Swift Compiled language and SwiftPM

CONCEPTS

Swift Package Manager (swift package, build, run, test) is built in; Package.swift is the manifest.

Package.swift manifest
↓ `swift package resolve`
Sources/ targets
↓ `swift build`
.build/ binaries
↓ `swift run`
Output your program

Prefer let over var; the compiler warns when a var is never mutated.

TOOLCHAIN

`swift --version`
Show Swift version

`swift <file>.swift`
Run a script directly

`swiftc <file>.swift -o <out>`
Compile to an executable

`swift run <target> <args>`
Run a named executable with args

`swift build -c release`
Optimized release build

PACKAGE MANAGER

`swift package init --name <name> --type executable`
Create an executable package

`swift package init --type library`
Create a library package

`swift package resolve`
Resolve and fetch dependencies

`swift package update`
Update to newest allowed versions

`.package(url: "<url>", from: "1.0.0")`
Declare a dependency

BASICS

`let x = 1`
Constant

BASICS (cont.)

`var x = 1`
Variable

`let x: Double = 1`
Explicit type

`"Hi \(name)"`
String interpolation

`let r = 1...5`
Closed range

`for i in 0..<n { }`
Loop over a range

`if <cond> { } else { }`
Conditional

`switch x { case 1: ... default: ... }`
Pattern match, must be exhaustive

`func f(_ a: Int, b: Int = 0) -> Int { }`
Function with labels

`f(1, b: 2)`
Call with argument label

`print(x)`
Print a value

OPTIONALS

`var x: Int? = nil`
Optional value

`if let v = x { }`
Unwrap if present

`guard let v = x else { return }`
Early exit unwrap

`x ?? <default>`
Fallback value

`x?.count`
Optional chaining

COLLECTIONS

`let a = [1, 2, 3]`
Array literal

`a.append(4)`
Add an element

`a[0]`
Subscript

`a.count / a.isEmpty`
Size checks

COLLECTIONS (cont.)

`let d = ["k": 1]`
Dictionary literal

`d["k"]`
Lookup returns an optional

`a.map { $0 * 2 }`
Transform elements

`a.filter { $0 > 1 }`
Keep matches

TYPES

`struct P { var x: Int }`
Value type

`class C { var x = 0 }`
Reference type

`enum D { case n, s }`
Enumeration

`protocol Shape { func area() -> Double }`
Protocol

`extension P: Shape { }`
Conform in an extension

`init(x: Int) { self.x = x }`
Initializer

`{ (a: Int) -> Int in a * 2 }`
Closure

ERRORS AND ASYNC

`func f() throws -> Int`
Function that can throw

`do { try f() } catch { }`
Handle an error

`try? f()`
Optional result on failure

`throw MyError.bad`
Throw an error

`enum MyError: Error { case bad }`
Define an error type

`func f() async -> Int`
Async function

`await f()`
Call an async function

`Task { await f() }`
Start async work