Swift Cheat Sheet

Swift 6 language mode · Swift 6.3 (6.3.2) · macOS & Linux · verified against TSPL
Irurueta Docs — Guides & References → Programming Languages → Swift Reference

BASICS & TYPES

```
let pi = 3.14159        // constant, cannot be reassigned
var count = 0                   // inferred Int
var name: String = "Ada"     // explicit annotation
let big = 1_000_000, hex = 0xFF, bin = 0b1010
let d = Double(count)   // NO implicit numeric conversion
let p = (code: 404, msg: "NF"); p.code   // tuple
```

Int/UInt	Word-sized; Int8…Int64 fixed widths
Double/Float	64/32-bit; Double is the default
Bool	true/false only, never 0/1
String/Character	Unicode-correct value types
Any/AnyObject	Any type / any class instance
Never	Uninhabited: never returns

OPTIONALS

```
var s: String? = "hi"   // Optional<String>; nil if unset
if let s { print(s) }    // shorthand binding, same name
if let v = s, !v.isEmpty { }    // bind + extra condition
guard let s else { return } // early exit; s bound after it
let n = s ?? "anon"           // nil-coalescing (chains)
let len = s?.count           // optional chaining -> Int?
a?.b?.run()                    // nil if ANY link is nil
let f = s!                     // TRAPS if nil — avoid
```

try?, as?, dictionary subscripts and failable inits all produce optionals.

Never use ! to silence a compiler error — bind it.

STRINGS & CHARACTERS

```
let i = "\(a) has \(a.count) chars"   // interpolation
let m = """
    line one       // closing delimiter sets indentation
    """
let raw = #"no \(escape) here"#    // extended delimiters
let idx = a.index(a.startIndex, offsetBy: 1)
a[idx]; a[a.startIndex..<idx]  // String.Index, never Int
```

.count	Grapheme clusters, not bytes (O(n))
.isEmpty	Prefer over count == 0
.hasPrefix/Suffix	Literal prefix/suffix test
.split(separator:)	Yields [Substring]
.lowercased()	Also uppercased, trimmingPrefix

Substring shares the parent's storage — wrap in String(…) to store.

ARRAY / SET / DICTIONARY

```
var xs = [1, 2, 3]; var e: [String] = []
var z = [Int](repeating: 0, count: 5)
xs.append(4); xs += [5]; xs.insert(0, at: 0)
xs.remove(at: 0); xs.removeLast(); xs.first; xs.last
xs[1...2]; xs.count; xs.isEmpty; xs.contains(3)

var st: Set<Int> = [1, 2, 2] // {1,2} unordered, Hashable
st.insert(9); st.union(o); st.intersection(o)

var ages = ["ada": 36]            // [String: Int]
ages["ada"]                  // Int? — missing key is nil
ages["eve", default: 0] += 1        // upsert idiom
ages.removeValue(forKey: "ada")
for (k, v) in ages { }          // order NOT guaranteed
ages.keys; ages.values; ages.mapValues { $0 + 1 }
```

All three are value types with copy-on-write: assignment copies, storage is shared until mutated.

8.2 InlineArray<3, Int> (fixed-size, inline) and Span/RawSpan (safe non-owning views).

HIGHER-ORDER FUNCTIONS

```
xs.map { $0 * 2 }                  // transform each
xs.filter { $0.isMultiple(of: 2) }
xs.reduce(0, +)                    // fold to one value
xs.reduce(into: [:]) { $0[$1] = $1 }
ns.compactMap { Int($0) }          // map + drop nils
nested.flatMap { $0 }          // concatenate sub-sequences
xs.sorted(); xs.sorted { $0.age > $1.age }
xs.first { $0 > 2 }                // Element?, stops early
xs.contains { $0 < 0 }; xs.allSatisfy { $0 > 0 }
xs.prefix(3); xs.dropFirst(); xs.reversed();
xs.shuffled()
xs.enumerated()                    // (offset, element)
zip(xs, ys); stride(from: 0, to: 10, by: 2)
Dictionary(grouping: people, by: \.city)
xs.lazy.map(f).filter(g)         // no intermediate arrays
```

CONTROL FLOW & PATTERNS

```
for i in 1...5 { }              // closed; 1..<5 half-open
for x in xs where x > 0 { }     // filtered loop
while c < 10 { c += 1 }; repeat { } while c > 0
func work() { defer { file.close() } } // runs on EVERY exit
-----------------------------------------------------------
let label = if n > 0 { "pos" } else { "non" } // if-expr
let kind = switch code {          // switch-expression
case 200..<300: "ok"              // range pattern
case 400, 404: "client"           // compound case
case let c where c >= 500: "server \(c)"
default: "other"                  // must be exhaustive
}
switch pt {
case (let x, 0), (0, let x): …    // tuple + value binding
case (_, let y) where y < 0: …    // wildcard + guard
}
if case let .success(v) = result { }
for case let .some(x) in opts { }
```

fallthrough enters the next case; break exits a case or loop.

FUNCTIONS

```
func greet(person name: String) -> String {
    "Hi, \(name)"   // implicit return, single expression
}
greet(person: "Ada")    // external label / internal name
func add(_ a: Int, _ b: Int) -> Int { a + b } // no label
func log(_ m: String, level: Level = .info) { } // default
func sum(_ ns: Int...) -> Int { ns.reduce(0, +) } // variadic
func bump(_ n: inout Int) { n += 1 }
bump(&v)            // inout = copy-in/copy-out, not a ref
let op: (Int, Int) -> Int = add    // function type value
func counter() -> () -> Int          // returns a closure
```

Overloads differ by labels, parameter types or return type; argument order is fixed.

CLOSURES

```
let full = { (a: Int, b: Int) -> Int in a + b }
xs.sorted(by: { a, b in a < b })   // types inferred
xs.sorted { a, b in a < b }        // trailing closure
xs.sorted { $0 < $1 }              // shorthand arg names
load { ok() } onFail: { fail() } // multi trailing
func start(_ done: @escaping () -> Void) // outlives the call
queue.async { [weak self] in          // capture list
    guard let self else { return }
    self.finish()
}
let snap = { [n] in print(n) } // captures BY VALUE
lazy var total: Int = { heavy() }() // immediately applied
```

Parameters are non-escaping by default; @autoclosure wraps the argument expression in a closure.

ENUMERATIONS

```
enum Direction { case north, south, east, west }
let d: Direction = .north        // inferred from context

enum Status: Int { case ok = 200, notFound = 404 }
Status(rawValue: 404)          // failable init -> Status?
Status.ok.rawValue                  // 200

enum Barcode {                      // associated values
    case upc(Int, Int), qr(String)
}
switch code {
case .upc(let sys, let id): …
case .qr(let s): …
}

enum Planet: String, CaseIterable {  // .allCases
    case earth, mars            // raw value == case name
    var isHome: Bool { self == .earth }  // computed prop
}
indirect enum Expr { case lit(Int), add(Expr, Expr) }
```

Value types: they take protocols, methods and static members — but no stored instance properties.

STRUCT VS CLASS

Struct / Enum	Class
Value type — copied	Reference type — shared
No inheritance	Single inheritance
Free memberwise init	Must write init
mutating to change self	Mutates freely
Inline, no ARC	Heap + ARC retain/release
let freezes contents	let freezes only the ref
No deinit	deinit on last release
== via Equatable	=== identity too
Sendable if members are	Needs final+immutable, or actor

```
struct Point { var x = 0.0, y = 0.0
    mutating func moveBy(dx: Double) { x += dx } }
let p = Point(x: 1, y: 2)      // free memberwise init
```

Default to struct; use class only for identity, inheritance or deinit.

PROPERTIES

```
struct Rect {
    var w = 0.0, h = 0.0              // stored
    var area: Double { w * h }      // read-only computed
    var diag: Double {
        get { (w*w + h*h).squareRoot() }
        set { w = newValue }       // newValue implicit
    }
}
class View {
    lazy var renderer = Renderer() // built on first access
    var title = "" {
        willSet { … }                  // newValue
        didSet { if title != oldValue { redraw() } }
    }
}
@propertyWrapper struct Clamped {
    private var v = 0
    var wrappedValue: Int {
        get { v }
        set { v = min(max(newValue, 0), 100) } } }
struct Level { @Clamped var percent: Int }
```

lazy requires var and is not thread-safe; observers never fire during init.

INITIALIZATION & DEINIT

```
class Vehicle {
    let wheels: Int; var name: String
    init(wheels: Int, name: String) {   // designated
        self.wheels = wheels // phase 1: stored
        self.name = name
    }
    convenience init(name: String) {  // delegates ACROSS
        self.init(wheels: 4, name: name)
    }
    init?(spec: String) {              // failable
        guard let n = Int(spec) else { return nil }
        wheels = n; name = "c" }
    deinit { … }                       // classes only
}
class Car: Vehicle {
    let doors: Int
    init(doors: Int) {
        self.doors = doors             // own props FIRST…
        super.init(wheels: 4, name: "car") // …then super
    }
}
```

Designated inits delegate *up*, convenience inits *across*. A struct keeps its free memberwise init if your own inits live in an extension.

PROTOCOLS & EXTENSIONS

```
protocol Drawable: Equatable {
    var area: Double { get }          // or { get set }
    associatedtype Unit: Numeric  // placeholder type
    init(area: Double)
    func draw() -> String
}
extension Drawable {            // default implementation
    func draw() -> String { "area \(area)" }
}
extension Drawable where Unit == Int { }  // constrained
struct Circle: Drawable { }
extension Circle: Codable { } // conformance in extension
extension Int {              // extend ANY type
    var squared: Int { self * self }
}
```

Extensions add methods, computed properties, inits, subscripts, nested types and conformances — never stored properties or overrides.

GENERICS, SOME / ANY

```
func largest<T: Comparable>(_ xs: [T]) -> T? { xs.max() }
struct Stack<Element> {
    private var items: [Element] = []
    mutating func push(_ e: Element) { items.append(e) }
}
extension Stack where Element: Equatable { }
func pair<A, B>(_ a: A, _ b: B) -> (A, B)
    where A: Hashable, B: Sendable { (a, b) }
```

some P **Opaque**: one concrete type hidden from the caller. No boxing, keeps identity.

any P **Existential**: any conforming type, chosen at run time. Boxed; needed for heterogeneous arrays.

```
func make() -> some Drawable { Circle() }
let shapes: [any Drawable] = [Circle(), Square()]
func render(_ s: some Drawable) { }   // == generic sugar
```

6.2 Integer generic parameters: InlineArray<3, Int> parameterises on a literal, not a type.

ERROR HANDLING

```
enum ParseError: Error { case empty, badToken(String) }

func parse(_ s: String) throws -> [String] {
    guard !s.isEmpty else { throw ParseError.empty }
    return s.split(separator: " ").map(String.init)
}
do {
    let parts = try parse(text)
} catch ParseError.empty { …
} catch let e as ParseError { …
} catch { print(error) }      // implicit `error` binding

let maybe = try? parse(t)   // -> [String]?, nil on throw
let forced = try! parse(t)     // TRAPS if it throws
func strict(_ s: String) throws(ParseError) // typed throws

let r: Result<[String], ParseError> = Result { try parse(t) }
switch r { case .success(let v): …; case .failure(let e): … }
let v = try r.get()
```

throws ≡ throws(any Error); throws(Never) cannot throw. Traps (fatalError, precondition) are for impossible states, not recoverable failure.

ARC & MEMORY

```
class Node {
    var children: [Node] = [] // parent owns kids
    weak var parent: Node?       // weak breaks the cycle
}
class Card { unowned let owner: Person } // shorter life
```

strong	Default; keeps the referent alive.
weak	Optional var, auto-niled on dealloc. For a peer that may vanish first.
unowned	Non-optional, **not** zeroed — traps if used after dealloc. Strictly shorter lifetime only.

```
queue.async { [weak self] in
    guard let self else { return } // cycle fix
    self.update() }
```

ARC applies to classes, actors and closures only — structs and enums are copied, never retained.

A cycle leaks silently: no crash, just memory that never frees.

ACCESS LEVELS

Level	Visible to
open	Other modules; subclassable/overridable outside. Classes & members only.
public	Other modules; **not** subclassable outside.
package	Every module in the same SwiftPM package.
internal	**Default.** The declaring module.
fileprivate	The declaring source file.
private	The enclosing declaration + its same-file extensions.

```
public struct Config {
    public private(set) var version: Int // public get
    public init() { version = 1 } // required
}
```

Nothing may be more visible than its own types: a public function cannot take an internal parameter.

CONCURRENCY: ASYNC / AWAIT

```
func fetch(_ u: URL) async throws -> Data {
    try await URLSession.shared.data(from: u).0
}
let d = try await fetch(url)  // await = suspension point

async let a = fetch(u1)          // start concurrently…
async let b = fetch(u2)
let both = try await (a, b)     // …join here

Task { await refresh() } // unstructured
let t = Task { try await fetch(u) }
let v = try await t.value; t.cancel()

try await withThrowingTaskGroup(of: Data.self) { g in
    for u in urls { g.addTask { try await fetch(u) } }
    for try await d in g { collect(d) } } // bounded lifetime
}
try Task.checkCancellation()  // cooperative
if Task.isCancelled { return }
for try await l in handle.bytes.lines { } // AsyncSequence
```

6.4-beta withTaskCancellationShield is still in development — don't rely on it.

ACTORS, ISOLATION & SENDABLE

```
actor Counter {
    private var n = 0              // isolated state
    func bump() { n += 1 }       // sync INSIDE the actor
    nonisolated let id = UUID() // immutable, no isolation
}
await c.bump()                     // await from OUTSIDE

@MainActor final class ViewModel { // all main-isolated
    var rows: [Row] = []
    nonisolated func pureHelper() { }
}
@MainActor func render() { }
await MainActor.run { label.text = "done" }

struct Row: Sendable { let id: Int } // crosses isolation
@unchecked Sendable    // escape hatch
```

Value types with Sendable members, final immutable classes and all actors are implicitly Sendable. @MainActor is the built-in global actor; declare others with @globalActor.

6.2 SE-0466 default actor isolation: a target can make @MainActor the default for every declaration.

Swift 6 mode makes data-race violations errors, not warnings.

REGEX & KEY PATHS

```
let re = /(\d{4})-(\d{2})-(\d{2})/ // typed captures
let re2 = try Regex(pattern) // -> AnyRegexOutput
if let m = "2026-09-13".firstMatch(of: re) {
    let (all, y, mo, d) = m.output // strongly typed tuple
}
text.contains(re); text.matches(of: re)
text.replacing(re, with: "[d]"); text.wholeMatch(of: re)
text.split(separator: /\s*,\s*/)
-----------------------------------------------------------
let kp = \Person.address.city  // KeyPath<Person, String>
person[keyPath: kp]                  // read
let w = \Person.name              // WritableKeyPath (var)
person[keyPath: w] = "Ada"
people.map(\.name)                // key path as a function
people.sorted(using: KeyPathComparator(\.age))
@dynamicMemberLookup struct Proxy { } // o.x -> subscript
```

MACROS

```
#warning("remove before release")   // freestanding macro
let p = #Predicate<Person> { $0.age > 18 }
@Observable final class Model { }       // attached macro
@freestanding(expression)
macro stringify<T>(_ v: T) -> (T, String) =
    #externalMacro(module: "MyMacros", type: "StringifyMacro")
@attached(member, names: named(_storage))
macro AddStorage() = #externalMacro(…)
```

Roles: @freestanding(expression|declaration); @attached(member|accessor|peer|extension|memberAttribute). Macros run on the syntax tree, are type-checked, and are hygienic.

ATTRIBUTES

@available	Platform gate; deprecated, renamed:, unavailable
@main	Program entry-point type
@discardableResult	Return value may be ignored
@escaping / @autoclosure	Closure outlives call / wrap argument
@inlinable	Expose body for cross-module inlining
@usableFromInline	Internal symbol used by @inlinable
@frozen	Layout will never change (ABI)
@backDeployed	Run on OS versions older than shipped
@objc / @objcMembers	Expose to the Objective-C runtime
@unknown default	Future-proof a switch over a library enum

```
#if os(Linux) … #elseif canImport(Darwin) … #endif
#available(iOS 17, *); #file, #line, #function
```

PACKAGE.SWIFT

```
// swift-tools-version: 6.1
import PackageDescription
let package = Package(
  name: "WeatherKit",
  products: [
    .library(name: "WeatherKit", targets: ["WeatherKit"]),
    .executable(name: "weather-cli",
                targets: ["WeatherCLI"])],
  dependencies: [
    .package(url: "…/swift-argument-parser.git",
             from: "1.5.0"),
  targets: [
    .target(name: "WeatherKit",
            dependencies: ["SharedModels"],
            swiftSettings: [.swiftLanguageMode(.v6)]),
    .executableTarget(name: "WeatherCLI", dependencies: [
        .product(name: "ArgumentParser",
                 package: "swift-argument-parser")]),
    .testTarget(name: "WeatherKitTests",
                dependencies: ["WeatherKit"])])
```

COMMAND LINE

```
swift build [-c release]        # build every target
swift run weather-cli --city Madrid
swift test [--filter Tests/name]
swift package resolve | update # Package.resolved
swift format lint --recursive Sources/

swiftc -swift-version 6 main.swift -o app
swiftc -O main.swift -o app     # -Osize for binary size
swiftc -strict-concurrency=complete main.swift
swiftc -enable-upcoming-feature ExistentialAny main.swift

swiftly install 6.3.2 && swiftly use 6.3.2   # toolchains
swift build --swift-sdk <triple> # cross-compile
```

SWIFT TESTING

```
import Testing          // modern default
@testable import WeatherKit
@Test func parsesCelsius() {
    #expect(r.celsius == 21.5) // keeps going
}
let c = try #require(City(name: "M"))  // unwrap or stop
@Suite("Reading parsing") struct ReadingTests {
    @Test("formats", arguments: [("21.5C", 21.5), ("70F", 21.1)])
    func parses(raw: String, exp: Double) { } // parameterized
}
@Test(.tags(.networking), .timeLimit(.seconds(5)))
@Suite(.serialized) // .enabled(if:) / .disabled("reason")
```

XCTEST

```
import XCTest              // still required for UI tests
final class T: XCTestCase {
  func testParses() throws {
    XCTAssertEqual(r.c, 21.5, accuracy: 0.01)
    XCTAssertTrue(f); XCTAssertNil(x)
    XCTAssertThrowsError(try parse(""))
  } }
```

Both run under swift test and can coexist in one target during a migration.