Practical Exercise – Swift Fundamentals and iOS Development

Introduction to Native iOS Development

Course: Mobile Application Development (Chapter 6)
Level: Technical University Students
Duration: 6–8 hours self-study + 1–2 hours online discussion
Format: Individual Assignment with Code Examples (3–4 pages)

Exercise Overview

In this exercise, you will master core Swift concepts and iOS development fundamentals: Swift syntax and data types, class structures and OOP principles, UIViewController lifecycle, Storyboards with Auto Layout constraints, AppDelegate architecture, and native module integration. These foundations form the basis of all native iOS applications.

Key Learning Outcomes

- Understand Swift language fundamentals: syntax, data types, optionals, and error handling
- Master object-oriented programming in Swift: classes, structures, protocols, and inheritance
- Design responsive layouts using Storyboards and Auto Layout constraints
- Navigate the complete UIViewController lifecycle and state management
- Comprehend AppDelegate and SceneDelegate roles in iOS app lifecycle
- Integrate native iOS frameworks and modules into Swift applications

---

Part A: Swift Language Fundamentals

Task 1: Swift Syntax and Data Types Mastery (Self-Study: 1.5–2 hours)

Objective: Master Swift syntax, data types, and optionals—core language features for iOS development.

Instructions:

1. **Swift Basic Syntax and Variables:**

Define and explain (1–2 sentences each):

- `var` vs. `let` — mutable vs. immutable variables
- Type inference — Swift's automatic type detection
- Type annotations — explicit type declarations

2. **Write code examples for five Swift data types:**

```
// String type
let message: String = "Hello, iOS"
let interpolation = "User: (message)"

// Integer and Floating-Point
let count: Int = 42
let temperature: Double = 98.6
let precision: Float = 3.14

// Boolean
let isLoggedIn: Bool = true
let isOffline = false

// Arrays
let numbers: [Int] = [1, 2, 3, 4, 5]
let mixed = ["iOS", 15, true] // Type: [Any]

// Dictionaries
let user: [String: Any] = [
"name": "Alice",
"age": 28,
"isStudent": false
]

// Tuples
let coordinate: (Int, Int) = (10, 20)
let namedTuple = (x: 10, y: 20)
let (x, y) = coordinate
```

3. **Optionals and Error Handling:**

Explain in 2–3 sentences each:

- What are optionals and why they're necessary in Swift
- Unwrapping optionals: forced unwrap (`!`), optional binding (`if let`), nil-coalescing (`??`)
- Error handling: `try-catch` blocks and `throws`

4. **Write code examples:**

```
// Optional types
var optionalName: String?
optionalName = "Bob"
```

```
// Forced unwrap (unsafe)
let name = optionalName! // Crash if nil

// Optional binding (safe)
if let name = optionalName {
print("Name is (name)")
} else {
print("Name is nil")
}

// Guard statement (early exit)
guard let name = optionalName else {
print("Name is missing")
return
}
print("Name is (name)")

// Nil-coalescing operator
let displayName = optionalName ?? "Unknown"

// Error handling
enum LoginError: Error {
case invalidEmail
case incorrectPassword
}

func login(email: String, password: String) throws -> String {
guard email.contains("@") else {
throw LoginError.invalidEmail
}
guard password.count >= 6 else {
throw LoginError.incorrectPassword
}
return "Login successful"
}

// Using try-catch
do {
let result = try login(email: "user@example.com", password: "secret123")
print(result)
} catch LoginError.invalidEmail {
print("Invalid email format")
} catch LoginError.incorrectPassword {
print("Password too short")
} catch {
print("Unknown error: (error)")
}
```

5. **Collections and Iteration:**

Write code examples for:

- Array operations: append, remove, filter, map
- Dictionary operations: accessing, adding, updating values

- Set operations: unique values, intersection, union

```
// Array operations
var fruits = ["Apple", "Banana", "Orange"]
fruits.append("Grape")
fruits.remove(at: 1) // Remove "Banana"

let filtered = fruits.filter { $0.count > 5 } // Longer than 5 chars
let mapped = fruits.map { $0.uppercased() } // Convert to uppercase

// Dictionary operations
var scores: [String: Int] = ["Alice": 90, "Bob": 85]
scores["Charlie"] = 92 // Add new entry
scores["Alice"] = 95 // Update existing
if let aliceScore = scores["Alice"] {
print("Alice's score: (aliceScore)")
}

// Set operations
let setA: Set = [1, 2, 3, 4]
let setB: Set = [3, 4, 5, 6]
let intersection = setA.intersection(setB) // [3, 4]
let union = setA.union(setB) // [1, 2, 3, 4, 5, 6]
```

Deliverable:

- Five data type code examples with explanations
- Optional and error handling code samples
- 3–4 sentence reflection on Swift's type system strengths

Part B: Object-Oriented Programming in Swift

Task 2: Classes, Structures, and Protocols (Self-Study: 1.5–2 hours)

Objective: Master OOP concepts in Swift: classes, structs, protocols, inheritance, and polymorphism.

Instructions:

1. **Define the three OOP pillars in Swift:**

Concept	Swift Implementation	Key Use Case
Encapsulation	Access modifiers (private, internal, public)	Hide implementation details, expose interfaces

Inheritance	Class inheritance with `class Parent: Child`	Share common functionality across types
Polymorphism	Method overriding and protocols	Multiple implementations of same interface

2. **Write code comparing Classes vs. Structures:**

// CLASSES: Reference types, support inheritance
class User {
var name: String
var email: String

```
init(name: String, email: String) {
    self.name = name
    self.email = email
}

func displayInfo() -> String {
    return "\(name) - \(email)"
}
```

}

// STRUCTURES: Value types, no inheritance
struct Point {
var x: Int
var y: Int

```
mutating func move(dx: Int, dy: Int) {
    x += dx
    y += dy
}
```

}

// Using classes (reference semantics)
let user1 = User(name: "Alice", email: "alice@example.com")
var user2 = user1
user2.name = "Alice Changed"
print(user1.name) // Output: "Alice Changed" (same reference)

// Using structs (value semantics)
let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 10
print(point1.x) // Output: 0 (separate copy)

3. **Inheritance Example:**

// Parent class
class Animal {
var name: String

```
init(name: String) {
    self.name = name
}

func speak() -> String {
    return "Generic sound"
}
```

}

// Child class
class Dog: Animal {
override func speak() -> String {
return "(name) says: Woof!"
}
}

let dog = Dog(name: "Rex")
print(dog.speak()) // Output: "Rex says: Woof!"

4. **Protocols and Polymorphism:**

// Protocol definition
protocol Drawable {
func draw()
var color: String { get set }
}

// Conforming class
class Circle: Drawable {
var color: String
var radius: Int

```
init(color: String, radius: Int) {
    self.color = color
    self.radius = radius
}

func draw() {
    print("Drawing circle with color \(color)")
}
```

}

// Conforming struct
struct Rectangle: Drawable {
var color: String
var width: Int
var height: Int

```
mutating func draw() {
    print("Drawing rectangle with color \(color)")
}
```

}

// Using polymorphism
let shapes: [Drawable] = [
Circle(color: "Red", radius: 5),
Rectangle(color: "Blue", width: 10, height: 20)
]

for shape in shapes {
shape.draw()
}

5. **Properties and Methods:**

class BankAccount {
private var balance: Double = 0 // Encapsulated data

```
// Computed property
var displayBalance: String {
    return String(format: "$%.2f", balance)
}

// Property observer
var transactions: Int = 0 {
    willSet {
        print("About to record transaction \(newValue)")
    }
    didSet {
        print("Transaction recorded. Total: \(transactions)")
    }
}

// Instance method
func deposit(_ amount: Double) {
    balance += amount
    transactions += 1
}

// Class method
static func interestRate() -> Double {
    return 0.05
}
```

}

let account = BankAccount()
account.deposit(100)
print(account.displayBalance) // Output: "$100.00"

Deliverable:

- Comparison table: Classes vs. Structures with 2–3 examples each
- Inheritance code example with parent and child classes
- Protocol conformance example with 2+ types
- 2–3 sentence explanation: when to use classes vs. structs

Part C: iOS UIViewController Lifecycle and Storyboards

Task 3: UIViewController Lifecycle and Auto Layout (Self-Study: 2–2.5 hours)

Objective: Master the UIViewController lifecycle, Storyboards, and constraint-based layout for responsive iOS apps.

Instructions:

1. **UIViewController Lifecycle Diagram:**

Draw or describe the complete lifecycle:

init(coder:) / init(frame:)
↓
loadView()
↓
viewDidLoad() ← Set up UI and initial state
↓
viewWillAppear(

:) ← Prepare for appearance↓viewDidAppear(:) ← View is visible
↓
--- User Interaction ---
↓
viewWillDisappear(

:) ← About to disappear↓viewDidDisappear(:) ← View is hidden
↓
deinit ← Memory cleanup

2. **Implement UIViewController with Lifecycle Callbacks:**

import UIKit

class MainViewController: UIViewController {

```
// MARK: - Outlets
@IBOutlet weak var titleLabel: UILabel!
@IBOutlet weak var contentView: UIView!
```

```swift
// MARK: - Lifecycle Methods

override func viewDidLoad() {
    super.viewDidLoad()
    // Called after view hierarchy is loaded
    // Setup: configure UI, add observers, fetch initial data
    titleLabel.text = "Welcome"
    setupAppearance()
    fetchInitialData()
}

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    // Called before view becomes visible
    // Refresh data, resume animations, subscribe to notifications
    print("View will appear")
    startLocationUpdates()
}

override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    // Called after view is fully visible
    // Play animations, start timers
    animateContentView()
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    // Called before view disappears
    // Save state, pause animations, unsubscribe
    stopLocationUpdates()
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    // Called after view is hidden
    print("View did disappear")
}

// MARK: - Setup Methods

private func setupAppearance() {
    view.backgroundColor = .white
    navigationItem.title = "Main"
}

private func fetchInitialData() {
    // Simulate API call
    DispatchQueue.main.asyncAfter(deadline: .now() + 1.0) {
        self.titleLabel.text = "Data Loaded"
    }
}

private func startLocationUpdates() {
    // Resume location tracking
}

private func stopLocationUpdates() {
    // Pause location tracking
}

private func animateContentView() {
    UIView.animate(withDuration: 0.3) {
        self.contentView.alpha = 1.0
    }
}

deinit {
    print("MainViewController deallocated")
}
```

}

3. **Storyboard-based UI Design (XML pseudocode):**

Describe a Login Screen using Storyboard elements:

4. **Auto Layout Constraints Code Example:**

// Programmatic constraint setup (without Storyboard)
import UIKit

class ConstraintViewController: UIViewController {

```
let emailTextField = UITextField()
let passwordTextField = UITextField()
let loginButton = UIButton()

override func viewDidLoad() {
    super.viewDidLoad()

    setupUI()
    setupConstraints()
}

private func setupUI() {
    // Configure TextFields
    emailTextField.placeholder = "Email"
    emailTextField.borderStyle = .roundedRect
    view.addSubview(emailTextField)

    passwordTextField.placeholder = "Password"
    passwordTextField.isSecureTextEntry = true
    passwordTextField.borderStyle = .roundedRect
    view.addSubview(passwordTextField)

    // Configure Button
    loginButton.setTitle("Login", for: .normal)
    loginButton.backgroundColor = .systemBlue
    loginButton.setTitleColor(.white, for: .normal)
    view.addSubview(loginButton)
}

private func setupConstraints() {
    emailTextField.translatesAutoresizingMaskIntoConstraints = false
    passwordTextField.translatesAutoresizingMaskIntoConstraints = false
    loginButton.translatesAutoresizingMaskIntoConstraints = false

    NSLayoutConstraint.activate([
        // Email TextField
        emailTextField.topAnchor.constraint(equalTo:
view.safeAreaLayoutGuide.topAnchor, constant: 30),
        emailTextField.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 16),
        emailTextField.trailingAnchor.constraint(equalTo:
view.trailingAnchor, constant: -16),
        emailTextField.heightAnchor.constraint(equalToConstant: 44),

        // Password TextField
        passwordTextField.topAnchor.constraint(equalTo:
emailTextField.bottomAnchor, constant: 16),
        passwordTextField.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 16),
        passwordTextField.trailingAnchor.constraint(equalTo:
view.trailingAnchor, constant: -16),
        passwordTextField.heightAnchor.constraint(equalToConstant: 44),

        // Login Button
        loginButton.topAnchor.constraint(equalTo:
passwordTextField.bottomAnchor, constant: 20),
        loginButton.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 16),
        loginButton.trailingAnchor.constraint(equalTo:
view.trailingAnchor, constant: -16),
        loginButton.heightAnchor.constraint(equalToConstant: 48)
    ])
}
```

}

5. **Layout Design Questions (2–3 sentences each):**

- **Why use Auto Layout over manual frame calculations?** Auto Layout adapts layouts automatically to different screen sizes, orientations, and device types. It eliminates manual positioning and makes apps responsive across iPhones and iPads.
- **How do Safe Area constraints improve layout compatibility?** Safe Area avoids system UI elements like notches, home indicators, and status bars, ensuring content displays correctly on all iOS devices.
- **What's one accessibility improvement you'd make to the login screen?** Add `accessibilityLabel` to buttons, ensure text contrast meets WCAG standards (4.5:1 ratio), and support VoiceOver with proper reading order.

Deliverable:

- UIViewController lifecycle diagram with all callback methods
- UIViewController code implementation with lifecycle callbacks
- Storyboard pseudo-code or description of login screen layout
- Programmatic constraint setup example
- 2–3 sentence answers to layout design questions

Part D: iOS App Lifecycle and Native Integration

Task 4: AppDelegate, SceneDelegate, and Native Module Integration (Self-Study: 1–1.5 hours)

Objective: Understand the complete iOS app lifecycle from launch through termination, and integrate native frameworks.

Instructions:

1. **iOS App Lifecycle Overview:**

// AppDelegate: Entry point and app-level events
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

```
// MARK: - App Lifecycle Methods

func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?
) -> Bool {
    // Called when app first launches
    // Initialize third-party SDKs, configure app state
    print("App launched")
    initializeThirdPartySolutions()
    return true
}

func applicationWillEnterForeground(_ application: UIApplication) {
    // Called when app is about to become active
    print("App entering foreground")
}

func applicationDidBecomeActive(_ application: UIApplication) {
    // Called when app is fully in foreground
    print("App is active")
}

func applicationWillResignActive(_ application: UIApplication) {
    // Called when app is about to lose focus
    print("App losing focus")
}

func applicationDidEnterBackground(_ application: UIApplication) {
    // Called when app enters background
    // Save data, pause heavy operations
    print("App in background")
    saveUserData()
}

func applicationWillTerminate(_ application: UIApplication) {
    // Called when app is about to terminate
    // Final cleanup
    print("App terminating")
}

// MARK: - Helper Methods

private func initializeThirdPartySolutions() {
    // Firebase, Crashlytics, Analytics, etc.
}

private func saveUserData() {
    // Save to UserDefaults, Core Data, etc.
}
```

}

// MARK: - SceneDelegate (iOS 13+)

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

```
var window: UIWindow?

func scene(
    _ scene: UIScene,
    willConnectTo session: UISceneSession,
    options connectionOptions: UIScene.ConnectionOptions
) {
    // Called when a new scene session is created
    guard let windowScene = (scene as? UIWindowScene) else { return }

    let window = UIWindow(windowScene: windowScene)
    let mainVC = ViewController()
    window.rootViewController = mainVC
    self.window = window
    window.makeKeyAndVisible()
}

func sceneDidBecomeActive(_ scene: UIScene) {
    // Scene is in foreground and interactive
}

func sceneWillResignActive(_ scene: UIScene) {
    // Scene will enter background
}

func sceneDidEnterBackground(_ scene: UIScene) {
    // Scene is in background
}
```

}

2. **Native Module Integration Examples:**

import UIKit
import AVFoundation // Camera and audio
import CoreLocation // GPS and location
import HealthKit // Health and fitness data
import Photos // Photo and video library
import MessageUI // In-app messaging

class NativeIntegrationViewController: UIViewController {

```
// MARK: - Camera Integration

func openCamera() {
    guard UIImagePickerController.isSourceTypeAvailable(.camera) else {
        print("Camera not available")
        return
    }

    let picker = UIImagePickerController()
    picker.sourceType = .camera
    picker.delegate = self
    present(picker, animated: true)
}

// MARK: - Location Services

private let locationManager = CLLocationManager()

func requestLocationPermission() {
    locationManager.delegate = self
    locationManager.requestWhenInUseAuthorization()
    locationManager.startUpdatingLocation()
}

// MARK: - HealthKit Integration

func requestHealthKitAuthorization() {
    guard HKHealthStore.isHealthDataAvailable() else {
        print("HealthKit not available")
        return
    }

    let healthStore = HKHealthStore()
    let stepType = HKQuantityType.quantityType(
        forIdentifier: .stepCount
    )!

    healthStore.requestAuthorization(toShare: [], read: [stepType]) {
success, error in
        if success {
            self.fetchStepCount(healthStore: healthStore)
        }
    }
}

private func fetchStepCount(healthStore: HKHealthStore) {
    let stepType = HKQuantityType.quantityType(forIdentifier:
.stepCount)!
    let query = HKStatisticsQuery(
        quantityType: stepType,
        quantitySamplePredicate: nil,
        options: .cumulativeSum
    ) { _, result, _ in
        guard let result = result, let sum = result.sumQuantity() else
{ return }
        let steps = Int(sum.doubleValue(for: HKUnit.count()))
        print("Steps today: \(steps)")
    }
    healthStore.execute(query)
}

// MARK: - Photo Library

func openPhotoLibrary() {
    guard UIImagePickerController.isSourceTypeAvailable(.photoLibrary)
else {
        return
    }

    let picker = UIImagePickerController()
    picker.sourceType = .photoLibrary
    picker.delegate = self
    present(picker, animated: true)
}

// MARK: - Message Sending

func sendMessage(recipient: String, message: String) {
    guard MFMessageComposeViewController.canSendText() else {
        print("SMS not available")
        return
    }

    let composer = MFMessageComposeViewController()
    composer.messageComposeDelegate = self
    composer.recipients = [recipient]
    composer.body = message
    present(composer, animated: true)
}
```

}

// MARK: - Delegates

extension NativeIntegrationViewController: UIImagePickerControllerDelegate, UINavigationControllerDelegate {

```
func imagePickerController(
    _ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info:
[UIImagePickerController.InfoKey: Any]
) {
    if let image = info[.originalImage] as? UIImage {
        print("Image selected: \(image)")
    }
    picker.dismiss(animated: true)
}
```

}

extension NativeIntegrationViewController: CLLocationManagerDelegate {

```
func locationManager(
    _ manager: CLLocationManager,
    didUpdateLocations locations: [CLLocation]
) {
    guard let location = locations.last else { return }
    print("Current location: \(location.coordinate)")
}
```

}

extension NativeIntegrationViewController: MFMessageComposeViewControllerDelegate {

```
func messageComposeViewController(
    _ controller: MFMessageComposeViewController,
    didFinishWith result: MessageComposeResult
) {
    controller.dismiss(animated: true)
}
```

}

3. **Answer Reflection Questions (200–250 words total):**

1. App Lifecycle Importance:

Why does iOS separate app lifecycle into distinct states (Not Running, Inactive, Active, Background, Suspended) rather than a simple "on/off" model?

Answer: The multi-state lifecycle allows iOS to manage limited resources (memory, battery, CPU) efficiently. The Background state lets apps perform critical tasks (downloads, location updates) without being fully active. The Suspended state preserves app state while freeing memory. This design balances user experience with system resource constraints, ensuring responsive apps while preventing battery drain from background processes.

2. Native Module Integration Philosophy:

Why does Apple encourage integrating native frameworks (AVFoundation, CoreLocation, HealthKit) instead of building custom implementations?

Answer: Native frameworks are optimized for iOS hardware, tested thoroughly for security and performance, and updated with new iOS versions. They handle permissions, privacy settings, and system integrations seamlessly. Using native modules reduces development complexity, ensures consistency with iOS conventions, and provides automatic support for new devices and OS features. This approach improves reliability and security.

3. AppDelegate vs. SceneDelegate:

Explain the difference and when each is used. Why did Apple introduce SceneDelegate with iOS 13?

Answer: AppDelegate handles app-level events (launch, termination, memory warnings). SceneDelegate (iOS 13+) manages individual window scenes, supporting multi-window apps on iPad and split-screen multitasking. Apple introduced SceneDelegate to support iPadOS multitasking and prepare for future devices with multiple screens. Modern apps prioritize SceneDelegate for state management while AppDelegate handles app-wide concerns.

Deliverable:

- Complete AppDelegate implementation with lifecycle methods
- SceneDelegate setup example
- At least 3 native module integration examples (Camera, Location, HealthKit)
- Answers to all 3 reflection questions (200–250 words total)

Evaluation Criteria

Criteria	Excellent (9–10)	Good (7–8)	Acceptable (5–6)	Needs Improvement (<5)
Swift Fundamentals	Complete coverage of syntax, types, optionals, error handling	Most concepts covered, minor omissions	Basic syntax shown, optionals unclear	Incomplete or incorrect fundamentals
OOP Implementation	All OOP pillars (encapsulation, inheritance, polymorphism) demonstrated	Classes/structs shown, inheritance present	Basic class structure, no protocols	Missing OOP concepts
UIViewController Lifecycle	Complete lifecycle mapped with proper callbacks and state management	Lifecycle mostly correct, callbacks accurate	Basic lifecycle shown, some confusion	Incomplete or incorrect lifecycle
Auto Layout & Constraints	Responsive layout, proper constraint implementation, accessibility	Good layout design, most constraints work	Functional layout, accessibility gaps	Poor or non-responsive layout
Native Module Integration	Multiple modules integrated correctly with proper delegation	2–3 modules working correctly	1–2 modules attempted	Minimal or incorrect integration

Code Quality	Clean Swift code following best practices and naming conventions	Generally clean, minor style issues	Readable but inconsistent	Difficult to read or unclear
Strategic Thinking	Thoughtful reflection on iOS design philosophy and architecture	Good understandin g of architecture rationale	Basic understandin g present	Minimal or missing reflection
Communication	Clear explanations, well-organized code, excellent comments	Generally clear, good code structure	Readable but lacks detail	Disorganize d or unclear

---

Deliverable Checklist

Submit as PDF or DOCX (3–4 pages) containing:

- [ ] Task 1: Five data type examples + optionals/error handling code + reflection
- [ ] Task 2: Classes vs. Structures comparison + inheritance + protocol examples + OOP reflection
- [ ] Task 3: UIViewController lifecycle diagram + lifecycle implementation + Storyboard pseudo-code + constraints code + design questions
- [ ] Task 4: AppDelegate/SceneDelegate implementation + 3+ native module integration examples + 3 reflection questions (200–250 words)

Resources & References

Official Apple Documentation:

- Swift Programming Language Guide
- UIViewController Lifecycle
- Auto Layout Guide
- AppDelegate Documentation
- SceneDelegate Documentation

Frameworks Overview:

- AVFoundation (Camera/Audio)
- CoreLocation (GPS)

- HealthKit (Health Data)
- Photos Framework

Best Practices:

- Swift API Design Guidelines
- iOS App Architecture
- WWDC Session Videos

Timeline Guidance

Self-Study (6–8 hours):

- Hours 1–2: Task 1 (Swift syntax, data types, optionals)
- Hours 2–4: Task 2 (OOP in Swift, classes, protocols)
- Hours 4–7: Task 3 (UIViewController lifecycle, Storyboards, constraints)
- Hours 7–8: Task 4 (AppDelegate, native modules, reflection)

Online Sessions (1–2 hours):

- 0.5 hours: Task 1–2 discussion (Swift fundamentals, OOP patterns)
- 0.5 hours: Task 3 demo (Storyboards, Auto Layout best practices)
- 0.5 hours: Task 4 walkthrough (App lifecycle, native frameworks)
- 0.5 hour: Q&A and advanced topics

Extension Tasks (Optional, Advanced)

1. **SwiftUI Alternative:** Rewrite the login screen using SwiftUI instead of UIKit with Storyboards
2. **MVVM Architecture:** Implement login flow using MVVM pattern with Combine framework
3. **Networking Integration:** Add API calls to fetch user data in UIViewController lifecycle
4. **Data Persistence:** Implement Core Data or UserDefaults for persisting login credentials
5. **Unit Testing:** Write unit tests for Swift classes using XCTest framework
6. **Custom UIView:** Create a custom UIView subclass with custom drawing and animation
7. **Advanced Constraints:** Build complex responsive layout with size class variations

---

Notes for Instructors

This exercise covers foundational iOS development concepts essential for all native applications. Students develop:

- Swift language mastery: syntax, types, OOP principles
- iOS architecture understanding: view controllers, lifecycle management
- UI design skills: Storyboards, Auto Layout, responsive layouts
- Framework integration: native modules for camera, location, health
- Strategic thinking: iOS design philosophy and best practices

Discussion Prompts:

- "Why does Swift emphasize optionals for nil-safety?"
- "What's the difference between value types (structs) and reference types (classes)?"
- "How does UIViewController lifecycle differ from Android Activity lifecycle?"
- "When would you use programmatic constraints vs. Storyboard constraints?"
- "How does AppDelegate differ from SceneDelegate in multitasking scenarios?"

Common Student Mistakes:

- Confusing reference semantics (classes) with value semantics (structs)
- Force-unwrapping optionals without nil checks
- Ignoring UIViewController lifecycle, causing memory leaks
- Creating overly complex constraint hierarchies
- Not requesting proper permissions for native frameworks
- Failing to handle errors in native module integration
- Ignoring Safe Area and device notch considerations

Assessment Notes

Grading Rubric Integration:

- Emphasizes understanding over syntax perfection
- Values proper lifecycle management and state handling
- Recognizes Swift idioms and OOP best practices
- Encourages responsive, accessible UI design

Partial Credit Guidance:

- Swift Fundamentals: 20% (syntax, types, optionals)

- OOP Implementation: 25% (classes, inheritance, protocols)
- UIViewController: 25% (lifecycle, Storyboards, constraints)
- Native Integration: 20% (frameworks, delegation, permissions)
- Strategic Thinking: 10% (reflection and architecture understanding)