Swift

Core syntax, optionals, and collections.

01 Run a Swift program

```
func greet(_ name: String) {
    print("Hello, \(name)!")
}
greet("Maya")
print("Welcome to Swift")
```

Save as hello.swift; run swift hello.swift. Compile: swiftc hello.swift -o hello; run ./hello. Each later snippet is an independent file with top-level code. Foundation imports are shown when needed.

02 Bindings & types

```
let name: String = "Maya"
var count: Int = 3
count += 1
let price: Double = 19.95
let ready: Bool = true
print(name, count, price, ready)
```

Types can be inferred. let prevents reassignment; var permits it. A let class reference can still mutate the object. Struct values bound with let cannot mutate their properties.

03 Operators & conversions

```
let whole = 7 / 2
let ratio = Double(7) / 2
let rest = 7 % 2
let ok = whole > 0 && rest == 1
print(whole, ratio, rest, ok)
```

Integer division truncates. Numeric conversion is explicit. == compares equality; === compares class identity. &&, ||, and ! combine Bool values. + - * / % do arithmetic.

04 Strings & characters

```
let name = "Maya"
let greeting = "Hello, \(name)!"
print(greeting.uppercased())
print(name.contains("ay"))
print(name.count)
for ch in name { print(ch) }
```

\(expression) inserts a value. count counts Characters (extended grapheme clusters). Use String.Index, not integer indices. Triple quotes delimit multiline strings.

05 Optionals & unwrapping

```
let text: String? = "42"
if let text = text, let n = Int(text) {
    print(n + 1)
}
print(text?.count ?? 0)
```

? permits nil; optional binding unwraps. ?. chains access; ?? defaults. Int(text) can fail. Avoid ! unless non-nil is guaranteed.

06 if & switch

```
let score = 82
if score >= 60 { print("Pass") }
switch score {
case 90...100: print("A")
case 60..<90: print("Pass")
default: print("Try again")
}
```

Conditions need Bool. switch must cover every case; default covers the rest. No automatic fallthrough. ... includes both ends; ..< excludes the upper bound.

07 Loops & ranges

```
for n in 1...3 { print(n) }
for n in 0..<3 { print(n) }
for n in stride(from: 6, to: 0, by: -2) {
    print(n)
}
var left = 2
while left > 0 { left -= 1 }
```

for-in visits sequences. stride(to:) excludes the end; stride(through:) includes it. break exits; continue skips ahead. repeat/while runs at least once.

08 Functions & labels

```
func add(_ a: Int, to b: Int = 0) -> Int {
    return a + b
}
print(add(2, to: 3))
print(add(2))
```

_ omits a call label; to is external, b local. Defaults are allowed. -> gives the return type; no result is Void. inout and & enable write-back parameters.

09 Arrays

```
var names = ["Maya", "Leo"]
names.append("Sam")
if let first = names.first { print(first) }
for (i, name) in names.enumerated() {
    print(i, name)
}
```

Arrays store ordered values with value semantics: assignment creates an independent value. first/last are optional. Invalid subscripts trap; check bounds.

Swift

Data models, behavior, and checked I/O.

10 Sets & dictionaries

```
let tags: Set<String> = ["web", "api"]
var ages = ["Maya": 28]
ages["Leo"] = 31
print(tags.contains("web"))
print(ages["Sam", default: 0])
```

Sets hold unique Hashable values; map keys need Hashable. Missing-key lookups return nil. Assign nil to remove a key. Both collections have value semantics.

11 Closures & transformations

```
let nums = [1, 2, 3, 4]
let twice = { (n: Int) in n * 2 }
let even = nums.filter { $0 % 2 == 0 }
let result = even.map(twice)
print(result)
print(nums.reduce(0, +))
```

Closures use { parameters in body }; $0 is argument one. filter selects, map transforms, reduce accumulates. A final closure can sit outside parentheses.

12 Structs & mutating methods

```
struct Counter {
    var value = 0
    mutating func add() { value += 1 }
}
var counter = Counter()
counter.add()
print(counter.value)
```

Structs have value semantics. mutating changes self or stored properties. A memberwise initializer is supplied when applicable. Computed properties use getters/setters.

13 Classes & identity

```
class Counter { var value = 0 }
let first = Counter()
let second = first
second.value += 1
print(first.value, first === second)
```

Classes have reference semantics: both names refer to one object. init initializes; deinit handles final cleanup. Automatic reference counting manages class lifetime.

14 Enums & associated values

```
enum State { case idle, ready(Int) }
let state = State.ready(42)
switch state {
case .idle: print("Waiting")
case .ready(let value): print(value)
}
```

Enums define fixed cases; associated values carry data. Patterns extract them. A switch covering every enum case needs no default.

15 Protocols

```
protocol Named { var name: String { get } }
struct User: Named {
    let name: String
}
let user = User(name: "Maya")
print(user.name)
```

Protocols require properties/methods; : declares conformance. { get } requires readability, not immutability. A generic bound T: Named requires that behavior.

16 Extensions

```
extension String {
    var shouted: String { uppercased() }
}
let name = "Maya"
print(name.shouted)
```

Extensions add methods, computed properties, or conformance, but no stored instance properties. This property returns new text without changing name.

17 Throw & catch errors

```
enum InputError: Error { case invalid }
do {
    let value = -1
    if value < 0 {throw InputError.invalid}
    print(value)
} catch { print("Error: \(error)") }
```

Declare throws; call with try. do/catch handles errors. try? converts failure to nil; try! traps. defer runs cleanup when the current scope exits.

18 Read a text file

```
import Foundation
do {
    let text = try String(
        contentsOfFile: "notes.txt",
        encoding: .utf8)
    print(text)
} catch { print("Read: \(error)") }
```

Provide notes.txt or handle read/decode failure. This loads the whole file. Foundation adds filesystem services. Use URL-based APIs for URLs.