Swift Concurrency Detailed Handbook

1. What is Async/Await?

`async` and `await` let you write asynchronous, non-blocking code that looks and behaves like synchronous code.

```
// Old GCD approach
DispatchQueue.global().async {
    let data = fetchData()
    DispatchQueue.main.async {
        updateUI(with: data)
    }
}

// New async/await approach
let (data, _) = try await URLSession.shared.data(from: url)
updateUI(with: data)
```

2. Basic Rules of Async/Await

- You can only call `await` inside an `async` function.
- Combine `try` with `await` for throwable async calls.
- Async code suspends tasks instead of blocking threads.
- Use `.task {}` or `.onAppear {}` in SwiftUI for async-safe contexts.

3. Managing UI and MainActor

All UI work must occur on the main thread. Use `@MainActor` or `await MainActor.run {}` for safety.

```
@MainActor
func updateUI() {
    title = "Loaded!"
}

func fetchData() async {
    let data = try await URLSession.shared.data(from: url)
    await MainActor.run {
        self.posts = data
    }
}
```

4. Delaying Async Work with Task.sleep

`Task.sleep` allows non-blocking delays (measured in nanoseconds).

```
try await Task.sleep(nanoseconds: 1_000_000_000) // 1 second delay
```

5. Handling Multiple API Calls

Scenario 1: Continue even if one API fails (use async let + try?)

```
async let posts = try? fetchPosts()
async let users = try? fetchUsers()
let results = await (posts, users)
```

Scenario 2: Stop all if one API fails (use withThrowingTaskGroup)

```
try await withThrowingTaskGroup(of: Void.self) { group in
    group.addTask { _ = try await fetchPosts() }
    group.addTask { _ = try await fetchUsers() }
    try await group.waitForAll()
}
```

6. Structured Concurrency in Action

Structured concurrency ensures all async work is properly scoped, cancelled, and error-handled.

```
func loadDashboard() async {
    async let posts = fetchPosts()
    async let users = fetchUsers()
    (self.posts, self.users) = try await (posts, users)
}
```

7. Integrating Async/Await with SwiftUI

```
struct DashboardView: View {
    @StateObject private var vm = DashboardViewModel()
    var body: some View {
        List(vm.posts) { post in Text(post.title) }
        .task { await vm.loadDashboard() }
    }
}
```

8. Async/Await vs Grand Central Dispatch (GCD)

| Goal | GCD | Async/Await |

|------|-----|--------------|

| Background Work | DispatchQueue.global().async | Task { } |

Swift Concurrency Detailed Handbook

Run on Main Thread	DispatchQueue.main.async	@MainActor / MainActor.run
Delay	asyncAfter	Task.sleep
Group Tasks	DispatchGroup	async let / TaskGroup
Set Priority	QoS	Task(priority:)
Thread Safety	Semaphores	Actor Isolation
Cancellation	Manual	Built-in
Synchronization	Barrier	Structured concurrency

9. Visual Flow Summary

- Async/Await Flow: Task created -> Suspends -> Background -> Resumes on main actor
- TaskGroup: Creates parallel child tasks -> Aggregates results -> Cancels all on failure
- MainActor: Ensures serial execution of UI-related tasks on the main thread

10. Best Practices

- Mark UI functions with `@MainActor`.
- Use `await MainActor.run {}` for isolated UI updates.
- Use `try?` to safely ignore expected failures.
- Replace `DispatchQueue` with `Task` or `TaskGroup`.
- Avoid blocking calls like `sleep()`; use `Task.sleep()`.
- Handle `Task.isCancelled` inside loops.

11. TL;DR Summary

async = can suspend
await = wait for result
@MainActor = UI thread safe
Task.sleep = async delay
async let = parallel calls
withThrowingTaskGroup = cancel on first failure
Task(priority:) = replaces QoS

Swift Concurrency replaces DispatchQueue, DispatchGroup, QoS, asyncAfter, and manual synchronization.