Swift 6 & Concurrency

Les Fondamentaux Modernes

Atelier Développeurs iOS pour Le Bon Coin

Agenda (3h)

- Module 1 (30min) : Introduction & Problématique
- Module 2 (50min) : async/await Fondamentaux
- Module 3 (40min) : Structured Concurrency
- Module 4 (40min) : Actors & Thread Safety
- Module 5 (20min) : Sendable & Data Race Safety

Module 1

Introduction & Problématique

Timeline Swift

- Swift 5.0 (2019) → Stable ABI
- Swift 5.5 (2021) → async/await introduit
- Swift 5.7 (2022) → Actors matures
- Swift 6.0 (2024) → Complete Concurrency Checking

Objectif Principal de Swift 6

Éliminer les data races au compile-time

Exemple : Le Problème

```
// Swift 5.x - Compile mais peut crasher
class Counter {
    var value = 0

    func increment() {
        value += 1  // ⚠️ Potentiel data race
    }
}

let counter = Counter()
DispatchQueue.concurrentPerform(iterations: 1000) { _ in
    counter.increment()  // 💥
}
```

Swift 6 : Ne Compile Pas

```
// Swift 6 - Error au compile-time
class Counter {
    var value = 0

    func increment() {
        value += 1
    }
}

let counter = Counter()
DispatchQueue.concurrentPerform(iterations: 1000) { _ in
    counter.increment()
    // ❌ Error: "Counter is not Sendable"
}
```

Les 3 Piliers de Swift 6

1. async/await : Syntaxe moderne pour code asynchrone
2. Actors : Isolation automatique pour thread-safety
3. Sendable : Garanties compile-time sur le partage de données

Avant Swift 5.5 : Completion Blocks & Callbacks

```swift
// L'enfer des callbacks
func fetchUserProfile(completion: @escaping (Data?, Error?) -> Void) {
    fetchFromAPI { data, error in
        guard let data = data else {
            completion(nil, error)
            return
        }
        processImage(data) { image in
            updateUI(image)
            completion(data, nil)
        }
    }
}

// Appel :
fetchUserProfile { data, error in
    print("Résultat prêt !")
}
```

The moumObjective-C : Blocks et Pyramide du Destin

```objc
// Objective-C style (2015)
[self fetchFromAPI:^(NSData *data, NSError *error) {
    if (data) {
        [self processImage:data completion:^(UIImage *image) {
            [self updateUI:image];
            NSLog(@"Fini !");
        }];
    }
}];
```

Pourquoi la Concurrence ?

Apps Modernes = Multiples Tâches Simultanées

```
App Moderne :
├── Fetch data depuis API       (réseau)
├── Process images              (CPU)
├── Update UI                   (main thread)
├── Save to database           (I/O)
└── Sync avec iCloud           (réseau)

Tout ça doit se passer SANS bloquer l'UI !
```

Exemple : Mauvaise Approche

```
// ❌ MAUVAIS - Bloque l'UI
func loadUserProfile() {
    let data = fetchFromAPI()          // ⏱️ 2 secondes
    let image = processImage(data)     // ⏱️ 1 seconde
    updateUI(image)                    // ⏱️ instant

    // Total: 3 secondes de freeze ! 😱
}
```

Objectif : Non-Blocking

```
// ✅ BON - UI reste fluide
func loadUserProfile() async {
    let data = await fetchFromAPI()
    let image = await processImage(data)
    await updateUI(image)

    // L'utilisateur peut continuer d'utiliser l'app !
}
```

L'Évolution : Callback Hell

```swift
// 😨 Pyramid of Doom
func fetchUser(id: String, completion: @escaping (User?, Error?) -> Void) {
    URLSession.shared.dataTask(with: url) { data, response, error in
        guard let data = data else {
            completion(nil, error)
            return
        }

        parseJSON(data) { user, error in
            guard let user = user else {
                completion(nil, error)
                return
            }

            fetchAvatar(user.avatarURL) { image, error in
                var updatedUser = user
                updatedUser.avatar = image
                completion(updatedUser, nil)
            }
        }
    }.resume()
}
```

L'Évolution : async/await

```swift
// Clair et lisible
func fetchUser(id: String) async throws -> User {
    let (data, _) = try await URLSession.shared.data(from: url)
    var user = try JSONDecoder().decode(User.self, from: data)
    user.avatar = try await fetchAvatar(user.avatarURL)
    return user
}

// Utilisation simple :
let user = try await fetchUser(id: "123")
updateUI(user)
```

Module 2

async/await - Les Fondamentaux

Déclarer une Fonction Asynchrone

```swift
// Fonction asynchrone simple
func fetchData() async -> Data {
    // Code asynchrone
}

// Fonction asynchrone qui peut throw
func fetchData() async throws -> Data {
    // Code asynchrone qui peut échouer
}

// Fonction synchrone (normale)
func processData(_ data: Data) -> String {
    // Code synchrone
}
```

Appeler une Fonction Asynchrone

```swift
// ❌ Erreur - await manquant
func loadData() {
    let data = fetchData()  // Compile Error!
}

// ✅ Correct - avec await
func loadData() async {
    let data = await fetchData()
    print(data)
}

// ✅ Avec error handling
func loadData() async {
    do {
        let data = try await fetchData()
        print(data)
    } catch {
        print("Error: \(error)")
    }
}
```

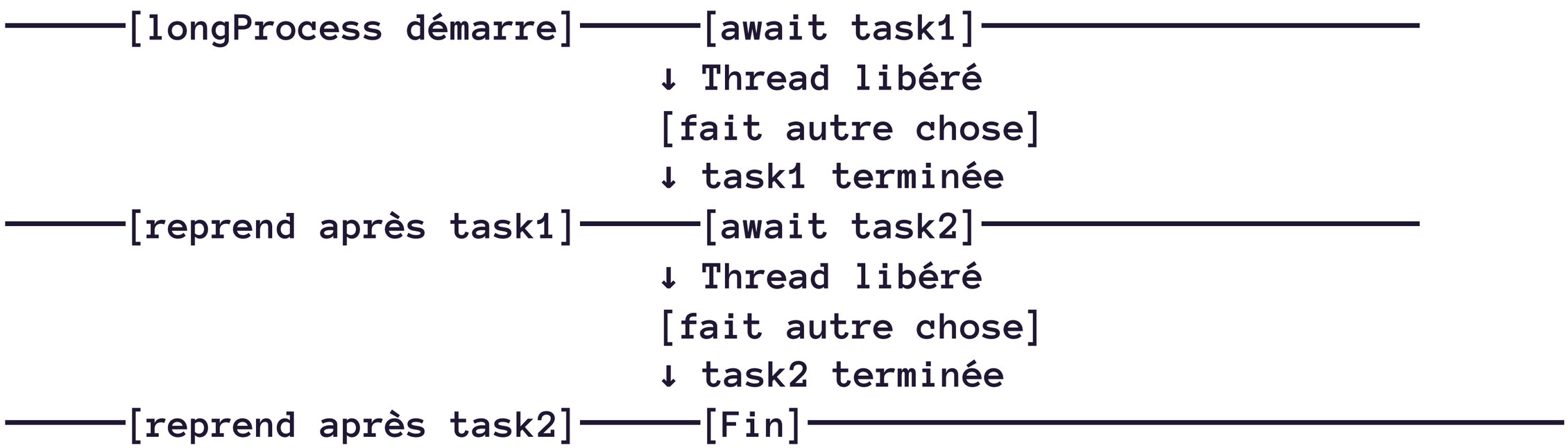
Règle : async Propagation

```
// Si une fonction appelle du code async,
// elle doit être async
func processAndSave() async throws {
    let data = try await fetchData()      // async
    let processed = process(data)         // sync
    try await save(processed)             // async
}

// ❌ Ne compile pas si processAndSave n'est pas async
```

Suspension Points

```
func longProcess() async {
    print("1. Début")

    await task1()  // 🔵 Suspension point
    print("2. Task1 terminée")

    await task2()  // 🔵 Suspension point
    print("3. Task2 terminée")

    print("4. Fin")
}
```

Visualisation : Timeline du Thread



Exemple Concret

```swift
func downloadAndProcess() async {
    print("Début téléchargement")

    // Suspension : thread libéré pendant le réseau
    let data = await download()  // 2 secondes

    print("Téléchargement terminé, processing...")

    // Suspension : thread libéré pendant le processing
    let result = await process(data)  // 1 seconde

    print("Tout terminé !")
}
```

async let : Parallélisme Simple

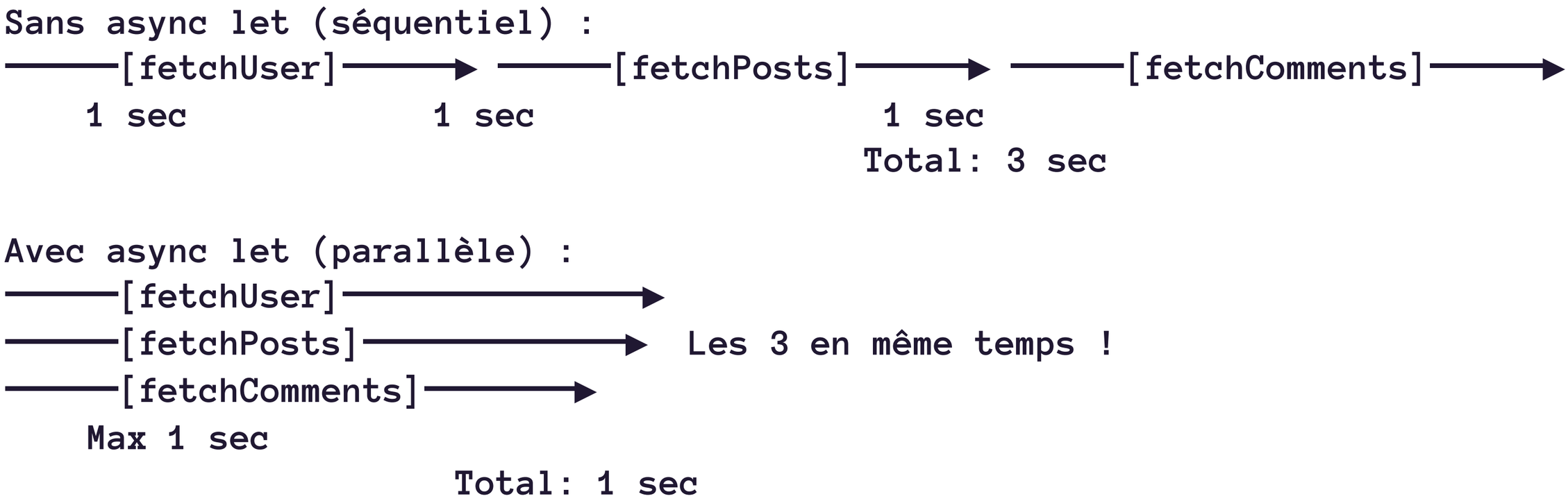
Exécution Séquentielle (Lente)

```
func fetchAllData() async throws -> (User, [Post], [Comment]) {
    let user = try await fetchUser()          // ⏱️ 1 sec
    let posts = try await fetchPosts()        // ⏱️ 1 sec
    let comments = try await fetchComments() // ⏱️ 1 sec

    return (user, posts, comments)
    // Total: 3 secondes 😰
}
```

async let : Exécution Parallèle (Rapide)

```swift
func fetchAllData() async throws -> (User, [Post], [Comment]) {
    // Démarrer les 3 en parallèle
    async let user = fetchUser()
    async let posts = fetchPosts()
    async let comments = fetchComments()

    // Attendre que tout soit fini
    return try await (user, posts, comments)
    // Total: ~1 seconde (le plus lent) 🚀
}
```

Comment ça Marche ?



Règles Importantes : async let

```
func example() async throws {
    // 1. Déclarer avec async let
    async let result1 = fetch1()
    async let result2 = fetch2()

    // 2. DOIT await avant la fin du scope
    let (r1, r2) = try await (result1, result2)

    // 3. Si on quitte le scope sans await, erreur !
}
```

❌ Erreur Courante

```
// ❌ Erreur - async let non awaité
func wrong() async {
    async let data = fetch()
    return  // Compile Error!
}

// ✅ Correct
func correct() async {
    async let data = fetch()
    let result = await data  // OK
    return
}
```

Exemple Pratique : Image Thumbnails

```swift
func generateThumbnails(_ images: [UIImage]) async -> [UIImage] {
    // Générer 4 thumbnails en parallèle
    async let thumb1 = resize(images[0])
    async let thumb2 = resize(images[1])
    async let thumb3 = resize(images[2])
    async let thumb4 = resize(images[3])

    return await [thumb1, thumb2, thumb3, thumb4]
    // 4x plus rapide que séquentiel !
}
```

Gestion d'Erreurs : try await

```swift
func fetchAndParse() async throws -> User {
    // Les deux peuvent throw
    let data = try await fetchData()
    let user = try parseJSON(data)
    return user
}

// Utilisation
do {
    let user = try await fetchAndParse()
    print("Success: \(user.name)")
} catch let error as NetworkError {
    print("Network error: \(error)")
} catch let error as ParseError {
    print("Parse error: \(error)")
} catch {
    print("Unknown error: \(error)")
}
```

Erreurs Partielles avec async let

```
func fetchMultiple() async throws -> (User?, [Post]?) {
    async let user = fetchUser()          // Peut throw
    async let posts = fetchPosts()        // Peut throw

    // Si UNE des deux throw, TOUT throw
    return try await (user, posts)
}
```

Alternative : Gérer Erreurs Individuellement

```
func fetchMultipleSafe() async -> (User?, [Post]?) {
    async let userResult = Result { try await fetchUser() }
    async let postsResult = Result { try await fetchPosts() }

    let (user, posts) = await (userResult, postsResult)

    return (try? user.get(), try? posts.get())
}
```

Pattern : Retry avec async/await

```swift
func fetchWithRetry(maxAttempts: Int = 3) async throws -> Data {
    var lastError: Error?

    for attempt in 1...maxAttempts {
        do {
            return try await fetchData()
        } catch {
            lastError = error
            print("Attempt \(attempt) failed: \(error)")

            if attempt < maxAttempts {
                // Exponential backoff
                try await Task.sleep(for: .seconds(attempt))
            }
        }
    }

    throw lastError ?? FetchError.unknown
}
```

Module 3

Structured Concurrency & Task Management

Problème : Unstructured Concurrency

```
// ❌ Dangereux - Tasks "orphelines"
func badExample() {
    Task {
        await longRunningOperation1()
    }

    Task {
        await longRunningOperation2()
    }

    return  // Function termine, mais les Tasks continuent !
}
```

Problèmes Unstructured

- Tasks peuvent outlive leur parent
- Pas de gestion des erreurs centralisée
- Impossible d'annuler facilement
- Memory leaks potentiels
- Comportement non-déterministe

Solution : Structured Concurrency

```swift
// ✅ Safe - Toutes les tasks sont gérées
func goodExample() async throws {
    try await withThrowingTaskGroup(of: Void.self) { group in
        group.addTask { await longRunningOperation1() }
        group.addTask { await longRunningOperation2() }

        // Attendre que TOUTES les tasks soient finies
        try await group.waitForAll()
    }
    // Ici, GARANTIE que tout est terminé
}
```

Avantages Structured Concurrency

- ✅ Lifetime contrôlé
- ✅ Erreurs propagées automatiquement
- ✅ Annulation en cascade
- ✅ Pas de leaks
- ✅ Prévisible et testable

withTaskGroup - Sans Erreurs

```swift
func processImages(_ images: [UIImage]) async -> [ProcessedImage] {
    await withTaskGroup(of: ProcessedImage.self) { group in
        // Ajouter une task pour chaque image
        for image in images {
            group.addTask {
                await processImage(image)
            }
        }

        // Collecter tous les résultats
        var results: [ProcessedImage] = []
        for await result in group {
            results.append(result)
        }
        return results
    }
}
```

withThrowingTaskGroup - Avec Erreurs

```
func fetchAllArticles(ids: [String]) async throws -> [Article] {
    try await withThrowingTaskGroup(of: Article.self) { group in
        for id in ids {
            group.addTask {
                try await fetchArticle(id: id)
            }
        }

        // Si UNE task throw, tout le groupe throw
        var articles: [Article] = []
        for try await article in group {
            articles.append(article)
        }
        return articles
    }
}
```

Gestion d'Erreurs Partielles

```
func fetchWithPartialFailure(ids: [String]) async -> [Article] {
    await withTaskGroup(of: Article?.self) { group in
        for id in ids {
            group.addTask {
                // Capturer l'erreur, retourner nil
                try? await fetchArticle(id: id)
            }
        }

        var articles: [Article] = []
        for await article in group {
            if let article = article {
                articles.append(article)
            }
        }
        return articles
    }
    // Résultat: articles qui ont réussi, ignore les erreurs
}
```

Pattern : Limitation de Concurrence

```swift
func processWithLimit(
    items: [Item],
    maxConcurrent: Int
) async -> [Result] {
    await withTaskGroup(of: Result.self) { group in
        var results: [Result] = []
        var iterator = items.makeIterator()
        var activeCount = 0

        // Lancer les premières tasks
        while activeCount < maxConcurrent,
              let item = iterator.next() {
            group.addTask { await process(item) }
            activeCount += 1
        }

        // Pour chaque résultat, lancer une nouvelle task
        for await result in group {
            results.append(result)
            if let nextItem = iterator.next() {
                group.addTask { await process(nextItem) }
            }
        }
        return results
    }
}
```

Task Lifecycle & Cancellation

```swift
// Créer une task
let task = Task {
    await longRunningOperation()
}

// Attendre le résultat
let result = await task.value

// Annuler
task.cancel()

// Vérifier si annulée
if Task.isCancelled {
    return  // Sortir tôt
}
```

Cooperative Cancellation

```swift
func processData(_ items: [Item]) async throws {
    for item in items {
        // Vérifier cancellation régulièrement
        try Task.checkCancellation()

        await process(item)
    }
}

// Alternative
func processData2(_ items: [Item]) async {
    for item in items {
        if Task.isCancelled {
            print("Task cancelled, stopping...")
            return
        }
        await process(item)
    }
}
```

Pattern : Debouncing

```swift
class SearchViewModel: ObservableObject {
    @Published var searchText = ""
    private var searchTask: Task<Void, Never>?

    func onSearchTextChanged() {
        // Annuler recherche précédente
        searchTask?.cancel()

        // Créer nouvelle recherche avec délai
        searchTask = Task {
            try? await Task.sleep(for: .milliseconds(300))

            guard !Task.isCancelled else { return }

            await performSearch(searchText)
        }
    }
}
```

Task Priority

```swift
// Priorités disponibles
Task(priority: .high) {
    await criticalOperation()
}

Task(priority: .medium) {
    await normalOperation()
}

Task(priority: .low) {
    await backgroundOperation()
}

Task(priority: .background) {
    await heavyComputation()
}
```

Module 4

Actors & Thread Safety

Le Problème : Data Races

```swift
// ❌ DANGEREUX - Data race !
class Counter {
    var value = 0

    func increment() {
        value += 1  // Pas thread-safe !
    }
}

let counter = Counter()

// Lancer 1000 incréments en parallèle
await withTaskGroup(of: Void.self) { group in
    for _ in 0..<1000 {
        group.addTask {
            counter.increment()
        }
    }
}

print(counter.value)
// Devrait être 1000, mais... 987, 992, aléatoire ! 💥
```

Visualisation du Data Race

```
Thread 1:                    Thread 2:
1. Lit value (0)             1. Lit value (0)
2. Calcule 0 + 1 = 1         2. Calcule 0 + 1 = 1
3. Écrit value = 1           3. Écrit value = 1

Résultat final: value = 1 (devrait être 2!)
```

Solutions Traditionnelles

```swift
// Option 1: Locks (verbeux, dangereux)
class Counter {
    private var value = 0
    private let lock = NSLock()

    func increment() {
        lock.lock()
        value += 1
        lock.unlock()  // Oubli = deadlock !
    }
}

// Option 2: DispatchQueue (mieux mais verbeux)
class Counter {
    private var value = 0
    private let queue = DispatchQueue(label: "counter")

    func increment() {
        queue.sync {
            value += 1
        }
    }
}
```

Solution : Actors

```swift
// ✅ SAFE - Actor garantit thread-safety
actor Counter {
    var value = 0

    func increment() {
        value += 1  // Thread-safe automatiquement !
    }

    func getValue() -> Int {
        value
    }
}

let counter = Counter()

// Utilisation avec await
await withTaskGroup(of: Void.self) { group in
    for _ in 0..<1000 {
        group.addTask {
            await counter.increment()
        }
    }
}

print(await counter.getValue())  // TOUJOURS 1000 ! ✅
```

Comment ça Marche ?

Actor = Mailbox Pattern

```
Thread 1 → [increment] →┐
Thread 2 → [increment] →├──→ Actor Counter
Thread 3 → [getValue]  →┘     value = 0

Timeline:
1. Thread 1 envoie "increment" → Actor traite → value = 1
2. Thread 2 envoie "increment" → Actor traite → value = 2
3. Thread 3 envoie "getValue"  → Actor traite → retourne 2

Pas de conflit possible !
```

Règles des Actors

```swift
actor DataStore {
    // 1. Propriétés privées (isolation)
    private var data: [String: Data] = [:]

    // 2. Méthodes accessibles avec await
    func save(key: String, value: Data) {
        data[key] = value  // Thread-safe
    }

    func load(key: String) -> Data? {
        data[key]  // Thread-safe
    }

    // 3. nonisolated pour méthodes synchrones
    nonisolated func description() -> String {
        "DataStore actor"
        // Pas d'accès aux propriétés
    }
}

// Utilisation
let store = DataStore()
await store.save(key: "user", value: userData)
let data = await store.load(key: "user")
```

Actor Reentrancy ⚠️

```
actor DownloadManager {
    private var cache: [URL: Data] = [:]

    func download(url: URL) async throws -> Data {
        // Si déjà en cache
        if let cached = cache[url] {
            return cached
        }

        // Télécharger (suspension point!)
        let data = try await URLSession.shared.data(from: url).0

        // ⚠️ ATTENTION : L'actor peut avoir été accédé
        // pendant le download !
        // cache pourrait avoir changé !

        cache[url] = data
        return data
    }
}
```

Solution : Re-vérifier après await

```swift
actor DownloadManager {
    private var cache: [URL: Data] = [:]

    func download(url: URL) async throws -> Data {
        if let cached = cache[url] {
            return cached
        }

        let data = try await URLSession.shared.data(from: url).0

        // Re-vérifier le cache (peut-être ajouté entre temps)
        if let cached = cache[url] {
            return cached
        }

        cache[url] = data
        return data
    }
}
```

@MainActor : UI Thread Safety

Le Problème UI

```
// ❌ ERREUR - UI update hors main thread !
func loadData() async {
    let data = await fetchFromAPI()

    // Cette ligne peut s'exécuter sur n'importe quel thread
    label.text = data  // 💥 Crash ou comportement bizarre !
}
```

Solution : @MainActor

```swift
// Méthode 1 : Marquer la classe entière
@MainActor
class ViewModel: ObservableObject {
    @Published var text = ""
    @Published var isLoading = false

    // Toutes ces méthodes sont garanties sur main thread
    func loadData() async {
        isLoading = true

        let data = await fetchFromAPI()

        text = data  // Automatiquement sur main thread !
        isLoading = false
    }
}
```

@MainActor sur Méthode

```
class Service {
    @MainActor
    func updateUI(data: String) {
        label.text = data  // Garanti main thread
    }

    func processData() async {
        let data = await fetch()
        await updateUI(data: data)  // Bascule sur main thread
    }
}
```

MainActor.run

```
func example() async {
    let data = await fetchData()

    // Forcer exécution sur main thread
    await MainActor.run {
        label.text = data
        progressBar.isHidden = true
        reloadButton.isEnabled = true
    }
}
```

@MainActor avec SwiftUI

```swift
// SwiftUI Views sont automatiquement @MainActor
struct ContentView: View {
    @StateObject var viewModel = ViewModel()  // @MainActor

    var body: some View {
        Text(viewModel.text)
            .task {
                // Démarre sur main actor
                await viewModel.loadData()
                // Retourne sur main actor
            }
    }
}

// Observable macro (iOS 17+) avec @MainActor
@Observable
@MainActor
class ViewModel {
    var text = ""

    func load() async {
        text = await fetchData()  // Safe
    }
}
```

nonisolated

```swift
@MainActor
class ViewModel {
    var data = ""

    // Cette méthode DOIT être sur main thread
    func updateUI() {
        // Access data OK
    }

    // Cette méthode peut être appelée de n'importe où
    nonisolated func heavyComputation() -> Int {
        // ❌ Ne peut PAS accéder à data
        // ✅ Peut faire du calcul pur
        return (1...1000).reduce(0, +)
    }
}
```

Module 5

Sendable & Data Race Safety

Qu'est-ce que Sendable ?

"Ce type peut être partagé entre threads sans danger"

```swift
// Types Sendable par défaut
let number: Int = 42                    // ✅ Sendable (value type)
let text: String = "Hello"              // ✅ Sendable (value type)
let array: [Int] = [1, 2, 3]            // ✅ Sendable (éléments Sendable)

// Types NON-Sendable par défaut
class User {                            // ❌ Pas Sendable
    var name: String                    // (reference type mutable)
}
```

Conformance Automatique

```swift
// Struct avec propriétés Sendable → Sendable automatiquement
struct Article: Sendable {  // ✅ OK
    let id: UUID
    let title: String
    let content: String
}

// Struct avec propriété non-Sendable → Erreur
struct Post: Sendable {     // ❌ Erreur
    let title: String
    var author: User        // User n'est pas Sendable
}
```

Rendre un Type Sendable

```swift
// Option 1 : Struct immuable
struct Point: Sendable {
    let x: Int
    let y: Int
}

// Option 2 : Class immuable (final + let)
final class Configuration: Sendable {
    let apiKey: String
    let baseURL: URL

    init(apiKey: String, baseURL: URL) {
        self.apiKey = apiKey
        self.baseURL = baseURL
    }
}

// Option 3 : Actor (toujours Sendable)
actor Cache: Sendable {  // Implicite
    private var data: [String: Data] = [:]
}
```

@unchecked Sendable ⚠️

```
// Utilisez SEULEMENT si vous garantissez
// thread-safety manuellement
class LegacyCache: @unchecked Sendable {
    private var data: [String: Data] = [:]
    private let lock = NSLock()

    func save(key: String, value: Data) {
        lock.lock()
        data[key] = value
        lock.unlock()
    }
}
```

Closures Sendable

```swift
// ❌ Erreur - capture de variable mutable
func example() {
    var count = 0

    Task {
        count += 1  // ❌ Erreur: mutation de capture non-Sendable
    }
}

// ✅ Solution 1 : Capturer la valeur
func example() {
    var count = 0
    let capturedCount = count

    Task {
        print(capturedCount)  // OK
    }
}

// ✅ Solution 2 : Actor
actor Counter {
    var count = 0
    func increment() { count += 1 }
}

func example() {
    let counter = Counter()
    Task {
        await counter.increment()  // OK
    }
}
```

@Sendable Closures

```swift
// Fonction qui prend une closure @Sendable
func performAsync(
    operation: @Sendable () async -> Void
) async {
    await operation()
}

// ✅ OK - closure ne capture rien
performAsync {
    await someWork()
}

// ❌ Erreur - capture non-Sendable
var mutableState = State()
performAsync {
    mutableState.update()  // Erreur
}

// ✅ OK - capture Sendable
let immutableValue = 42
performAsync {
    print(immutableValue)  // OK
}
```

Exemples Pratiques

```swift
// URLSession - retourne Sendable
func fetch() async throws -> Data {
    let (data, _) = try await URLSession.shared.data(from: url)
    return data  // Data est Sendable ✅
}

// TaskGroup avec Sendable
func processItems(_ items: [Item]) async -> [Result] {
    await withTaskGroup(of: Result.self) { group in
        // Result doit être Sendable
        for item in items {
            group.addTask {
                return process(item)
            }
        }

        var results: [Result] = []
        for await result in group {
            results.append(result)
        }
        return results
    }
}
```

Models Typiques

```swift
// Pour API responses
struct User: Codable, Sendable {
    let id: UUID
    let name: String
    let email: String
}

// Pour SwiftUI
struct Article: Identifiable, Sendable {
    let id: UUID
    let title: String
    let content: String
}
```

Récapitulatif : Les 5 Concepts Clés

1. async/await : Syntaxe moderne pour code asynchrone
2. Structured Concurrency : TaskGroup pour parallélisme contrôlé
3. Actors : Thread-safety automatique
4. @MainActor : UI updates garantis sur main thread
5. Sendable : Compile-time safety pour partage entre threads

Avant → Après

```swift
// AVANT (Swift 5 - Callback Hell)
func fetchUser(completion: @escaping (User?, Error?) -> Void) {
    DispatchQueue.global().async {
        URLSession.shared.dataTask(with: url) { data, _, error in
            guard let data = data else {
                DispatchQueue.main.async {
                    completion(nil, error)
                }
                return
            }
            let user = try? JSONDecoder().decode(User.self, from: data)
            DispatchQueue.main.async {
                completion(user, nil)
            }
        }.resume()
    }
}
```

Avant → Après

```swift
// APRÈS (Swift 6 - async/await)
func fetchUser() async throws -> User {
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(User.self, from: data)
}
```

Checklist Migration Swift 6

- ☑ Remplacer completion handlers par async/await
- ☑ Convertir DispatchQueue en actors
- ☑ Marquer ViewModels avec @MainActor
- ☑ Rendre models Sendable
- ☑ Activer Complete Concurrency Checking
- ☑ Corriger warnings progressivement
- ☑ Tests de non-régression

Ressources

- Documentation : docs.swift.org/swift-book/.../concurrency
- WWDC Sessions :
  - WWDC21: Meet async/await in Swift
  - WWDC21: Protect mutable state with Swift actors
  - WWDC22: Eliminate data races using Swift Concurrency
  - WWDC23: Beyond the basics of structured concurrency

Swift Evolution Proposals

- SE-0296 : async/await
- SE-0304 : Structured Concurrency
- SE-0306 : Actors
- SE-0302 : Sendable and @Sendable closures
- SE-0414 : Region based isolation

Questions Fréquentes

Q: Dois-je migrer tout mon code vers async/await ?

R: Non, migration progressive. Nouveaux features d'abord, puis refacto.

Q: async/await remplace complètement GCD ?

R: Pour la plupart des cas oui, mais GCD reste pour cas spécifiques.

Q: Les actors ont-ils un coût performance ?

R: Minimal. Comparable à un dispatch_async.

Q: Puis-je utiliser async/await avec Combine ?

R: Oui ! `.values` convertit un Publisher en AsyncSequence.

Merci !

Questions ?

Prochaine étape : TPs pratiques 🚀