Swift 6 · strict concurrency & data-race safety

In one line: In the **Swift 6 language mode** data-race safety is a **compile-time guarantee**: all mutable state belongs to one **isolation domain** (an actor, a global actor like `@MainActor`, or `nonisolated`); a value crosses domains only if it is **`Sendable`** or provably **never used again** by the sender (region isolation / `sending`). Swift 5 warnings become **errors**.

How it works

- **Domains.** Each `actor` instance = one domain (serial executor). A **global actor** (`@MainActor`, or your own `@globalActor`) = one domain shared by everything marked with it. **`nonisolated`** = no domain: sees only `let`/Sendable state. Crossing = `await`.
- **`Sendable`** = marker protocol, safe to share. Implicit for **non-public** structs/enums with Sendable members; **`public`** ones must declare it (unless `@frozen`). A `final class` with only `let` Sendable stored properties may declare it (checked). Actors and `@MainActor` classes are Sendable for free. `@unchecked Sendable` = "trust me, I lock".
- **`@Sendable` closure** may run concurrently: Sendable captures only, no captured `var`. `Task {}`, `Task.detached`, `addTask` take a **`sending`** closure instead (6.0): it may capture what the caller gives up.
- **Region-based isolation** (SE-0414, Swift 6.0): a *non*-Sendable value in a **disconnected region** (fresh, not reachable from any actor's state) may be sent if the sender never uses it again. **`sending`** (SE-0430) spells it in a signature: `f(_ x: sending Box)`, `-> sending T`.
- **`@preconcurrency import M`** turns Sendable errors (Swift 6 mode) into warnings for types of a not-yet-migrated module.
- **Swift 6.2, opt-in per module** (not implied by Swift 6 mode): default isolation `MainActor` (SE-0466); with Xcode's **Approachable Concurrency** setting, nonisolated `async` runs on the caller's actor (SE-0461), `@concurrent` opts out; isolated conformance `extension V: @MainActor P` (SE-0470).

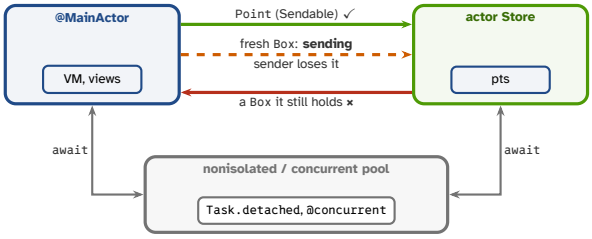
Example

```
final class Box { var n = 0 }           // not Sendable
struct Point: Sendable { var x, y: Int } // value: fine
actor Store { var pts: [Point] = []
  func add(_ p: Point) { pts.append(p) }
  func take(_ b: sending Box) { b.n += 1 } }
@MainActor final class VM {             // UI domain
  let store = Store()
  func run() async {
    await store.add(Point(x: 1, y: 2))  // Sendable crosses
    let b = Box()                        // disconnected region
    await store.take(b)                  // OK: sent away…
    // b.n = 5   error: 'b' used after being sent
  }
}
```

Migration strategy

1. Swift 5 mode + **complete** checking *as warnings*: `SWIFT_STRICT_CONCURRENCY = complete` (SwiftPM `.enableUpcomingFeature("StrictConcurrency")`).
2. **One module at a time**: UI types `@MainActor`, models as Sendable values, deps behind `@preconcurrency import`.
3. Flip it: `SWIFT_VERSION = 6` / `swiftLanguageModes: [.v6]`; modules in 5 and 6 mode link together (`swift-tools-version: 6.0` alone flips *every* package target to 6). Then shrink every `@unchecked` / `(unsafe)` / `@preconcurrency` — each is debt.

Domains and what may cross



*Inside a domain: synchronous. Across: `await` + the value is **Sendable** or a **disconnected region**.*

Common errors → fix

Compiler says (abridged)	Fix, best first
Capture of 'x' with non-sendable type 'X' in a '@Sendable' closure	make `X` a Sendable value; capture a copy `[v = x.v]`; isolate both to one actor
Sending 'x' risks causing data races	don't touch `x` after the send; make it Sendable; param `sending`
Main actor-isolated … can not be referenced from a nonisolated context	mark the caller `@MainActor`; or make it `async` + `await`
Static property 'shared' is not concurrency-safe because it is nonisolated global shared mutable state	`static let` + Sendable type; `@MainActor static var`; last resort `nonisolated(unsafe)`
Stored property 'n' of 'Sendable'-conforming class 'C' is mutable	`let`; make it an `actor`; or `Mutex`/lock + `@unchecked Sendable`
Main actor-isolated method cannot satisfy nonisolated protocol requirement	`nonisolated` + `MainActor.assumeIsolated`; `@MainActor` protocol; 6.2 isolated conformance

Traps

- ! **Swift 6 compiler ≠ Swift 6 mode**: an old project on the 6.x compiler still builds in Swift 5 mode; mode is per module.
- ! `@unchecked Sendable` over an unlocked `var` mutes the checker and *keeps* the race.
- ! `Task.detached` in an actor is **not** isolated to it; `Task {}` is only if it strongly captures `self` (SE-0420).
- ! Actors stop *data* races, not *logic* races — state can change across any `await` (reentrancy).
- ! Runtime too: in Swift 6 mode, isolated code reached off its actor from unchecked code (`@objc`, `@preconcurrency` conformances) **crashes instead of racing** (SE-0423).

Remember

Every value has one home. It may travel if it is the 3 S: **S**endable, **S**ent (disconnected, never touched again), or **S**ynchronised by you (`@unchecked`, at your own risk).

Related: concurrency (async/await, structured concurrency) · actors & reentrancy (`@MainActor` UI confinement) · AsyncSequence & AsyncStream (continuations) · locks & synchronisation primitives (`Mutex`, iOS 18)