Comprehensive Swift Programming Guide

Swift Language • SwiftUI • Combine • Networking
iOS Development • Data Structures & Algorithms

A complete guide with 120+ topics, code examples, and practical implementations

Generated on: September 13, 2025

Table of Contents

PART I: SWIFT LANGUAGE FUNDAMENTALS

Chapter 1: Swift Basics

1.1 Variables and Constants

Swift uses 'var' for mutable variables and 'let' for immutable constants. Type inference allows Swift to automatically determine types.

Code Example:

```
// Variables (mutable)
var playerName = "Alice"
var score = 100
var isActive = true

// Constants (immutable)
let maxPlayers = 4
let gameTitle = "Swift Adventure"
let pi = 3.14159

// Explicit type annotations
var temperature: Double = 98.6
let items: [String] = ["sword", "shield", "potion"]

// Multiple declarations
var x = 0.0, y = 0.0, z = 0.0
let red, green, blue: Double
```

Key Points:

• Use 'let' by default, 'var' only when you need to change the value
• Type inference reduces verbosity while maintaining type safety
• Constants improve performance and prevent accidental mutations

Notes:

Swift encourages immutability through 'let'. The compiler optimizes constants more effectively than variables.

1.2 Data Types

Swift provides various built-in data types including integers, floating-point numbers, booleans, strings, and more.

Code Example:

```
// Integer types
let smallNumber: Int8 = 127
let regularNumber: Int = 42
let bigNumber: Int64 = 9223372036854775807

// Floating-point types
let pi: Float = 3.14159
let precisePi: Double = 3.141592653589793

// Boolean
let isSwiftFun: Bool = true

// Character and String
let letter: Character = "A"
let greeting: String = "Hello, Swift!"

// Type conversion
let integerValue = 42
let floatValue = Float(integerValue)
let stringValue = String(integerValue)

// Type checking
if floatValue is Float {
    print("It's a Float!")
}
```

Key Points:

• Int and Double are the most commonly used numeric types
• Swift doesn't perform implicit type conversions
• Use type conversion initializers for explicit conversions
• Type checking with 'is' operator helps ensure type safety

Notes:

Swift is a type-safe language, preventing type-related errors at compile time.

1.3 Optionals

Optionals represent either a value or nil (absence of value). They're fundamental to Swift's safety model.

Code Example:

```
// Declaring optionals
var optionalString: String? = "Hello"
var optionalInt: Int? = nil

// Optional binding with if-let
if let actualString = optionalString {
    print("The string is: \(actualString)")
} else {
    print("No string value")
}
```

```
// Guard statement
func processString(_ str: String?) {
    guard let unwrapped = str else {
        print("String is nil")
        return
    }
    print("Processing: \(unwrapped)")
}

// Nil-coalescing operator
let defaultName = "Anonymous"
let userName = optionalString ?? defaultName

// Optional chaining
class Person {
    var residence: Residence?
}
class Residence {
    var address: String?
}

let person = Person()
let address = person.residence?.address

// Implicitly unwrapped optionals
var assumedString: String! = "An implicitly unwrapped optional string."
```

Key Points:

• Use optionals to handle absence of values safely
• Prefer optional binding over force unwrapping
• Guard statements provide early exit for nil values
• Optional chaining prevents crashes when accessing nested optionals

Notes:

Optionals eliminate null pointer exceptions and make your code more robust.

1.4 Control Flow

Swift provides various control flow statements including if, switch, loops, and control transfer statements.

Code Example:

```
// If statements
let temperature = 75
if temperature > 80 {
    print("It's hot!")
} else if temperature > 60 {
    print("It's warm")
} else {
    print("It's cool")
}

// Switch statements (powerful in Swift)
let character = "a"
switch character {
case "a", "e", "i", "o", "u":
    print("It's a vowel")
case "b"..."z":
    print("It's a consonant")
default:
    print("Not a letter")
}

// Switch with ranges and where clauses
let point = (2, 3)
switch point {
case (0, 0):
    print("Origin")
case (_, 0):
    print("On x-axis")
case (0, _):
    print("On y-axis")
case let (x, y) where x == y:
    print("On diagonal")
case let (x, y):
    print("Point at (\(x), \(y))")
}

// For loops
for i in 1...5 {
    print("Count: \(i)")
}

let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}

// While loops
var counter = 0
while counter < 3 {
    print(counter)
    counter += 1
}

// Repeat-while (do-while equivalent)
repeat {
    print("This executes at least once")
    counter -= 1
} while counter > 0
```

Key Points:

• Swift's switch statement is exhaustive and doesn't fall through by default
• Pattern matching in switch makes complex conditions elegant
• Range operators (...) and (..<) are useful in loops and switches
• Control transfer statements: continue, break, fallthrough, return, throw

Notes:

Swift's control flow statements are more powerful than many other languages, especially switch statements.

1.5 Functions

Functions are self-contained chunks of code that perform specific tasks. Swift functions are flexible and powerful.

Code Example:

```
// Basic function
func greet(person: String) -> String {
    return "Hello, \(person)!"
}
let greeting = greet(person: "Taylor")

// Function with multiple parameters
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))

// Function with default parameters
func greet(person: String, from hometown: String = "Unknown") -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}

// Variadic parameters
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
print(arithmeticMean(1, 2, 3, 4, 5))

// In-out parameters
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)

// Function types
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

let mathFunction: (Int, Int) -> Int = addTwoInts
print(mathFunction(2, 3))

// Nested functions
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }

    return backward ? stepBackward : stepForward
}
```

Key Points:

- **Parameter labels improve code readability**
- **Default parameters reduce function overloading**
- **inout parameters allow functions to modify external variables**
- **Functions are first-class types in Swift**

Notes:

Swift functions support many advanced features like closures, higher-order functions, and functional programming patterns.

1.6 Closures

Closures are self-contained blocks of functionality that can be passed around. They're similar to lambdas in other languages.

Code Example:

```
// Basic closure syntax
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

// Full closure syntax
let reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})

// Inferring type from context
let reversed1 = names.sorted(by: { s1, s2 in return s1 > s2 })

// Implicit returns
let reversed2 = names.sorted(by: { s1, s2 in s1 > s2 })

// Shorthand argument names
let reversed3 = names.sorted(by: { $0 > $1 })

// Operator method
let reversed4 = names.sorted(by: >)
```

```
// Trailing closure syntax
let reversed5 = names.sorted { $0 > $1 }

// Capturing values
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)
print(incrementByTen()) // 10
print(incrementByTen()) // 20

// Escaping closures
var completionHandlers: [() -> Void] = []

func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}

// Autoclosures
func simpleAssert(_ condition: @autoclosure () -> Bool, _ message: String) {
    if !condition() {
        print(message)
    }
}

let testNumber = 5
simpleAssert(testNumber > 0, "Number must be positive")
```

Key Points:

• Closures can capture and store references to variables and constants
• Trailing closure syntax makes code more readable
• @escaping closures outlive the function that calls them
• @autoclosure automatically wraps expressions in closures

Notes:

Closures are extensively used in Swift for callbacks, functional programming, and asynchronous operations.

1.7 Collections

Swift provides three primary collection types: arrays, sets, and dictionaries for storing multiple values.

Code Example:

```swift
// Arrays
var fruits = ["apple", "banana", "orange"]
fruits.append("grape")
fruits.insert("kiwi", at: 1)

// Array methods
let numbers = [1, 2, 3, 4, 5]
let doubled = numbers.map { $0 * 2 }
let evens = numbers.filter { $0 % 2 == 0 }
let sum = numbers.reduce(0, +)

// Sets
var uniqueNumbers: Set<Int> = [1, 2, 3, 2, 1]
print(uniqueNumbers) // [1, 2, 3]

let set1: Set = [1, 2, 3]
let set2: Set = [3, 4, 5]
let intersection = set1.intersection(set2) // [3]
let union = set1.union(set2) // [1, 2, 3, 4, 5]

// Dictionaries
var studentGrades = ["Alice": 95, "Bob": 87, "Charlie": 92]
studentGrades["Diana"] = 89
studentGrades.updateValue(88, forKey: "Bob")

// Dictionary iteration
for (name, grade) in studentGrades {
    print("\(name): \(grade)")
}

// Nested collections
let matrix: [[Int]] = [[1, 2], [3, 4], [5, 6]]
let coordinates = [(x: 1, y: 2), (x: 3, y: 4)]
```

Key Points:

• Arrays are ordered collections of values
• Sets store unique values in no defined ordering
• Dictionaries store key-value associations
• All collections support functional programming methods

Notes:

Swift collections are type-safe and provide powerful methods for data manipulation.

1.8 Strings

Swift strings are Unicode-compliant and provide powerful manipulation methods.

Code Example:

```swift
// String basics
let greeting = "Hello, World!"
let multilineString = """
    This is a multiline
    string in Swift with
    proper formatting
    """

// String interpolation
let name = "Swift"
let version = 5.0
let message = "Welcome to \(name) \(version)!"

// String methods
let text = "Hello, Swift Programming"
print(text.count) // Character count
print(text.uppercased())
print(text.lowercased())
print(text.hasPrefix("Hello"))
print(text.hasSuffix("Programming"))

// String manipulation
let sentence = "Swift is awesome"
let words = sentence.split(separator: " ")
let joined = words.joined(separator: "-")

// Character iteration
for character in greeting {
    print(character)
}

// String indices
let str = "Swift"
let startIndex = str.startIndex
let endIndex = str.endIndex
let secondChar = str[str.index(after: startIndex)]

// String slicing
let range = str.index(str.startIndex, offsetBy: 1)..<str.index(str.endIndex, offsetBy: -1)
let substring = str[range]

// Regular expressions (iOS 16+)
let pattern = #"\d+"#
if let regex = try? Regex(pattern) {
    let numbers = "Age: 25, Score: 100"
    let matches = numbers.matches(of: regex)
}
```

Key Points:

• Strings are value types and use copy-on-write optimization
• String interpolation with \() is preferred over concatenation
• Strings use String.Index for position-based operations
• Regular expressions provide powerful pattern matching

Notes:

Swift strings are designed for Unicode correctness and international text support.

1.9 Error Handling

Swift provides first-class error handling with do-catch blocks, throwing functions, and the Result type.

Code Example:

```
// Define errors
enum ValidationError: Error {
    case tooShort
    case tooLong
    case invalidCharacters
    case empty
}

// Throwing function
func validatePassword(_ password: String) throws -> Bool {
    if password.isEmpty {
        throw ValidationError.empty
    }

    if password.count < 8 {
        throw ValidationError.tooShort
    }

    if password.count > 50 {
        throw ValidationError.tooLong
    }

    return true
}

// Do-catch block
func testPassword() {
    do {
        try validatePassword("secret")
        print("Password is valid")
    } catch ValidationError.tooShort {
        print("Password is too short")
    } catch ValidationError.empty {
        print("Password cannot be empty")
    } catch {
        print("Unknown error: \(error)")
    }
}

// Try variants
let password1 = try? validatePassword("mypassword") // Returns nil on error
let password2 = try! validatePassword("validpassword") // Crashes on error

// Result type
func validatePasswordResult(_ password: String) -> Result<Bool, ValidationError> {
    do {
        let isValid = try validatePassword(password)
        return .success(isValid)
    } catch let error as ValidationError {
        return .failure(error)
    } catch {
        return .failure(.invalidCharacters)
    }
}

// Using Result
let result = validatePasswordResult("test")
switch result {
case .success(let isValid):
    print("Validation result: \(isValid)")
case .failure(let error):
    print("Validation failed: \(error)")
}

// Rethrowing functions
func processPasswords<T>(_ passwords: [String], processor: (String) throws -> T) rethrows ->
    return try passwords.map(processor)
}
```

Key Points:

- **Use specific error types conforming to Error protocol**
- **Do-catch blocks handle errors gracefully**
- **try? converts errors to optionals, try! force-unwraps**
- **Result type provides functional error handling**

Notes:

Swift's error handling is designed to be explicit and safe, preventing runtime crashes from unhandled errors.

Chapter 2: Object-Oriented Programming

2.1 Classes vs Structures

Swift provides both classes and structures. Understanding when to use each is crucial for effective Swift programming.

Code Example:

```
// Structure (Value Type)
struct Point {
    var x: Double
    var y: Double

    func distanceFromOrigin() -> Double {
        return sqrt(x * x + y * y)
    }

    // Mutating method for value types
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

// Class (Reference Type)
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }

    func makeNoise() {
        // Override in subclass
    }
}

class Bicycle: Vehicle {
    var hasBasket = false

    override func makeNoise() {
        print("Ring ring!")
    }
}

// Identity operators for reference types
let vehicle1 = Vehicle()
let vehicle2 = Vehicle()
let vehicle3 = vehicle1

if vehicle1 === vehicle3 {
    print("Same instance")
}

// Copy behavior difference
var point1 = Point(x: 1.0, y: 2.0)
var point2 = point1  // Copies the value
point2.x = 3.0
print(point1.x)  // Still 1.0

let bike1 = Bicycle()
let bike2 = bike1  // Same reference
bike2.currentSpeed = 10.0
print(bike1.currentSpeed)  // Also 10.0
```

Key Points:

• Structures are value types (copied), classes are reference types (shared)
• Use structures for simple data containers and value semantics
• Use classes when you need inheritance or reference semantics
• Identity operators (=== and !==) compare reference equality

Notes:

Choose structures by default and classes when you specifically need reference semantics.

2.2 Properties

Properties associate values with classes, structures, and enumerations. Swift provides stored and computed properties.

Code Example:

```
// Stored properties
struct FixedLengthRange {
    var firstValue: Int
    let length: Int  // Constant stored property
}

// Lazy stored properties
class DataImporter {
    var filename = "data.txt"
    // Expensive initialization
}

class DataManager {
    lazy var importer = DataImporter()
    var data: [String] = []
}

// Computed properties
struct Point {
    var x = 0.0, y = 0.0
```

```swift
}

struct Size {
    var width = 0.0, height = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()

    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }

    // Read-only computed property
    var area: Double {
        return size.width * size.height
    }
}

// Property observers
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}

// Property wrappers
@propertyWrapper
struct Clamped<T: Comparable> {
    private var value: T
    private let range: ClosedRange<T>

    init(wrappedValue: T, _ range: ClosedRange<T>) {
        self.range = range
        self.value = max(range.lowerBound, min(range.upperBound, wrappedValue))
    }

    var wrappedValue: T {
        get { value }
        set { value = max(range.lowerBound, min(range.upperBound, newValue)) }
    }
```

```
}

struct Player {
    @Clamped(0...100) var health: Int = 100
    @Clamped(0...10) var level: Int = 1
}
```

Key Points:

- **Stored properties store constant and variable values**
- **Computed properties calculate values on-the-fly**
- **Property observers respond to changes in property values**
- **Property wrappers provide reusable property behavior**

Notes:

Properties are a fundamental part of Swift's type system, providing flexible data access patterns.

2.3 Inheritance

Classes can inherit methods, properties, and characteristics from another class. Swift supports single inheritance.

Code Example:

```
// Base class
class Vehicle {
    var currentSpeed = 0.0

    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }

    func makeNoise() {
        // Default implementation
        print("Some generic vehicle noise")
    }
}

// Subclass
class Bicycle: Vehicle {
    var hasBasket = false

    override func makeNoise() {
        print("Ring ring!")
    }
}

// Further subclassing
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
    override var description: String {
        return super.description + " with \(currentNumberOfPassengers) passengers"
    }
}

// Preventing inheritance
final class FinalVehicle: Vehicle {
    // Cannot be subclassed
}

// Overriding properties
class Car: Vehicle {
    var gear = 1

    override var description: String {
        return super.description + " in gear \(gear)"
    }

    // Overriding property observers
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}

// Initialization inheritance
class ElectricCar: Car {
    var batteryLevel: Double

    init(batteryLevel: Double) {
        self.batteryLevel = batteryLevel
        super.init()
        self.currentSpeed = 25.0
    }

    override func makeNoise() {
        print("Whisper quiet...")
    }
}
```

Key Points:

- **Only classes support inheritance in Swift**
- **Use 'override' keyword to override methods and properties**
- **Call superclass methods with 'super'**
- **Use 'final' to prevent inheritance**

Notes:

Inheritance enables code reuse and polymorphism, but favor composition over inheritance when possible.

2.4 Initialization

Initialization is the process of preparing an instance for use. Swift provides designated and convenience initializers.

Code Example:

```
// Basic initialization
struct Celsius {
    var temperatureInCelsius: Double

    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }

    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }

    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}

// Class initialization
class Food {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[Unnamed]")
    }
}

class RecipeIngredient: Food {
    var quantity: Int

    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}

// Failable initializers
struct Animal {
    let species: String

    init?(species: String) {
        if species.isEmpty {
            return nil
        }
        self.species = species
    }
}

// Required initializers
class SomeClass {
    required init() {
        // Implementation
    }
}

class SomeSubclass: SomeClass {
    required init() {
        // Must implement required initializer
    }
}

// Memberwise initializers (structs only)
struct Point {
    var x: Double
    var y: Double
    // Automatically gets init(x:y:)
}

// Deinitialization
class Player {
    let playerName: String

    init(name: String) {
        self.playerName = name
        print("\(playerName) has joined the game")
    }

    deinit {
        print("\(playerName) has left the game")
    }
}
```

Key Points:

- **Designated initializers fully initialize all properties**
- **Convenience initializers call other initializers**
- **Failable initializers return nil if initialization fails**
- **Required initializers must be implemented by all subclasses**

Notes:

Swift's initialization system ensures all properties are initialized before the instance is ready for use.

Chapter 3: Advanced Swift

3.1 Generics

Generics enable you to write flexible, reusable functions and types that can work with any type, subject to requirements you define.

Code Example:

```
// Generic function
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)

// Generic types
struct Stack<Element> {
    var items: [Element] = []

    mutating func push(_ item: Element) {
        items.append(item)
    }

    mutating func pop() -> Element {
        return items.removeLast()
    }

    func peek() -> Element? {
        return items.last
    }
}

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")

// Type constraints
func findIndex<T: Equatable>(of valueToFind: T, in array: [T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

```
// Associated types in protocols
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

struct IntStack: Container {
    typealias Item = Int
    var items: [Int] = []

    mutating func append(_ item: Int) {
        items.append(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> Int {
        return items[i]
    }
}

// Generic where clauses
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

    if someContainer.count != anotherContainer.count {
        return false
    }

    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }

    return true
}
```

Key Points:

- **Generics provide type safety while maintaining flexibility**
- **Type constraints ensure generic types conform to required protocols**
- **Associated types make protocols more flexible**
- **Where clauses add additional requirements to generic functions**

Notes:

Generics are extensively used in Swift's standard library and are key to creating reusable, type-safe code.

3.2 Protocols

Protocols define a blueprint of methods, properties, and requirements that suit a particular task or piece of functionality.

Code Example:

```
// Basic protocol
protocol Drawable {
    func draw()
    var area: Double { get }
    var perimeter: Double { get }
}

// Protocol implementation
struct Circle: Drawable {
    let radius: Double

    func draw() {
        print("Drawing a circle with radius \(radius)")
    }

    var area: Double {
        return .pi * radius * radius
    }

    var perimeter: Double {
        return 2 * .pi * radius
    }
}

// Protocol inheritance
protocol Shape3D: Drawable {
    var volume: Double { get }
}

// Multiple protocol conformance
protocol Identifiable {
    var id: String { get }
}

struct User: Identifiable, CustomStringConvertible {
    let id: String
    let name: String

    var description: String {
        return "User(id: \(id), name: \(name))"
    }
}

// Protocol extensions
extension Drawable {
    func drawWithBorder() {
        print("Drawing border...")
        draw()
        print("Border complete")
    }

    // Default implementation
    var description: String {
        return "A shape with area \(area)"
    }
}

// Protocol with associated types
protocol Container {
    associatedtype Item
    var count: Int { get }
    mutating func append(_ item: Item)
    subscript(i: Int) -> Item { get }
}

struct Stack<Element>: Container {
    var items: [Element] = []

    var count: Int {
        return items.count
    }

    mutating func append(_ item: Element) {
        items.append(item)
    }

    subscript(i: Int) -> Element {
        return items[i]
    }
}

// Protocol composition
protocol Named {
    var name: String { get }
}

protocol Aged {
    var age: Int { get }
}

func greetPerson(_ person: Named & Aged) {
    print("Hello, \(person.name), you are \(person.age) years old")
}

// Checking protocol conformance
if let circle = someObject as? Drawable {
    circle.draw()
}
```

Key Points:

• Protocols define contracts that types must fulfill
• Protocol extensions provide default implementations
• Associated types make protocols generic
• Protocol composition combines multiple protocols

Notes:

Protocols are fundamental to Swift's protocol-oriented programming paradigm.

3.3 Extensions

Extensions add new functionality to existing classes, structures, enumerations, or protocol types without modifying their source code.

Code Example:

```
// Basic extension
extension Double {
    var squared: Double {
        return self * self
    }

    func rounded(toDecimalPlaces places: Int) -> Double {
        let multiplier = pow(10, Double(places))
        return (self * multiplier).rounded() / multiplier
    }
}

let number = 3.14159
print(number.squared) // 9.8696
print(number.rounded(toDecimalPlaces: 2)) // 3.14

// Extension with initializers
extension String {
    init(repeating character: Character, count: Int) {
        self = String(Array(repeating: character, count: count))
    }

    var isPalindrome: Bool {
        let cleaned = self.lowercased().filter { $0.isLetter }
        return cleaned == String(cleaned.reversed())
    }
}

let stars = String(repeating: "*", count: 5) // "*****"
print("racecar".isPalindrome) // true

// Extension with subscripts
extension Array {
    subscript(safe index: Int) -> Element? {
        return indices.contains(index) ? self[index] : nil
    }
}

let numbers = [1, 2, 3, 4, 5]
print(numbers[safe: 10]) // nil instead of crash

// Extension with nested types
```

```
extension Character {
    enum Kind {
        case vowel
        case consonant
        case other
    }

    var kind: Kind {
        switch lowercased() {
        case "a", "e", "i", "o", "u":
            return .vowel
        case "a"..."z":
            return .consonant
        default:
            return .other
        }
    }
}

let char: Character = "E"
print(char.kind) // vowel

// Generic extension
extension Array where Element: Comparable {
    func quickSorted() -> [Element] {
        guard count > 1 else { return self }

        let pivot = self[count / 2]
        let less = self.filter { $0 < pivot }
        let equal = self.filter { $0 == pivot }
        let greater = self.filter { $0 > pivot }

        return less.quickSorted() + equal + greater.quickSorted()
    }
}

let unsorted = [3, 1, 4, 1, 5, 9, 2, 6]
let sorted = unsorted.quickSorted()
```

Key Points:

- **Extensions add functionality without modifying original code**
- **Can add computed properties, methods, initializers, and subscripts**
- **Generic extensions can add conditional functionality**
- **Extensions can conform types to protocols**

Notes:

Extensions are a powerful way to organize code and add functionality to existing types.

PART II: CONCURRENCY & MODERN SWIFT

Chapter 4: Concurrency

4.1 Async/Await

Swift's async/await syntax provides a clean way to write asynchronous code that reads like synchronous code.

Code Example:

```
// Basic async function
func fetchUserData(id: String) async throws -> User {
    let url = URL(string: "https://api.example.com/users/\(id)")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(User.self, from: data)
}

// Calling async functions
func loadUserProfile() async {
    do {
        let user = try await fetchUserData(id: "123")
        print("Loaded user: \(user.name)")
    } catch {
        print("Failed to load user: \(error)")
    }
}

// Async properties
class ImageLoader {
    private var cache: [URL: UIImage] = [:]

    var imageCount: Int {
        cache.count
    }

    func image(from url: URL) async throws -> UIImage {
        // Check cache first
        if let cachedImage = cache[url] {
            return cachedImage
        }

        // Download image
        let (data, _) = try await URLSession.shared.data(from: url)
        guard let image = UIImage(data: data) else {
            throw ImageError.invalidData
        }

        // Cache the image
        cache[url] = image
        return image
    }
}
```

```
// Async sequences
func countDown(from number: Int) -> AsyncStream<Int> {
    AsyncStream { continuation in
        Task {
            for i in (0...number).reversed() {
                continuation.yield(i)
                try await Task.sleep(nanoseconds: 1_000_000_000) // 1 second
            }
            continuation.finish()
        }
    }
}

// Using async sequences
func runCountdown() async {
    for await count in countDown(from: 5) {
        print("Count: \(count)")
    }
    print("Done!")
}

// Async/await with completion handlers
func legacyNetworkCall(completion: @escaping (Result<Data, Error>) -> Void) {
    // Legacy callback-based code
}

// Convert to async/await
func modernNetworkCall() async throws -> Data {
    return try await withCheckedThrowingContinuation { continuation in
        legacyNetworkCall { result in
            continuation.resume(with: result)
        }
    }
}

// Multiple concurrent operations
func loadMultipleUsers() async throws -> [User] {
    async let user1 = fetchUserData(id: "1")
    async let user2 = fetchUserData(id: "2")
    async let user3 = fetchUserData(id: "3")

    return try await [user1, user2, user3]
}
```

Key Points:

- **async functions must be called with await**
- **async/await eliminates callback hell**
- **Use async let for concurrent operations**
- **AsyncSequence provides asynchronous iteration**

Notes:

Async/await makes asynchronous code more readable and easier to debug than callback-based approaches.

4.2 Tasks

Tasks represent units of asynchronous work. Swift provides Task, TaskGroup, and various cancellation mechanisms.

Code Example:

```
// Basic Task creation
func startBackgroundWork() {
    Task {
        let result = await performLongRunningOperation()
        await updateUI(with: result)
    }
}

// Task with priority
func highPriorityWork() {
    Task(priority: .high) {
        await performCriticalOperation()
    }
}

// Detached tasks
func detachedWork() {
    Task.detached {
        // This task doesn't inherit context
        await performIndependentWork()
    }
}

// TaskGroup for multiple concurrent operations
func processItemsConcurrently(_ items: [String]) async -> [ProcessedItem] {
    await withTaskGroup(of: ProcessedItem.self) { group in
        var results: [ProcessedItem] = []

        for item in items {
            group.addTask {
                return await processItem(item)
            }
        }

        for await result in group {
            results.append(result)
        }

        return results
    }
}

// Error handling in TaskGroup
func processWithErrorHandling(_ items: [String]) async -> [ProcessedItem] {
    await withTaskGroup(of: Result<ProcessedItem, Error>.self) { group in
        var results: [ProcessedItem] = []
        for item in items {
            group.addTask {
                do {
                    let processed = try await processItemThrowing(item)
                    return .success(processed)
                } catch {
                    return .failure(error)
                }
            }
        }

        for await result in group {
            switch result {
            case .success(let item):
                results.append(item)
            case .failure(let error):
                print("Failed to process item: \(error)")
            }
        }

        return results
    }
}

// Task cancellation
class DataProcessor {
    private var currentTask: Task<Void, Error>?

    func startProcessing() {
        currentTask = Task {
            for i in 1...1000 {
                // Check for cancellation
                try Task.checkCancellation()

                await processItem(i)

                // Alternative cancellation check
                if Task.isCancelled {
                    print("Task was cancelled")
                    return
                }
            }
        }
    }

    func cancelProcessing() {
        currentTask?.cancel()
    }
}

// Task local values
enum TaskLocals {
    @TaskLocal static var userID: String?
    @TaskLocal static var requestID: String = UUID().uuidString
}

func performUserOperation() async {
    await TaskLocals.$userID.withValue("user123") {
        await TaskLocals.$requestID.withValue("req456") {
            await someOperation()
            // userID and requestID are available here
        }
    }
}
```

Key Points:

- **Task represents a unit of asynchronous work**
- **TaskGroup enables structured concurrency for multiple operations**
- **Tasks can be cancelled cooperatively**
- **Task-local values provide context inheritance**

Notes:

Tasks provide structured concurrency, making concurrent code predictable and manageable.

4.3 Actors

Actors provide data isolation and protect against data races in concurrent programming.

Code Example:

```
// Basic actor
actor Counter {
    private var value = 0

    func increment() -> Int {
        value += 1
        return value
    }

    func getValue() -> Int {
        return value
    }

    func reset() {
        value = 0
    }
}

// Using actors
func useCounter() async {
    let counter = Counter()

    let value1 = await counter.increment() // Must use await
    let value2 = await counter.getValue()

    print("Counter values: \(value1), \(value2)")
}
```

```
// Actor with async methods
actor ImageCache {
    private var images: [URL: UIImage] = [:]

    func image(for url: URL) async -> UIImage? {
        if let cached = images[url] {
            return cached
        }

        // Fetch image
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            let image = UIImage(data: data)
            images[url] = image
            return image
        } catch {
            return nil
        }
    }

    func clearCache() {
        images.removeAll()
    }
}

// MainActor for UI updates
@MainActor
class ViewModel: ObservableObject {
    @Published var data: [String] = []

    func loadData() async {
        let newData = await fetchDataFromNetwork()

        // This runs on main actor automatically
        self.data = newData
    }

    // Non-isolated methods can be called from any context
    nonisolated func validateInput(_ input: String) -> Bool {
        return !input.isEmpty
    }
}

// Global actor
@globalActor
actor DatabaseActor {
    static let shared = DatabaseActor()
    private init() {}
}

@DatabaseActor
func saveToDatabase(_ data: Data) {
    // All calls to this function are serialized
    // through the DatabaseActor
}

// Actor inheritance (only from protocols)
```

```
protocol Drawable {
    func draw() async
}

actor DrawingCanvas: Drawable {
    private var shapes: [Shape] = []

    func draw() async {
        for shape in shapes {
            await shape.render()
        }
    }

    func addShape(_ shape: Shape) {
        shapes.append(shape)
    }
}

// Sendable types for actor boundaries
struct SafeData: Sendable {
    let id: String
    let value: Int
}

actor DataProcessor {
    func process(_ data: SafeData) async -> ProcessedData {
        // Safe to pass Sendable types across actor boundaries
        return await processData(data)
    }
}
```

Key Points:

- **Actors provide data isolation and prevent data races**
- **Actor methods are called with await from outside the actor**
- **MainActor ensures UI updates happen on the main thread**
- **Sendable types can be safely passed between actors**

Notes:

Actors are Swift's solution to thread-safe programming without explicit locks or queues.

PART III: SwiftUI FRAMEWORK

Chapter 5: SwiftUI Fundamentals

5.1 Views and Modifiers

SwiftUI uses a declarative syntax where you describe what your UI should look like. Views are modified using modifiers that return new views.

Code Example:

```
import SwiftUI

// Basic views
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, SwiftUI!")
                .font(.largeTitle)
                .foregroundColor(.blue)
                .padding()

            Image(systemName: "star.fill")
                .foregroundColor(.yellow)
                .font(.system(size: 50))

            Button("Tap Me") {
                print("Button tapped!")
            }
            .padding()
            .background(Color.blue)
            .foregroundColor(.white)
            .cornerRadius(10)
        }
    }
}

// Custom view with modifiers
struct CustomCard: View {
    let title: String
    let subtitle: String

    var body: some View {
        VStack(alignment: .leading) {
            Text(title)
                .font(.headline)
                .fontWeight(.bold)

            Text(subtitle)
                .font(.subheadline)
                .foregroundColor(.gray)
        }
        .padding()
        .background(Color.white)
        .cornerRadius(10)
        .shadow(radius: 5)
    }
}

// View composition
struct MainView: View {
    var body: some View {
        ScrollView {
            LazyVStack(spacing: 16) {
                CustomCard(title: "SwiftUI", subtitle: "Declarative UI framework")
                CustomCard(title: "Combine", subtitle: "Reactive programming")
                CustomCard(title: "Swift", subtitle: "Programming language")
            }
            .padding()
        }
    }
}

// ViewBuilder and conditional views
struct ConditionalView: View {
    @State private var showDetails = false

    var body: some View {
        VStack {
            Text("Main Content")

            if showDetails {
                Text("Additional Details")
                    .transition(.opacity)
            }

            Button(showDetails ? "Hide" : "Show") {
                withAnimation {
                    showDetails.toggle()
                }
            }
        }
    }
}
```

Key Points:

- **Views are value types that describe UI declaratively**
- **Modifiers return new views, enabling method chaining**
- **View composition creates reusable components**
- **ViewBuilder enables conditional and loop-based view construction**

Notes:

SwiftUI's declarative approach means you describe the desired end state, and SwiftUI figures out how to get there.

5.2 Layout System

SwiftUI provides powerful layout containers like VStack, HStack, ZStack, and LazyGrids for organizing views.

Code Example:

```
import SwiftUI

// Basic stacks
struct LayoutExamples: View {
    var body: some View {
        VStack(spacing: 20) {
            // Horizontal stack
            HStack {
                Text("Left")
                Spacer()
                Text("Right")
            }
            .padding()
            .background(Color.gray.opacity(0.2))

            // Vertical stack with alignment
            VStack(alignment: .leading, spacing: 10) {
                Text("Title")
                    .font(.headline)
                Text("This is a longer subtitle that demonstrates alignment")
                    .font(.caption)
            }
            .frame(maxWidth: .infinity, alignment: .leading)
            .padding()
            .background(Color.blue.opacity(0.1))

            // Overlay stack
            ZStack {
                Rectangle()
                    .fill(Color.orange)
                    .frame(width: 100, height: 100)

                Text("Overlay")
                    .foregroundColor(.white)
                    .font(.caption)
            }
        }
    }
}

// Grid layouts
struct GridExample: View {
    let items = Array(1...20)

    let columns = [
        GridItem(.flexible()),
        GridItem(.flexible()),
        GridItem(.flexible())
    ]
```

```swift
    var body: some View {
        ScrollView {
            LazyVGrid(columns: columns, spacing: 10) {
                ForEach(items, id: \.self) { item in
                    RoundedRectangle(cornerRadius: 8)
                        .fill(Color.blue)
                        .frame(height: 50)
                        .overlay(
                            Text("\(item)")
                                .foregroundColor(.white)
                        )
                }
            }
            .padding()
        }
    }
}

// Adaptive grids
struct AdaptiveGridExample: View {
    let items = Array(1...50)

    var body: some View {
        ScrollView {
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 80))], spacing: 10) {
                ForEach(items, id: \.self) { item in
                    Circle()
                        .fill(Color.green)
                        .frame(height: 80)
                        .overlay(
                            Text("\(item)")
                                .foregroundColor(.white)
                        )
                }
            }
            .padding()
        }
    }
}

// GeometryReader for custom layouts
struct CustomLayoutView: View {
    var body: some View {
        GeometryReader { geometry in
            VStack {
                Rectangle()
                    .fill(Color.red)
                    .frame(width: geometry.size.width * 0.8, height: 50)

                HStack {
                    Rectangle()
                        .fill(Color.blue)
                        .frame(width: geometry.size.width * 0.4, height: 100)

                    Spacer()

                    Rectangle()
```

```
                        .fill(Color.green)
                        .frame(width: geometry.size.width * 0.4, height: 100)
                }
            }
            .frame(width: geometry.size.width, height: geometry.size.height)
        }
    }
}
```

Key Points:

• VStack, HStack, and ZStack are the fundamental layout containers
• Spacer() pushes views apart or centers them
• LazyVGrid and LazyHGrid create efficient grid layouts
• GeometryReader provides access to parent view dimensions

Notes:

SwiftUI's layout system is designed to be predictable and easy to understand while being highly flexible.

PART IV: REACTIVE PROGRAMMING

Chapter 7: Combine Framework

7.1 Publishers and Subscribers

Combine is Apple's framework for handling asynchronous events by combining event-processing operators. Publishers emit values over time, and subscribers receive them.

Code Example:

```
import Combine
import Foundation

// Basic publisher and subscriber
class CombineBasics {
    var cancellables = Set<AnyCancellable>()

    func basicPublisherSubscriber() {
        // Simple publisher
        let publisher = Just("Hello, Combine!")

        publisher
            .sink { value in
                print("Received: \(value)")
            }
            .store(in: &cancellables)

        // Array publisher
        let numbers = [1, 2, 3, 4, 5]
        numbers.publisher
            .sink { number in
                print("Number: \(number)")
            }
            .store(in: &cancellables)
    }

    // PassthroughSubject
    func passthroughSubjectExample() {
        let subject = PassthroughSubject<String, Never>()

        subject
            .sink { value in
                print("PassthroughSubject received: \(value)")
            }
            .store(in: &cancellables)

        subject.send("First message")
        subject.send("Second message")
        subject.send(completion: .finished)
    }

    // CurrentValueSubject
func currentValueSubjectExample() {
    let currentValueSubject = CurrentValueSubject<Int, Never>(0)

    currentValueSubject
        .sink { value in
            print("CurrentValueSubject: \(value)")
        }
        .store(in: &cancellables)

    currentValueSubject.send(1)
    currentValueSubject.send(2)

    print("Current value: \(currentValueSubject.value)")
}

// Custom publisher
struct CountdownPublisher: Publisher {
    typealias Output = Int
    typealias Failure = Never

    let start: Int

    func receive<S>(subscriber: S) where S : Subscriber, Never == S.Failure, Int == S.In
        let subscription = CountdownSubscription(subscriber: subscriber, start: start)
        subscriber.receive(subscription: subscription)
    }
}

class CountdownSubscription<S: Subscriber>: Subscription where S.Input == Int, S.Failure
    private var subscriber: S?
    private var current: Int

    init(subscriber: S, start: Int) {
        self.subscriber = subscriber
        self.current = start
    }

    func request(_ demand: Subscribers.Demand) {
        var demand = demand

        while demand > 0 && current > 0 {
            _ = subscriber?.receive(current)
            current -= 1
            demand -= 1
        }

        if current == 0 {
            subscriber?.receive(completion: .finished)
        }
    }

    func cancel() {
        subscriber = nil
    }
}

func customPublisherExample() {
```

```
            CountdownPublisher(start: 5)
                .sink { value in
                    print("Countdown: \(value)")
                }
                .store(in: &cancellables)
        }
    }
```

Key Points:

- **Publishers emit values over time, subscribers receive them**
- **PassthroughSubject sends values to subscribers without storing current value**
- **CurrentValueSubject maintains and emits the current value to new subscribers**
- **Custom publishers implement the Publisher protocol**

Notes:

Combine follows the reactive programming paradigm, making asynchronous code more manageable and composable.

PART V: NETWORKING & APIs

Chapter 8: Networking

8.1 URLSession

URLSession is the foundation of networking in iOS. It provides APIs for making HTTP requests with modern async/await support.

Code Example:

```
import Foundation

// Basic URLSession with async/await
class NetworkManager {
    static let shared = NetworkManager()
    private init() {}

    // Simple GET request
    func fetchData(from url: URL) async throws -> Data {
        let (data, response) = try await URLSession.shared.data(from: url)

        guard let httpResponse = response as? HTTPURLResponse,
              httpResponse.statusCode == 200 else {
            throw NetworkError.invalidResponse
        }

        return data
    }

    // POST request with JSON
    func postJSON<T: Codable>(to url: URL, body: T) async throws -> Data {
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(AuthManager.token)", forHTTPHeaderField: "Authorization")

        request.httpBody = try JSONEncoder().encode(body)

        let (data, response) = try await URLSession.shared.data(for: request)

        guard let httpResponse = response as? HTTPURLResponse,
              200...299 ~= httpResponse.statusCode else {
            throw NetworkError.serverError(response)
        }

        return data
    }

    // Download file with progress
    func downloadFile(from url: URL) -> AsyncThrowingStream<DownloadProgress, Error> {
        AsyncThrowingStream { continuation in
            let task = URLSession.shared.downloadTask(with: url) { localURL, response, error
```

```
                if let error = error {
                    continuation.finish(throwing: error)
                    return
                }

                guard let localURL = localURL else {
                    continuation.finish(throwing: NetworkError.noData)
                    return
                }

                // Move file to permanent location
                // continuation.yield(.completed(localURL))
                continuation.finish()
            }

            task.resume()
        }
    }

    // URLSession with custom configuration
    func createCustomSession() -> URLSession {
        let config = URLSessionConfiguration.default
        config.timeoutIntervalForRequest = 30
        config.timeoutIntervalForResource = 60
        config.httpMaximumConnectionsPerHost = 5
        config.requestCachePolicy = .reloadIgnoringLocalCacheData

        return URLSession(configuration: config)
    }

    // Retry mechanism
    func fetchWithRetry<T: Codable>(url: URL, type: T.Type, maxRetries: Int = 3) async throw
        var lastError: Error?

        for attempt in 1...maxRetries {
            do {
                let data = try await fetchData(from: url)
                return try JSONDecoder().decode(T.self, from: data)
            } catch {
                lastError = error
                if attempt < maxRetries {
                    let delay = Double(attempt * 2) // Exponential backoff
                    try await Task.sleep(nanoseconds: UInt64(delay * 1_000_000_000))
                }
            }
        }

        throw lastError ?? NetworkError.maxRetriesExceeded
    }
}

// Error handling
enum NetworkError: Error, LocalizedError {
    case invalidURL
    case noData
    case invalidResponse
    case serverError(URLResponse?)
```

```
        case decodingError
        case maxRetriesExceeded

        var errorDescription: String? {
            switch self {
            case .invalidURL:
                return "Invalid URL"
            case .noData:
                return "No data received"
            case .invalidResponse:
                return "Invalid response"
            case .serverError:
                return "Server error"
            case .decodingError:
                return "Failed to decode data"
            case .maxRetriesExceeded:
                return "Maximum retry attempts exceeded"
            }
        }
    }

    // Progress tracking
    struct DownloadProgress {
        let bytesWritten: Int64
        let totalBytesWritten: Int64
        let totalBytesExpectedToWrite: Int64

        var progress: Double {
            guard totalBytesExpectedToWrite > 0 else { return 0 }
            return Double(totalBytesWritten) / Double(totalBytesExpectedToWrite)
        }
    }
```

Key Points:

- **async/await makes networking code more readable and maintainable**
- **Always handle HTTP status codes and potential errors**
- **URLSession configuration allows customization of timeouts and caching**
- **Implement retry mechanisms for robust networking**

Notes:

Modern Swift networking leverages async/await for cleaner asynchronous code without callback hell.

APPENDIX: DATA STRUCTURES & ALGORITHMS

A.1 Array Problems

Two Sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

Code Example:

```
func twoSum(_ nums: [Int], _ target: Int) -> [Int] {
    var numToIndex: [Int: Int] = [:]

    for (index, num) in nums.enumerated() {
        let complement = target - num

        if let complementIndex = numToIndex[complement] {
            return [complementIndex, index]
        }

        numToIndex[num] = index
    }

    return []
}

// Example usage:
let nums = [2, 7, 11, 15]
let target = 9
let result = twoSum(nums, target)
print(result) // [0, 1]
```

Key Points:

• Time Complexity: O(n)
• Space Complexity: O(n)

Notes:

Uses hash map to store each number and its index, enabling O(1) lookup time.

Maximum Subarray (Kadane's Algorithm)

Find the contiguous subarray with the largest sum.

Code Example:

```
func maxSubArray(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }

    var maxSum = nums[0]
    var currentSum = nums[0]

    for i in 1..<nums.count {
        currentSum = max(nums[i], currentSum + nums[i])
        maxSum = max(maxSum, currentSum)
    }

    return maxSum
}

// Example usage:
let nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
let result = maxSubArray(nums)
print(result) // 6 (subarray [4, -1, 2, 1])
```

Key Points:

- **Time Complexity: O(n)**
- **Space Complexity: O(1)**

Notes:

Kadane's algorithm maintains the maximum sum ending at each position.