Swift Concurrency: Safeguarding Data with @Sendable and Actors

VINODH KUMAR · Follow

6 min read · Feb 20, 2024

Photo by Paul Esch-Laurent on Unsplash

Before delving deeper, let's recall the issue of data synchronization and data races that we encountered with threads and dispatch queues. When multiple threads or queues access mutable state simultaneously, it opens the door to data races. For example, if 1,000 threads increment a counter concurrently, the count might end up being slightly less than 1,000 due to conflicting writes and reads. This occurs when one thread writes the count between another thread's read and write, causing inconsistencies in the data.

Now, let's investigate the new tools Swift offers to address these data synchronization problems.

Managing Data Races: Introducing "`Sendable`" and "@Sendable"

Previously, to prevent data races when multiple threads accessed mutable state concurrently, we used a lock within a class:

```
class Counter {
 let lock = NSLock()
 var count = 0

 func increment() {
 self.lock.lock()
 defer { self.lock.unlock() }
 self.count += 1
 }
}
```

This ensured safe access to the mutable `count` variable by synchronizing read, update, and write operations.

```
let counter = Counter()
for _ in 0..<workCount {
 Task {
 counter.increment()
 }
}
Thread.sleep(forTimeInterval: 2)
print("counter.count", counter.count) // Outputs: 1000
```

Running this code consistently yields the expected count of 1000, seemingly resolving our data race. However, Swift offers further tools to tackle these issues more effectively.

Swift's compiler is designed to catch potential problems in concurrent contexts. For instance, it disallows capturing mutable variables inside asynchronous contexts, as seen below:

```
func doSomething() {
 var count = 0
 Task {
 print(count) // 🛑 Error: Reference to captured var 'count' in concurrently-exe
 }
}
```

This strictness prevents confusion regarding variable access between different threads or closures. Mutable captures are disallowed to avoid potential race conditions or non-linear behavior.

Even in escaping closures, where mutable captures are allowed, race conditions can occur without compiler warnings:

```
var count = 0
for _ in 0..<workCount {
 Thread.detachNewThread {
 count += 1
 }
}
Thread.sleep(forTimeInterval: 2)
print("count", count) // Outputs a value less than 1000, e.g., 978
```

Swift disallows mutable captures in concurrent contexts to avoid such ambiguities. However, it allows immutable captures without risk:

```
let count = 0
Task {
 print(count)
}
```

Immutable captures pose no risk of race conditions. Even if a variable is mutable, explicitly capturing it as immutable within a closure allows its safe use:

```
var count = 0
Task { [count] in
 print(count)
}
```

Explicit capture makes it clear that the closure uses an immutable snapshot of the variable, disconnected from any external changes.

Understanding `Sendable` Protocol in Swift

Sendable Protocol Basics:

The `Sendable` protocol, although seemingly trivial, holds significant importance in concurrent code safety.

```
public protocol Sendable {
}
```

Types conforming to `Sendable` are deemed safe for concurrent usage, essentially indicating their capability to cross concurrent boundaries without causing race conditions.

Sendable Conformance:

Swift automatically extends `Sendable` conformance to several types, especially value types with sendable properties:

```
// Int conforms to Sendable
let count = 0
Task {
 print(count)
}
// Struct composed of sendable properties
struct User {
 var id: Int
 var name: String
}
let user = User(id: 42, name: "Blob")
Task {
 print(user)
}
```

The majority of standard library types, including simple value types like booleans, strings, arrays, dictionaries of sendables, easily conform to `Sendable`.

Exclusions and Oversight:
However, some types, like `AttributedString`, might not conform to `Sendable`, leading to warnings:

```
struct User {
 var id: Int
 var name: String
 var bio: AttributedString // ⚠ Not Sendable
}
```

If a type or a property within a type doesn't adhere to `Sendable`, warnings are raised, alerting that the type might not be safe for concurrent use.

Making Reference Types Sendable:
Making reference types like classes conform to `Sendable` is more complex. For instance, marking a class with `Sendable` raises warnings due to mutable properties:

```
class User: Sendable {
 var id: Int // ⚠ Mutable property
 var name: String
}
```

Swift requires classes to be `final` and their properties immutable to be considered `Sendable`:

```
final class User: Sendable {
 let id: Int
 let name: String
}
```

While this ensures safety, it restricts the class's ability to modify internal state, resembling the behavior of value types.

Using `@unchecked Sendable`:

In cases where Swift can't infer safety but you're confident about the concurrent safety of a type, you can use **`@unchecked Sendable`** to bypass compiler warnings:

```
class Counter: @unchecked Sendable {
 let lock = NSLock()
 var count = 0
 func increment() {
 self.lock.lock()
 defer { self.lock.unlock() }
 self.count += 1
 }
}
```

However, using **`@unchecked Sendable`** means operating outside the compiler's safety checks, requiring caution as the compiler won't detect potential unsafe modifications that might compromise concurrent safety.

@Sendable Closures

Absolutely! In Swift, when you work with closures, their behavior can significantly impact concurrent or asynchronous operations. The `@Sendable` attribute is used to signal that a closure is safe to be used in a concurrent context. Let's delve deeper into the concepts and how they affect the code.

Escaping Closures and Asynchronous Operations

When a closure is marked with `@escaping`, it means it can be stored and executed after the enclosing function has finished executing. For instance:

```
func perform(work: @escaping () -> Void) {
 DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
 work()
 }
}
```

Here, `work` is an escaping closure that's executed after a one-second delay on the main queue. Escaping closures are crucial for scenarios like completion handlers in asynchronous operations where the closure might execute at a later time.

Async/Await for Asynchronous Operations

Swift's `async` and `await` keywords enable you to write asynchronous code that looks synchronous. For instance:

```
func perform(work: @escaping () -> Void) async throws {
 try await Task.sleep(nanoseconds: NSEC_PER_SEC)
```

```
 work()
}
```

This function is marked as `async` and utilizes `Task.sleep` to asynchronously delay execution for a specified duration before invoking the `work` closure.

@Sendable and Concurrent Contexts

Now, when we talk about `@Sendable`, it's related to concurrency. It ensures that a closure can be safely used in a concurrent context without causing race conditions or accessing mutable states inappropriately.

```
func perform(work: @escaping @Sendable () -> Void) {
 Task {
 // Using the Sendable-marked closure in a concurrent context
 }
}
```

By applying `@Sendable` to a closure, it restricts the closure's behavior to prevent capturing mutable states or creating concurrency issues when used concurrently.

Closure Safety and Concurrent Operations

Consider a scenario where you have a `DatabaseClient` struct with async operations:

```
struct DatabaseClient {
 var fetchUsers: () async throws -> [User]
```

```
 var createUser: (User) async throws -> Void
}
```

When using this in concurrent tasks, ensuring that closures provided by `DatabaseClient` are `@Sendable` prevents potential issues:

```
func perform(client: DatabaseClient, work: @escaping @Sendable () -> Void) {
 Task {
 _ = try await client.fetchUsers()
 // …
 }
 Task {
 _ = try await client.fetchUsers()
 // …
 }
 // …
}
```

Applying `@Sendable` to the closure ensures that it won't accidentally modify shared mutable states or cause race conditions when used concurrently.

Actors

Actors for Concurrency Safety

Actors in Swift guarantee data safety across threads without explicit locking. Actors handle synchronization internally, requiring asynchronous invocation:

```
actor CounterActor {
 var count = 0
```

```
 var maximum = 0
func increment() {
 self.count += 1
 self.maximum = max(self.count, self.maximum)
 }
func decrement() {
 self.count -= 1
 }
}
```

Async Invocation with Actors

Invoking actor methods must occur in an asynchronous context to ensure synchronization:

```
let counter = CounterActor()
for _ in 0..<workCount {
 Task {
 await counter.increment() // Invoking actor methods asynchronously
 }
}
```

Actor Isolation and Property Access

Accessing actor properties or methods from outside must be asynchronous due to actor isolation:

```
Task {
 await print("Counter count:", counter.count) // Accessing actor property asynch
 await print("Counter maximum:", counter.maximum)
}
```

Non-determinism and Data Safety

Non-determinism might occur due to concurrent tasks, but it's distinct from data races. Actors ensure data safety while allowing non-deterministic behavior based on scheduling:

```
// Non-deterministic behavior due to concurrent tasks
print("Counter count:", counter.count)
print("Counter maximum:", counter.maximum)
```

Summary:

Swift introduces powerful tools to manage data synchronization and prevent data races in concurrent programming. It addresses issues of mutable state access across threads, offering solutions like actors and the `@Sendable` attribute. This article covers the challenges of data races, Swift's `Sendable` protocol for concurrent safety, managing race conditions with closures, and the role of actors in synchronizing data access. By exploring these features and best practices, Swift developers can create more robust and safer concurrent code.

Series Navigation

- **Part 1:** Exploring Threads in Swift
- **Part 2:** Exploring Swift's Operation Queues: Enhancing Threading with Limitations

- **Part 3:** Mastering Dispatch Queues in Swift: Understanding, Implementation, and Limitations
- **Part 4:** Mastering Concurrency: Task
- **Part 5:** Swift Concurrency: Safeguarding Data with @Sendable and Actors

IOS | IOS App Development | Swift | Swiftui | Swift Programming

Written by VINODH KUMAR

Follow

50 Followers

📱 Senior iOS Developer | Swift Enthusiast | Tech Blogger 🖥️

More from VINODH KUMAR

VINODH KUMAR

Understanding Memory Allocation in Swift: Stack vs. Heap

Memory management is a fundamental aspect of programming, critical for efficient...

5 min read · Apr 17, 2024

28 2

VINODH KUMAR

Mastering Concurrency: Task

Swift 5.5 brought in powerful concurrency tools that simplify and strengthen our...

6 min read · Feb 20, 2024

105

VINODH KUMAR

Exploring Swift's Operation Queues: Enhancing Threading wit...

Apple has introduced abstractions over threads to address the issues we discussed...

4 min read · Feb 20, 2024

3

VINODH KUMAR

Automated Text Recognition with Vision Framework in Swift

Text recognition has become an essential part of modern applications, enabling them to...

4 min read · Jul 18, 2023

54

See all from VINODH KUMAR

Recommended from Medium

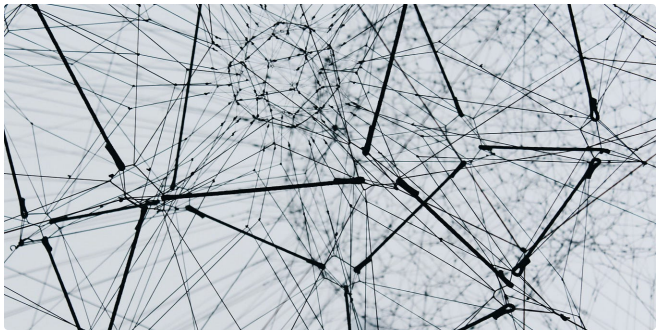
Vladyslav Shkodych

Mastering SwiftUI: Are You Really as Good as You Think?

How deep do you think you know what SwiftUI is and how it works? Have you ever wondere...

16 min read · May 10, 2024

413 6



Kevin Abram

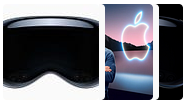
iOS app modularization: the conceptual understanding

Modularization is the process of separating the functionality of a program into separate...

8 min read · Apr 9, 2024

80 1

Lists

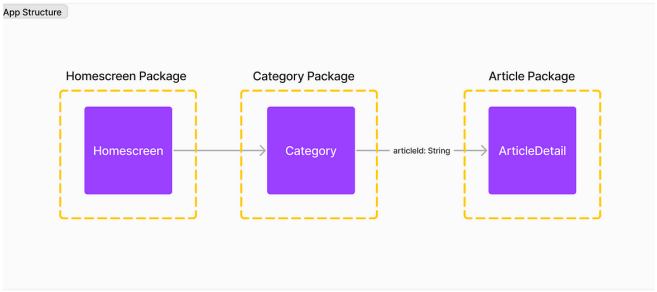


Apple's Vision Pro
7 stories · 68 saves

Tech & Tools
16 stories · 237 saves

Icon Design
36 stories · 310 saves

Productivity
241 stories · 434 saves



Jan Maloušek

Modular SwiftUI Navigation (Part 1)

With iOS 16 and above, we finally have all the tools needed to implement complex...

6 min read · Feb 25, 2024

61

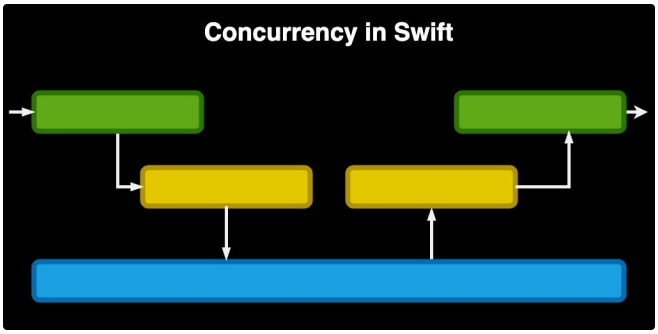
Valentin Jahanmanesh

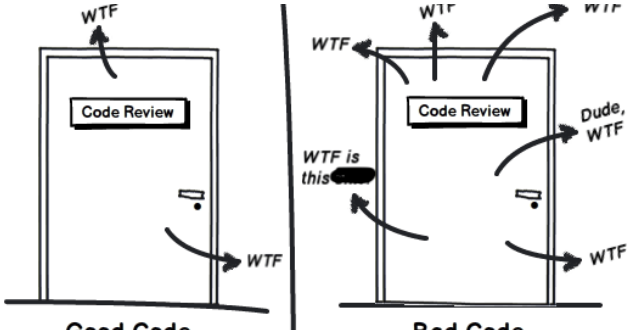
Swift Actors—in depth

Learn, don't memorize.

15 min read · Nov 23, 2023

465 6





Mahdi Saedi

Concurrency in Swift

Swift includes support for concurrency, letting you write code that can handle multiple...

7 min read · May 4, 2024

108

Christopher Saez 📱

The cool Swift features that you should not (ab)use

As an iOS developer, I used to work on a lot of legacy code, or I tried some Swift features...

✦ · 5 min read · Apr 22, 2024

163 4

See more recommendations