Section I: Swift Language

1. What are the main differences between let and var in Swift?

Mutability (Can the value change?)

- `let` is used to declare a constant. Once a value is assigned, it cannot be changed.
- `var` is used to declare a variable. Its value can be changed after assignment.

Use Case

- Use `let` for values that should never change, like configuration constants, fixed numbers, or names.
- Use `var` for values that might change, like user input, counters, or temporary values.

2. How do you handle optionals in Swift? Explain with examples.

- An optional is a variable that can have a value or be nil (no value).Declared by adding ? after the type.
- Example**:**

```
var name: String? = "Alice"
    name = nil
```

- You can safely unwrap optionals using:

Optional Binding (if let): Check if the optional has a value, and safely use it.

```
if let username = name {
        print("Hello, \(username)")
} else {
        print("No name found")
}
```

Guard Let: Used inside functions to exit early if the optional is nil.

```
func greet(_ name: String?) {
        guard let username = name else {  print("No name") ; return }
        print("Hello, \(username)")}
}
```

Force Unwrapping (!): Directly accesses the value of an optional. Dangerous if the optional is nil will crash. Use only if you are sure the value exists.

```
var firstName: String? = "Alice"
print("Hello, \(firstName!)") // Prints: Hello, Alice
var lastName: String? = nil
print(lastName!) //  CRASH! Cannot unwrap nil
```

3. What are Swift Concurrency features (async/await), and how do they differ from Grand Central Dispatch (GCD) or threads?

Swift Concurrency (async/await)

1. **Purpose**
   Swift Concurrency is a modern way to handle asynchronous tasks (tasks that take time, like fetching data from the internet) without blocking the main thread.
2. **Key Features**

- async functions: Functions that can perform asynchronous operations.
  Example:

```
func fetchData() async -> String {
        return "Hello"
}
```

- await keyword: Waits for an async function to finish without freezing the UI.
  Example:
  ```
  let result = await fetchData()
  print(result) // Prints "Hello"
  ```

- Structured concurrency: Swift automatically manages tasks in a hierarchy, making it easier to cancel or handle multiple tasks safely.
- Task creation: You can create parallel tasks using Task { }

3. **Advantages of async/await**

- Code looks sequential, easy to read, and understand.
- Less chance of race conditions compared to manually managing threads.
- Automatic cancellation and error handling in tasks.

Grand Central Dispatch (GCD) / Threads

1. **Purpose**
   GCD is an older way to run code asynchronously or in parallel. Threads are low-level execution units.
2. **How it works**

- You dispatch work to a queue (main, global, or custom queue).
- You manually handle background tasks and UI updates.
- Example:
  ```
  DispatchQueue.global().async {
      let result = heavyWork()
          DispatchQueue.main.async {
              print(result)
          }
      }
  ```

3. **Challenges with GCD/Threads**

- Code can become nested and hard to read (“callback hell”).

- Error handling and cancellation are manual and tricky.
- Managing shared data can cause race conditions.

Key Differences: async/await vs GCD/Threads

- Syntax: async/await is clear and sequential-looking; GCD uses nested closures, harder to read.
- Error handling: async/await has built-in try/catch; GCD requires manual error propagation.
- Cancellation: async/await supports automatic task cancellation; GCD requires manual cancellation.
- Safety: async/await reduces race conditions with structured concurrency; GCD needs manual management.
- Parallel tasks: async/await uses Task { } or async let; GCD uses DispatchQueue.asyn

Section II: UIKit & SwiftUI

1. What is SwiftUI, and how does it differ from UIKit in terms of UI development?

SwiftUI:

- Declarative framework: You describe *what* the UI should look like, not *how* to create it step by step.
- Less code: You can create a complex UI with fewer lines of code.
- Real-time preview: Xcode shows live previews of your UI while you code.
- Cross-platform: Works on iOS, macOS, watchOS, and tvOS.
- State-driven: UI automatically updates when underlying data changes using `@State`, `@Binding`, or `@ObservedObject`.

  Easier animations: Built-in support for smooth animations with less effort.

UIKit:

- Imperative framework: You describe *how* to create UI step by step.
- More code: Requires writing more lines to achieve the same UI.

  Storyboard/XIB: Visual tools for layout; changes in code may need manual sync with UI.
- iOS only: Works only on iOS (and tvOS partially).
- Manual state handling: You must update the UI manually when data changes.

  Animations require more setup: Need `UIView.animate` or other APIs.

2. How do you manage state in SwiftUI? Explain with examples (@State, @Binding, @ObservedObject, @StateObject).

Managing State in SwiftUI

In SwiftUI, state refers to the data that can change over time and affects what is shown in the UI. SwiftUI provides property wrappers to manage state in different situations: @State, @Binding, @ObservedObject, @StateObject.

1. **@State**

- Use @State for local state inside a single view.
- SwiftUI automatically updates the view when the state changes.
- Example:

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

          var body: some View {

             VStack {

                Text("Count: \(count)")

                Button("Increment") {

                   count += 1

                }

             }

          }

  }
```

2. **@Binding**

- Use @Binding to share state between a parent and child view.
- It creates a two-way connection: a child can read and modify the parent's state.
- Example:

```
struct ParentView: View {
@State private var isOn = false

                var body: some View {

                   ToggleView(isOn: $isOn) // Pass binding to child

                }

          }
```

```
struct ToggleView: View {
 @Binding var isOn: Bool

        var body: some View {

            Toggle("Switch", isOn: $isOn)

        }

}
```

3. **@ObservedObject**

- Use @ObservedObject for external data models that conform to ObservableObject.

- SwiftUI updates the view whenever a published property changes.
- Example:

```
 class UserModel: ObservableObject {
        @Published var name = "Alice"
 }
```

```
struct UserView: View {
 @ObservedObject var user: UserModel

          var body: some View {

             VStack {

                Text("Name: \(user.name)")

                Button("Change Name") {

                   user.name = "Bob"

                }

             }

          }

}
```

4. **@StateObject**

- Use @StateObject to create and own an ObservableObject in a view.
- Use it when the view is responsible for initializing the model.
- It ensures the object is created only once for the view's lifecycle.
- Example:

```
 class CounterModel: ObservableObject {
          @Published var count = 0
 }
```

```
struct CounterView: View {
 @StateObject private var counter = CounterModel()

          var body: some View {

             VStack {

                 Text("Count: \(counter.count)")

                 Button("Increment") {

                     counter.count += 1

                 }

             }

      }

}
```

3. What is the @State property wrapper in SwiftUI, and how is it used?

- @State is used for managing local, temporary state inside a view.
- When the state changes, the view updates automatically.
- Example:

```
struct CounterView: View {

@State private var count = 0

  var body: some View {

          VStack {

          Text("Count: \(count)")

          Button("Increase") { count += 1 }

          }

    }

}
```

4. How can you implement navigation in a SwiftUI application?

In SwiftUI, navigation is handled using `NavigationView` as a container and `NavigationLink` to move between views. `NavigationLink`.

Example:

```
struct DetailView: View {
    var body: some View {
        Text("Detail View")
            .navigationTitle("Detail")
    }
}
struct ContentView: View {
    @State private var navigate = false
    var body: some View {
        NavigationView {
            VStack {
                Button("Go to Detail") { navigate = true }
                    .padding()
                    .background(Color.blue)
                NavigationLink("", destination: DetailView(), isActive: $navigate)
            }
            .navigationTitle("Home")
        }
    }
}
```

Section III: Data Persistence & Security

1. What is UserDefaults in iOS, and when should it be used? Provide examples.

1. **UserDefaults**
   - `UserDefaults` is a storage system in iOS for saving small pieces of data.
   - It stores data persistently, meaning the data remains even after the app is closed and reopened.
   - It's best for simple data, like settings, preferences, or flags.
     It is not suitable for large data (like images, videos, or big arrays) or sensitive data (like passwords use Keychain instead).
2. **When to Use UserDefaults**
   - Saving user preferences, like theme (dark/light mode).
   - Remembering if the user is logged in.
   - Storing simple app settings or configuration.
3. **Example:**

   UserDefaults.standard.set(true, forKey: "isLoggedIn")

   let loggedIn = UserDefaults.standard.bool(forKey: "isLoggedIn")

2. What is the Keychain in iOS, and how is it different from UserDefaults?

- Keychain is a secure storage system in iOS.It is used to store sensitive information, like passwords, tokens, or private keys.Data stored in Keychain is encrypted and protected by the device’s security (Touch ID, Face ID, or passcode).
  Data persists even if the app is deleted and reinstalled (optional, depending on settings).
- **Difference Between Keychain and UserDefaults**
  1. Purpose: Keychain is for sensitive data, while UserDefaults is for small, non-sensitive data like settings or preferences.
  2. Security: Keychain data is encrypted and secure, but UserDefaults data is not encrypted by default.

3. Persistence: Keychain can survive app uninstall and reinstall, while UserDefaults data is deleted when the app is uninstalled.
4. Usage Complexity: Keychain requires more code to store and retrieve data, while UserDefaults is very simple to use.
5. Examples: Keychain is used for passwords, API tokens, or biometric keys. UserDefaults is used for theme preference, login flags, counters, or simple app settings..

3. How would you securely store a user’s login credentials in iOS?

To securely store a user’s login credentials in iOS:

- Use Keychain: Encrypts and safely stores sensitive data like passwords or tokens.
- Avoid UserDefaults: It is not secure for credentials.
- Prefer tokens over raw passwords: Store authentication tokens when possible. Add biometrics: Use Face ID or Touch ID for extra security.
- Encrypt if needed: Always encrypt sensitive data if not using Keychain.