count ?? 0  // Optional chaining + nil coalescing
struct User: Codable { // Struct (value type — preferred over class for models)
    let id: Int
    let name: String
    let email: String
}
// Enum with associated values
enum Result<T> {
    case success(T)
    case failure(String)
}
// Async/await (Swift 5.5+)
func fetchUsers() async throws -> [User] {
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode([User].self, from: data)
}
```

iOS App Architecture

An iOS app is built from several key components:

Core Components

- **AppDelegate:** App lifecycle events.
- **SceneDelegate:** Multi-window management (iOS 13+).
- **ViewController:** Manages a screen's UI and logic.
- **View:** Visual element (UIView or SwiftUI View).
- **Model:** Data structures and business logic.

App Entry Point

```
// SwiftUI app (modern)
@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

// UIKit app (traditional)
class AppDelegate: UIResponder,
    UIApplicationDelegate {
    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        return true
    }
}
```

SwiftUI — Declarative UI

Apple's modern UI framework. Very similar to Flutter!

```
struct UserCard: View {
    let user: User
    var onTap: () -> Void
    var body: some View {
        HStack(spacing: 12) {
            Image(systemName: "person.circle.fill")
                .font(.largeTitle)
                .foregroundColor(.blue)
            VStack(alignment: .leading) {
                Text(user.name)
                    .font(.headline)
                Text(user.email)
                    .font(.subheadline)
                    .foregroundColor(.secondary)
            }
            Spacer()
        }
        .padding()
        .background(Color(.systemBackground))\
        .shadow(radius: 4)
        .onTapGesture { onTap() }
```

SwiftUI — State Management

```swift
struct CounterView: View {
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Count: \(count)")
                .font(.title)
            HStack {
                Button("−") { count -= 1 }
                    .padding().background(Color.red).foregroundColor(.white)
                    .cornerRadius(8)
                Button("+") { count += 1 }
                    .padding().background(Color.green).foregroundColor(.white)
                    .cornerRadius(8)
            }
        }
    }
}

// Observable pattern (like ChangeNotifier)
class UserStore: ObservableObject {
    @Published var users: [User] = []
    func load() async {
        users = try! await fetchUsers()
    }
}
```

UIKit — Imperative UI

The traditional iOS UI framework. Still widely used and important to understand.

```
class UserViewController: UIViewController {
    private let tableView = UITableView()
    private var users: [User] = []
    override func viewDidLoad() {
        super.viewDidLoad()
        title = "Users"
        view.addSubview(tableView)
        tableView.frame = view.bounds
        tableView.dataSource = self
        tableView.delegate = self
        loadUsers()
    }
    private func loadUsers() {
        Task {
            users = try await fetchUsers()
            tableView.reloadData()
        }
    }
}
```

UIKit — Imperative UI (cont.)

```
extension UserViewController: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int {
        return users.count
    }
    func tableView(_ tableView: UITableView,
                   cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell",
                                                 for: indexPath)
        cell.textLabel?.text = users[indexPath.row].name
        return cell
    }
}
```

View Controller Lifecycle

Understanding the lifecycle is critical for proper resource management.

Key Lifecycle Methods:

1. `loadView()` — Create the view hierarchy programmatically.
2. `viewDidLoad()` — View loaded into memory. Setup here.
3. `viewWillAppear()` — About to appear on screen.
4. `viewDidAppear()` — Fully visible. Start animations.
5. `viewWillDisappear()` — About to leave screen.
6. `viewDidDisappear()` — No longer visible. Stop timers.
7. `didReceiveMemoryWarning()` — Free memory or be terminated.

> **Golden Rule:** Always pair setup in `viewWillAppear` with cleanup in `viewWillDisappear`.

Background Modes & Tasks

iOS is **very strict** about background execution.

Allowed Background Modes

- Audio playback
- Location updates
- VoIP (internet telephony)
- Background fetch
- Remote notifications
- Background processing (iOS 14+)

BGTaskScheduler (iOS 14+)

```
// Schedule a background task
let request = BGProcessingTaskRequest(
    identifier: "com.example.sync"
)
request.requiresNetworkConnectivity = true
request.requiresExternalPower = false
request.earliestBeginDate = Date(
    timeIntervalSinceNow: 3600
)

try? BGTaskScheduler.shared
    .submit(request)
```

> **Key Difference:** Unlike Android's WorkManager, iOS decides *when* to run your task — you only specify constraints.

iOS Permissions & Privacy

iOS requires explicit user permission for sensitive data.

Privacy Descriptions (Info.plist):

- `NSCameraUsageDescription` — Camera access.
- `NSLocationWhenInUseUsageDescription` — Location (foreground).
- `NSPhotoLibraryUsageDescription` — Photo library.
- `NSContactsUsageDescription` — Contacts.
- `NSMicrophoneUsageDescription` — Microphone.

iOS Permissions & Privacy (cont.)

```swift
// Request permission
import AVFoundation

func requestCameraAccess() {
    AVCaptureDevice.requestAccess(for: .video) { granted in
        DispatchQueue.main.async {
            if granted {
                // Open camera
            } else {
                // Show denied message
            }
        }
    }
}
```

> **Important:** If you forget the Info.plist entry, the app **crashes** silently.

Platform Channels (Flutter w/ iOS)

Communicate between Flutter and native iOS code.

```
// Flutter side
const channel = MethodChannel('com.example/device');

Future<String> getDeviceModel() async {
  try {
    return await channel.invokeMethod<String>('getDeviceModel') ?? 'Unknown';
  } on PlatformException catch (e) {
    return 'Error: ${e.message}';
  }
}
```

Platform Channels (Flutter w/ iOS - cont.)

```swift
// iOS side (AppDelegate.swift)
class AppDelegate: FlutterAppDelegate {
    override func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        let controller = window?.rootViewController as! FlutterViewController
        let channel = FlutterMethodChannel(
            name: "com.example/device",
            binaryMessenger: controller.binaryMessenger
        )
        channel.setMethodCallHandler { (call, result) in
            if call.method == "getDeviceModel" {
                let model = UIDevice.current.model
                result(model)
            } else {
                result(FlutterMethodNotImplemented)
            }
        }
        return super.application(application, didFinishLaunchingWithOptions: launchOptions)
    }
}
```

Mini Lab

Native iOS Bridge — Device Info

Objective: Create a Flutter app that calls native iOS APIs via Platform Channels.

Requirements:

1. Create a `MethodChannel` named `com.example.device`.
2. Implement `getDeviceModel` — returns iPhone model name.
3. Implement `getOSVersion` — returns iOS version string.
4. Implement `getBatteryLevel` — returns battery percentage.
5. Implement `showAlert` — shows a native iOS `UIAlertController`.
6. Display all results in a Flutter UI.

Summary: Key Takeaways

iOS Native

- Swift is the modern language.
- SwiftUI ≈ Flutter's declarative model.
- UIKit is the traditional framework.
- Very strict background execution rules.

Permissions

- Must declare in Info.plist.
- App crashes if description is missing.
- User can revoke at any time.

Platform Channels

- Same concept as Android.
- `FlutterMethodChannel` for method calls.
- `FlutterEventChannel` for streaming.
- Swift handler in AppDelegate.

Key Differences from Android

- No true background services.
- Code signing is mandatory.
- Sandbox is stricter.
- App Store review process.

Thank You! 🙌

Questions?

"A Swift developer walks into a bar.
The bartender says, 'What can I get you?'
The developer replies, 'I'll have an Optional beer.'
The bartender: 'Sorry, that's nil right now.'"