

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 `@globalActor actor DB { static let shared = DB() }`) = 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. A `final class` with only `let Sendable` stored properties conforms (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. (`addTask`, `Task.detached`.)
- **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`.
- **Globals** (SE-0412): a global/static var must be global-actor isolated, or a `Sendable let`. Hatch: `nonisolated(unsafe)`.
- **@preconcurrency import M** silences `Sendable` errors for types of a not-yet-migrated module.
- **Swift 6.2 / Xcode 26 “Approachable Concurrency”**: default isolation `MainActor` (SE-0466); `nonisolated async` runs on the caller’s actor (SE-0461), `@concurrent` opts out; isolated conformance extension `V: @MainActor P` (SE-0470).

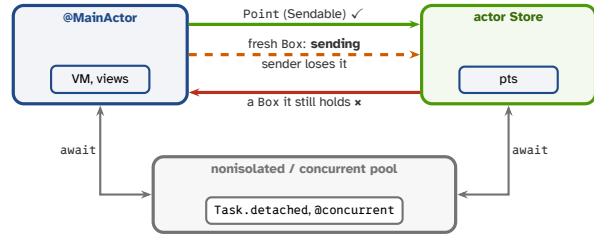
Example

```
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
  }
  nonisolated func id() -> String { "vm" } // no hop
}
```

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. Then shrink every `@unchecked` / `(unsafe)` / `@preconcurrency` — each is debt.

Picture — domains and what may cross



Inside a domain: **synchronous**. Across: **await** + the value is **Sendable** or a **disconnected region**. **Non-Sendable** state still reachable from its home (red) never leaves.

Common errors → fix

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

Interview 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 {}` inherits the actor.
- ! Actors stop *data* races, not *logic* races — state can change across any `await` (reentrancy).
- ! A `public struct` is **not** implicitly `Sendable`.

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).

Likely questions

1. Data race vs race condition? — concurrent unsynchronised access with a write vs bad ordering; Swift 6 rules out only the first.
2. sending vs `Sendable`? — one value handed over once vs a type always safe.
3. Why is an actor `Sendable`? — every access to its state is serialised.
4. `@preconcurrency import`? — mutes `Sendable` errors for that module’s types.
5. How can `Task {}` capture a non-`Sendable` local? — its closure is `sending`: legal if you never touch the local afterwards.