

In one line: Swift errors are **values** (any type conforming to the empty protocol `Error`) returned on a **separate, checked path**: a function says `throws`, every call site says `try`, and the error travels up until a `do/catch` handles it. No stack unwinding across unmarked frames, no cost on the success path, no stack trace.

How it works

- try** propagates (caller must throw or catch); **try?** turns the error into `nil` and *flattens* (`throws` → `Int?` gives `Int?`, since Swift 5); **try!** traps on error.
- catch clauses** are patterns, checked **top to bottom**: `catch E.timeout`, `catch E.status(let c) where c >= 500`, `catch let e as DecodingError`, `catch is CancellationError`; a bare `catch` binds error. In a non-throwing function the catches must be **exhaustive** (usually: end with bare `catch`).
- rethrows**: throws only if its closure argument does — `map`, `filter` need `try` only for a throwing closure.
- Result<Success, Failure: Error>** = the outcome as a value: store it, pass it to a callback, collect many. `Result { try f() } ↔ try r.get(); map, mapError, flatMap`.
- Typed throws** (Swift 6, SE-0413): `throws(ParseError)`. `throws` ≡ `throws(any Error)`; `non-throwing` ≡ `throws(Never)`. A `do` whose calls all throw `E` gives `catch` an error: `E` — exhaustive switch, no casting. `Generic throws(E)` replaces `rethrows`. Advice: **keep public APIs untyped** (adding a case later breaks clients); typed for closed internal domains/Embedded.
- defer** runs on every scope exit, **LIFO**; after a return expression is evaluated; it cannot throw or return itself.
- LocalizedError**: `errorDescription`, `failureReason`, `recoverySuggestion` — feeds `localizedDescription`. `CustomNSError`: `domain/code` for `NSError` bridging.

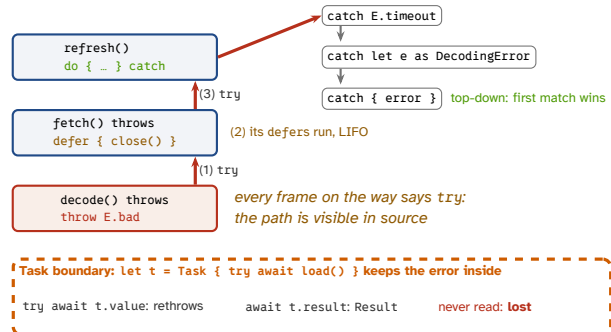
Example

```
enum NetErr: Error { case timeout, status(Int) }
func refresh() async { // non-throwing
    isLoading = true
    defer { isLoading = false } // on every exit
    do {
        let data = try await api.fetch() // throws NetErr
        items = try JSONDecoder().decode([Item].self, from: data)
    } catch NetErr.status(let c) where c >= 500 {
        banner = "Server down (\(c))"
    } catch is CancellationError {
        return // user left: no banner
    } catch {
        // required catch-all
        log(error); banner = error.localizedDescription
    }
}
```

Never-crash rules

- No `try!` / `!` on input you do not own (network, disk, user). `try!` only for invariants: a bundled fixture, a literal regex.
- `assert` = debug only (gone in -O); precondition stays in release; `fatalError` always. All three are for *programmer* errors, not runtime conditions.
- Wrap with context, never swallow: `catch { throw AppError.parse(underlying: error) }`.

Picture — how an error travels



Across async & Task

- `async throws` composes: call with `try await`. Order of keywords: `try await f()`.
- `Unstructured Task{}` stores the error in `Task<T, any Error>`; nothing forces you to read it.
- `async let`: error surfaces at the `await`; siblings are cancelled when the scope exits. `withThrowingTaskGroup`: first thrown child error leaves the group → the rest are cancelled.
- Cancellation** is cooperative: `try Task.checkCancellation()` and `Task.sleep throw CancellationError`; `URLSession` throws `URLError(.cancelled)` instead — treat both as "not a failure".

Interview traps

- `try?` on `throws` → `Int?`: `nil` means "threw" or "returned nil" — use `do/catch`.
- In `do { try a(); try b() }`, if `a` throws, `b` **never** runs.
- A `defer` that mutates the returned variable does **not** change the already-evaluated return value.
- Typed `throws` in a public API = a breaking change for every new case.

Remember

"try marks, throw sends, catch ends; defer cleans backwards." `try?` = forget, `try!` = bet the app, `Result` = keep for later.

Likely questions

- Result vs throws? — throws for inline flow; Result to store/pass/collect outcomes.
- What is `rethrows`? — throws only when the passed closure throws.
- `throws` is sugar for? — `throws(any Error)`.
- Two `defers` — order? — reverse of declaration (LIFO).
- Error in a `Task` nobody awaits? — lost; read `.value/.result`.