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

Chapter 10: Testing & Architecture

APPENDIX: DATA STRUCTURES & ALGORITHMS

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:

```
// 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:

```
// 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
```

```
}

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:

```swift
// 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)")
}
```

```swift
// 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.

5.3 State Management

SwiftUI uses various property wrappers to manage state and data flow in your application.

Code Example:

```
import SwiftUI

// @State for local state
struct CounterView: View {
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Count: \(count)")
                .font(.largeTitle)

            HStack {
                Button("Increment") {
                    count += 1
                }

                Button("Decrement") {
                    count -= 1
                }

                Button("Reset") {
                    count = 0
                }
            }
        }
    }
}

// @Binding for two-way data flow
struct ToggleView: View {
    @Binding var isOn: Bool

    var body: some View {
        Toggle("Feature Enabled", isOn: $isOn)
            .padding()
    }
}

struct ParentView: View {
    @State private var featureEnabled = false

    var body: some View {
        VStack {
            Text("Feature is \(featureEnabled ? "ON" : "OFF")")
            ToggleView(isOn: $featureEnabled)
        }
    }
}

// @ObservableObject and @Published
```

```swift
class UserStore: ObservableObject {
    @Published var users: [User] = []
    @Published var isLoading = false
    @Published var errorMessage: String?

    func loadUsers() async {
        isLoading = true
        errorMessage = nil

        do {
            let fetchedUsers = try await NetworkService.fetchUsers()
            await MainActor.run {
                self.users = fetchedUsers
                self.isLoading = false
            }
        } catch {
            await MainActor.run {
                self.errorMessage = error.localizedDescription
                self.isLoading = false
            }
        }
    }
}

struct UserListView: View {
    @StateObject private var userStore = UserStore()

    var body: some View {
        NavigationView {
            Group {
                if userStore.isLoading {
                    ProgressView("Loading users...")
                } else if let error = userStore.errorMessage {
                    Text("Error: \(error)")
                        .foregroundColor(.red)
                } else {
                    List(userStore.users) { user in
                        UserRowView(user: user)
                    }
                }
            }
            .navigationTitle("Users")
            .task {
                await userStore.loadUsers()
            }
        }
    }
}

// @EnvironmentObject for dependency injection
struct ContentView: View {
    var body: some View {
        TabView {
            UserListView()
                .tabItem {
                    Image(systemName: "person.3")
                    Text("Users")
                }

            SettingsView()
                .tabItem {
                    Image(systemName: "gear")
                    Text("Settings")
                }
        }
        .environmentObject(UserStore())
    }
}

// @AppStorage for UserDefaults
struct SettingsView: View {
    @AppStorage("username") private var username = ""
    @AppStorage("isDarkMode") private var isDarkMode = false
    @AppStorage("fontSize") private var fontSize = 16.0

    var body: some View {
        Form {
            Section("User Preferences") {
                TextField("Username", text: $username)

                Toggle("Dark Mode", isOn: $isDarkMode)

                Stepper("Font Size: \(Int(fontSize))", value: $fontSize, in: 12...24)
            }
        }
        .preferredColorScheme(isDarkMode ? .dark : .light)
    }
}
```

Key Points:

- **@State manages local view state**
- **@Binding creates two-way data connections**
- **@StateObject creates and owns ObservableObject instances**
- **@EnvironmentObject shares data across the view hierarchy**
- **@AppStorage automatically syncs with UserDefaults**

Notes:

> *SwiftUI's reactive state management system automatically updates views when data changes.*

5.4 Navigation

SwiftUI provides various navigation patterns including NavigationView, NavigationLink, and programmatic navigation.

Code Example:

```
import SwiftUI
```

```
// Basic Navigation
struct ContentView: View {
    var body: some View {
        NavigationView {
            List {
                NavigationLink("Profile", destination: ProfileView())
                NavigationLink("Settings", destination: SettingsView())
                NavigationLink("About", destination: AboutView())
            }
            .navigationTitle("Main Menu")
            .navigationBarTitleDisplayMode(.large)
        }
    }
}

// NavigationStack (iOS 16+)
struct ModernNavigationView: View {
    @State private var path = NavigationPath()

    var body: some View {
        NavigationStack(path: $path) {
            List {
                Button("Go to Detail") {
                    path.append("detail")
                }

                Button("Go to Settings") {
                    path.append("settings")
                }
            }
            .navigationDestination(for: String.self) { value in
                switch value {
                case "detail":
                    DetailView()
                case "settings":
                    SettingsView()
                default:
                    Text("Unknown destination")
                }
            }
            .navigationTitle("Navigation Stack")
        }
    }
}

// Programmatic navigation
class NavigationController: ObservableObject {
    @Published var isShowingDetail = false
    @Published var selectedUser: User?

    func showUserDetail(_ user: User) {
        selectedUser = user
        isShowingDetail = true
    }

    func dismissDetail() {
        isShowingDetail = false
        selectedUser = nil
    }
}

struct UserListView: View {
    @StateObject private var navigation = NavigationController()
    @State private var users: [User] = []

    var body: some View {
        NavigationView {
            List(users) { user in
                Button(user.name) {
                    navigation.showUserDetail(user)
                }
            }
            .navigationTitle("Users")
            .sheet(isPresented: $navigation.isShowingDetail) {
                if let user = navigation.selectedUser {
                    UserDetailView(user: user)
                }
            }
        }
    }
}

// Tab Navigation
struct MainTabView: View {
    @State private var selectedTab = 0

    var body: some View {
        TabView(selection: $selectedTab) {
            HomeView()
                .tabItem {
                    Image(systemName: "house")
                    Text("Home")
                }
                .tag(0)

            SearchView()
                .tabItem {
                    Image(systemName: "magnifyingglass")
                    Text("Search")
                }
                .tag(1)

            ProfileView()
                .tabItem {
                    Image(systemName: "person")
                    Text("Profile")
                }
                .tag(2)
        }
        .accentColor(.blue)
    }
}

// Modal presentation
```

```
struct ModalExampleView: View {
    @State private var isShowingModal = false
    @State private var isShowingFullScreen = false

    var body: some View {
        VStack(spacing: 20) {
            Button("Show Sheet") {
                isShowingModal = true
            }
            .sheet(isPresented: $isShowingModal) {
                ModalContentView(isPresented: $isShowingModal)
            }

            Button("Show Full Screen") {
                isShowingFullScreen = true
            }
            .fullScreenCover(isPresented: $isShowingFullScreen) {
                FullScreenView(isPresented: $isShowingFullScreen)
            }
        }
    }
}

// Navigation with data passing
struct ProductListView: View {
    @State private var products: [Product] = []

    var body: some View {
        NavigationView {
            List(products) { product in
                NavigationLink(destination: ProductDetailView(product: product)) {
                    ProductRowView(product: product)
                }
            }
            .navigationTitle("Products")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        // Add new product
                    }
                }
            }
        }
    }
}
```

Key Points:

- **NavigationView provides the foundation for navigation**
- **NavigationLink creates navigable connections between views**
- **NavigationStack (iOS 16+) offers more flexible navigation**
- **Sheet and fullScreenCover present modal views**
- **TabView creates tab-based navigation**

Notes:

SwiftUI navigation is declarative and integrates seamlessly with the framework's reactive patterns.

5.5 Animations

SwiftUI provides powerful animation capabilities with simple, declarative syntax.

Code Example:

```
import SwiftUI

// Basic animations
struct AnimationExamples: View {
    @State private var isRotated = false
    @State private var scale: CGFloat = 1.0
    @State private var offset: CGFloat = 0

    var body: some View {
        VStack(spacing: 40) {
            // Rotation animation
            Rectangle()
                .fill(Color.blue)
                .frame(width: 50, height: 50)
                .rotationEffect(.degrees(isRotated ? 180 : 0))
                .animation(.easeInOut(duration: 1), value: isRotated)
                .onTapGesture {
                    isRotated.toggle()
                }

            // Scale animation
            Circle()
                .fill(Color.green)
                .frame(width: 50, height: 50)
                .scaleEffect(scale)
                .animation(.spring(response: 0.5, dampingFraction: 0.6), value: scale)
                .onTapGesture {
                    scale = scale == 1.0 ? 1.5 : 1.0
                }

            // Offset animation
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.orange)
                .frame(width: 50, height: 50)
                .offset(x: offset)
                .animation(.bouncy, value: offset)
                .onTapGesture {
                    offset = offset == 0 ? 100 : 0
                }
        }
    }
}

// withAnimation for explicit animation
```

```
struct ExplicitAnimationView: View {
    @State private var isExpanded = false

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.purple)
                .frame(width: isExpanded ? 200 : 100, height: isExpanded ? 200 : 100)

            Button("Animate") {
                withAnimation(.spring(duration: 0.8)) {
                    isExpanded.toggle()
                }
            }
        }
    }
}

// Transitions
struct TransitionView: View {
    @State private var showDetail = false

    var body: some View {
        VStack {
            if showDetail {
                VStack {
                    Text("Detail View")
                        .font(.largeTitle)
                        .padding()

                    Text("This is additional detail information")
                        .padding()
                }
                .background(Color.gray.opacity(0.1))
                .cornerRadius(10)
                .transition(.asymmetric(
                    insertion: .move(edge: .trailing).combined(with: .opacity),
                    removal: .move(edge: .leading).combined(with: .opacity)
                ))
            }

            Button(showDetail ? "Hide" : "Show") {
                withAnimation(.easeInOut) {
                    showDetail.toggle()
                }
            }
        }
        .padding()
    }
}

// Custom animations
struct WaveView: View {
    @State private var animateWave = false

    var body: some View {
        ZStack {
            ForEach(0..<3) { index in
                Circle()
                    .stroke(Color.blue.opacity(0.3), lineWidth: 2)
                    .frame(width: 50, height: 50)
                    .scaleEffect(animateWave ? 3 : 1)
                    .opacity(animateWave ? 0 : 1)
                    .animation(.easeOut(duration: 2).repeatForever(autoreverses: false).dela
            }
        }
        .onAppear {
            animateWave = true
        }
    }
}

// Complex animation sequences
struct LoadingView: View {
    @State private var isLoading = false

    var body: some View {
        HStack {
            ForEach(0..<3) { index in
                Circle()
                    .fill(Color.blue)
                    .frame(width: 10, height: 10)
                    .scaleEffect(isLoading ? 1.0 : 0.5)
                    .animation(.easeInOut(duration: 0.6).repeatForever().delay(0.2 * Double(
            }
        }
        .onAppear {
            isLoading = true
        }
    }
}

// Gesture-driven animations
struct DragView: View {
    @State private var dragAmount = CGSize.zero

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.red)
                .frame(width: 100, height: 100)
                .offset(dragAmount)
                .gesture(
                    DragGesture()
                        .onChanged { dragAmount = $0.translation }
                        .onEnded { _ in
                            withAnimation(.spring()) {
                                dragAmount = .zero
                            }
                        }
                )

            Text("Drag the square!")
                .padding()
        }
    }
}
```

Key Points:

- **Use .animation() modifier for implicit animations**
- **withAnimation provides explicit control over animations**
- **Transitions define how views appear and disappear**
- **Spring animations provide natural motion**
- **Combine animations with gestures for interactive experiences**

Notes:

SwiftUI animations are declarative and automatically handle the complex details of smooth transitions.

Chapter 6: Advanced SwiftUI

6.1 Custom Views and ViewModifiers

Create reusable custom views and view modifiers to build sophisticated UI components.

Code Example:

```
import SwiftUI

// Custom View Components
struct PrimaryButton: View {
    let title: String
    let action: () -> Void
    @State private var isPressed = false

    var body: some View {
        Button(action: action) {
            Text(title)
                .font(.headline)
                .foregroundColor(.white)
                .frame(maxWidth: .infinity)
                .padding()
                .background(
                    RoundedRectangle(cornerRadius: 12)
                        .fill(Color.blue)
                        .scaleEffect(isPressed ? 0.95 : 1.0)
                )
        }
        .buttonStyle(PlainButtonStyle())
        .onLongPressGesture(minimumDuration: 0, maximumDistance: .infinity, pressing: { pres
            withAnimation(.easeInOut(duration: 0.1)) {
                isPressed = pressing
            }
        }, perform: {})
    }
}

// Custom ViewModifier
struct CardModifier: ViewModifier {
    let cornerRadius: CGFloat
    let shadowRadius: CGFloat

    func body(content: Content) -> some View {
        content
            .background(Color(.systemBackground))
            .cornerRadius(cornerRadius)
            .shadow(color: Color.black.opacity(0.1), radius: shadowRadius, x: 0, y: 2)
            .padding(.horizontal)
    }
}
```

```
extension View {
    func cardStyle(cornerRadius: CGFloat = 12, shadowRadius: CGFloat = 8) -> some View {
        self.modifier(CardModifier(cornerRadius: cornerRadius, shadowRadius: shadowRadius))
    }
}

// Custom Shape
struct CurvedRectangle: Shape {
    var cornerRadius: CGFloat
    var curveHeight: CGFloat

    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: 0, y: curveHeight))
        path.addQuadCurve(to: CGPoint(x: rect.width, y: curveHeight),
                         control: CGPoint(x: rect.width / 2, y: 0))
        path.addLine(to: CGPoint(x: rect.width, y: rect.height - cornerRadius))
        path.addQuadCurve(to: CGPoint(x: rect.width - cornerRadius, y: rect.height),
                         control: CGPoint(x: rect.width, y: rect.height))
        path.addLine(to: CGPoint(x: cornerRadius, y: rect.height))
        path.addQuadCurve(to: CGPoint(x: 0, y: rect.height - cornerRadius),
                         control: CGPoint(x: 0, y: rect.height))
        path.closeSubpath()

        return path
    }
}

// Custom Progress View
struct CircularProgressView: View {
    let progress: Double
    let lineWidth: CGFloat = 8

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color.gray.opacity(0.3), lineWidth: lineWidth)

            Circle()
                .trim(from: 0, to: progress)
                .stroke(
                    AngularGradient(colors: [.blue, .purple], center: .center),
                    style: StrokeStyle(lineWidth: lineWidth, lineCap: .round)
                )
                .rotationEffect(.degrees(-90))
                .animation(.easeInOut, value: progress)

            Text("\(Int(progress * 100))%")
                .font(.headline)
                .fontWeight(.semibold)
        }
    }
}

// Usage examples
struct CustomViewExamples: View {
    @State private var progress: Double = 0.0

    var body: some View {
        ScrollView {
            VStack(spacing: 20) {
                // Custom button
                PrimaryButton(title: "Custom Button") {
                    print("Button tapped!")
                }

                // Custom card view
                VStack {
                    Text("Card Content")
                        .font(.headline)
                    Text("This content uses the custom card modifier")
                        .font(.body)
                        .multilineTextAlignment(.center)
                }
                .cardStyle()

                // Custom shape
                CurvedRectangle(cornerRadius: 20, curveHeight: 30)
                    .fill(LinearGradient(colors: [.orange, .red], startPoint: .leading, endP
                    .frame(height: 100)

                // Custom progress view
                CircularProgressView(progress: progress)
                    .frame(width: 100, height: 100)

                Button("Update Progress") {
                    withAnimation {
                        progress = Double.random(in: 0...1)
                    }
                }
            }
            .padding()
        }
    }
}
```

Key Points:

- **Custom views encapsulate reusable UI components**
- **ViewModifiers provide reusable styling and behavior**
- **Custom shapes enable unique visual designs**
- **Extensions make custom modifiers easy to use**

Notes:

Custom views and modifiers promote code reuse and maintainable UI architecture in SwiftUI.

6.2 Gesture Handling

SwiftUI provides powerful gesture recognition for creating interactive user experiences.

Code Example:

```
import SwiftUI

struct GestureExamples: View {
    @State private var offset = CGSize.zero
    @State private var scale: CGFloat = 1.0
    @State private var rotation: Angle = .degrees(0)
    @State private var longPressCount = 0

    var body: some View {
        ScrollView {
            VStack(spacing: 40) {
                // Drag Gesture
                VStack {
                    Text("Drag Gesture")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 10)
                        .fill(Color.blue)
                        .frame(width: 100, height: 100)
                        .offset(offset)
                        .gesture(
                            DragGesture()
                                .onChanged { value in
                                    offset = value.translation
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        offset = .zero
                                    }
                                }
                        )
                }

                // Magnification Gesture
                VStack {
                    Text("Pinch to Scale")
                        .font(.headline)

                    Circle()
                        .fill(Color.green)
                        .frame(width: 80, height: 80)
                        .scaleEffect(scale)
                        .gesture(
                            MagnificationGesture()
                                .onChanged { value in
                                    scale = value
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        scale = 1.0
                                    }
                                }
                        )
                }

                // Rotation Gesture
                VStack {
                    Text("Rotation Gesture")
                        .font(.headline)

                    Rectangle()
                        .fill(Color.orange)
                        .frame(width: 100, height: 60)
                        .rotationEffect(rotation)
                        .gesture(
                            RotationGesture()
                                .onChanged { value in
                                    rotation = value
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        rotation = .degrees(0)
                                    }
                                }
                        )
                }

                // Long Press Gesture
                VStack {
                    Text("Long Press Count: \(longPressCount)")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 10)
                        .fill(Color.purple)
                        .frame(width: 120, height: 60)
                        .overlay(
                            Text("Long Press")
                                .foregroundColor(.white)
                                .font(.caption)
                        )
                        .onLongPressGesture(minimumDuration: 1.0) {
                            longPressCount += 1
                        }
                }

                // Combined Gestures
                VStack {
                    Text("Combined Gestures")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 15)
                        .fill(Color.red)
                        .frame(width: 100, height: 100)
                        .scaleEffect(scale)
                        .rotationEffect(rotation)
                        .offset(offset)
                        .gesture(
                            SimultaneousGesture(
                                DragGesture()
```

```
                                    .onChanged { value in
                                        offset = value.translation
                                    },
                                MagnificationGesture()
                                    .onChanged { value in
                                        scale = value
                                    }
                            )
                        )
                }

                Button("Reset All") {
                    withAnimation(.spring()) {
                        offset = .zero
                        scale = 1.0
                        rotation = .degrees(0)
                    }
                }
            }
            .padding()
        }
    }
}

// Custom gesture for swipe detection
struct SwipeGestureExample: View {
    @State private var swipeDirection: String = "None"

    var body: some View {
        VStack {
            Text("Swipe Direction: \(swipeDirection)")
                .font(.headline)
                .padding()

            Rectangle()
                .fill(Color.gray.opacity(0.3))
                .frame(width: 200, height: 200)
                .overlay(
                    Text("Swipe Me")
                        .font(.title)
                )
                .gesture(
                    DragGesture(minimumDistance: 50)
                        .onEnded { value in
                            let horizontalAmount = value.translation.x
                            let verticalAmount = value.translation.y

                            if abs(horizontalAmount) > abs(verticalAmount) {
                                swipeDirection = horizontalAmount < 0 ? "Left" : "Right"
                            } else {
                                swipeDirection = verticalAmount < 0 ? "Up" : "Down"
                            }
                        }
                )
        }
    }
}
```

Key Points:

- Drag, magnification, and rotation gestures provide intuitive interaction
- Long press gestures enable context-sensitive actions
- SimultaneousGesture combines multiple gestures
- Custom gesture logic enables app-specific interactions

Notes:

> *SwiftUI gestures make apps feel responsive and natural to use across all Apple platforms.*

6.3 Performance Optimization

Techniques for optimizing SwiftUI app performance and responsiveness.

Code Example:

```
import SwiftUI

// Lazy loading with LazyVStack and LazyHStack
struct LazyLoadingExample: View {
    let items = Array(1...10000)

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 8) {
                ForEach(items, id: \.self) { item in
                    ExpensiveView(number: item)
                        .onAppear {
                            // Only create view when it appears
                            print("View \(item) appeared")
                        }
                }
            }
            .padding()
        }
    }
}

// Expensive view that should be lazy loaded
struct ExpensiveView: View {
    let number: Int

    var body: some View {
        HStack {
            // Simulate expensive operation
            Text("Item #\(number)")
            Spacer()
            Text(String(repeating: "•", count: Int.random(in: 1...5)))
        }
        .padding()
        .background(Color.blue.opacity(0.1))
        .cornerRadius(8)
    }
}

// Using @State vs @StateObject properly
class ExpensiveDataModel: ObservableObject {
    @Published var data: [String] = []

    init() {
        // Expensive initialization
        loadData()
    }

    private func loadData() {
        // Simulate expensive data loading
        data = Array(1...1000).map { "Item \($0)" }
    }
}

struct OptimizedStateExample: View {
    // Use @StateObject for owned objects
    @StateObject private var dataModel = ExpensiveDataModel()

    // Use @State for simple values
    @State private var searchText = ""

    var filteredData: [String] {
        if searchText.isEmpty {
            return dataModel.data
        }
        return dataModel.data.filter { $0.localizedCaseInsensitiveContains(searchText) }
    }

    var body: some View {
        NavigationView {
            VStack {
                SearchBar(text: $searchText)

                List(filteredData, id: \.self) { item in
                    Text(item)
                }
            }
            .navigationTitle("Optimized List")
        }
    }
}

// Efficient list updates with identifiable data
struct IdentifiableDataExample: View {
    @State private var users: [User] = []

    var body: some View {
        List {
            ForEach(users) { user in
                UserRowView(user: user)
                    .id(user.id) // Explicit ID for efficient updates
            }
```

```swift
            .onDelete(perform: deleteUsers)
        }
        .onAppear {
            loadUsers()
        }
    }

    private func deleteUsers(at offsets: IndexSet) {
        users.remove(atOffsets: offsets)
    }

    private func loadUsers() {
        // Load users efficiently
        users = UserService.loadUsers()
    }
}

// Using PreferenceKey for efficient data passing up the view hierarchy
struct ScrollOffsetPreferenceKey: PreferenceKey {
    static var defaultValue: CGFloat = 0

    static func reduce(value: inout CGFloat, nextValue: () -> CGFloat) {
        value = nextValue()
    }
}

struct ScrollOffsetReader: View {
    @State private var scrollOffset: CGFloat = 0

    var body: some View {
        ScrollView {
            LazyVStack {
                ForEach(0..<50) { index in
                    Text("Row \(index)")
                        .frame(maxWidth: .infinity)
                        .padding()
                        .background(Color.gray.opacity(0.1))
                }
            }
            .background(
                GeometryReader { geometry in
                    Color.clear
                        .preference(key: ScrollOffsetPreferenceKey.self,
                                  value: geometry.frame(in: .named("scrollView")).minY)
                }
            )
        }
        .coordinateSpace(name: "scrollView")
        .onPreferenceChange(ScrollOffsetPreferenceKey.self) { value in
            scrollOffset = value
        }
        .overlay(
            Text("Offset: \(Int(scrollOffset))")
                .padding()
                .background(Color.black.opacity(0.7))
                .foregroundColor(.white)
                .cornerRadius(8)
                .padding(.top),
            alignment: .topTrailing
        )
    }
}

// Memory-efficient image loading
struct AsyncImageExample: View {
    let imageURL: URL

    var body: some View {
        AsyncImage(url: imageURL) { image in
            image
                .resizable()
                .aspectRatio(contentMode: .fill)
        } placeholder: {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.gray.opacity(0.3))
                .overlay(
                    ProgressView()
                        .scaleEffect(0.5)
                )
        }
        .frame(width: 150, height: 150)
        .clipped()
        .cornerRadius(10)
    }
}
```

Key Points:

• Use LazyVStack/LazyHStack for large lists to improve performance
• Choose @State vs @StateObject appropriately
• Provide explicit IDs for efficient list updates
• Use PreferenceKey for efficient upward data flow
• AsyncImage provides memory-efficient image loading

Notes:

Performance optimization in SwiftUI focuses on lazy loading, proper state management, and efficient data flow.

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())
    ]

    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
```

```swift
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.*

7.2 Combine Operators

Combine provides dozens of operators for transforming, filtering, and combining publisher streams.

Code Example:

```
import Combine
import Foundation

class CombineOperatorsExample: ObservableObject {
    var cancellables = Set<AnyCancellable>()

    func transformationOperators() {
        // Map - transform each element
        [1, 2, 3, 4, 5].publisher
            .map { $0 * 2 }
            .sink { print("Doubled: \($0)") }
            .store(in: &cancellables)

        // FlatMap - flatten nested publishers
        ["apple", "banana", "cherry"].publisher
            .flatMap { fruit in
                Just(fruit.uppercased())
                    .delay(for: .seconds(1), scheduler: RunLoop.main)
            }
            .sink { print("Uppercased: \($0)") }
            .store(in: &cancellables)

        // CompactMap - filter out nil values
        ["1", "2", "three", "4", "five"].publisher
            .compactMap { Int($0) }
            .sink { print("Valid number: \($0)") }
            .store(in: &cancellables)
        // Scan - accumulate values
        [1, 2, 3, 4, 5].publisher
            .scan(0, +)
            .sink { print("Running sum: \($0)") }
            .store(in: &cancellables)
    }

    func filteringOperators() {
        // Filter - include only matching elements
        (1...10).publisher
            .filter { $0 % 2 == 0 }
            .sink { print("Even number: \($0)") }
            .store(in: &cancellables)

        // RemoveDuplicates - filter consecutive duplicates
        [1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 4].publisher
            .removeDuplicates()
            .sink { print("Unique: \($0)") }
            .store(in: &cancellables)

        // DropFirst/DropLast - skip elements
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].publisher
            .dropFirst(3)
            .dropLast(2)
            .sink { print("Middle values: \($0)") }
            .store(in: &cancellables)

        // Prefix - take only first n elements
        (1...100).publisher
            .prefix(5)
            .sink { print("First 5: \($0)") }
            .store(in: &cancellables)
    }

    func combiningOperators() {
        let publisher1 = PassthroughSubject<String, Never>()
        let publisher2 = PassthroughSubject<String, Never>()

        // Merge - combine multiple publishers
        Publishers.Merge(publisher1, publisher2)
            .sink { print("Merged: \($0)") }
            .store(in: &cancellables)

        // CombineLatest - emit when any publisher emits
        Publishers.CombineLatest(publisher1, publisher2)
            .sink { value1, value2 in
                print("Combined: \(value1) + \(value2)")
            }
            .store(in: &cancellables)

        // Zip - pair corresponding elements
        let numbers = [1, 2, 3, 4, 5].publisher
        let letters = ["A", "B", "C", "D", "E"].publisher

        Publishers.Zip(numbers, letters)
            .sink { number, letter in
                print("Zipped: \(number)\(letter)")
            }
            .store(in: &cancellables)
    }

    func timingOperators() {
        // Debounce - wait for pause in emissions
        let searchSubject = PassthroughSubject<String, Never>()

        searchSubject
            .debounce(for: .milliseconds(500), scheduler: RunLoop.main)
            .removeDuplicates()
            .sink { searchTerm in
                print("Searching for: \(searchTerm)")
                // Perform search here
            }
            .store(in: &cancellables)

        // Throttle - limit emission frequency
        Timer.publish(every: 0.1, on: .main, in: .common)
            .autoconnect()
            .throttle(for: .seconds(1), scheduler: RunLoop.main, latest: true)
            .sink { date in
                print("Throttled timer: \(date)")
            }
            .store(in: &cancellables)

        // Delay - delay emissions
        ["Immediate", "Delayed"].publisher
            .delay(for: .seconds(2), scheduler: DispatchQueue.main)
            .sink { print("After delay: \($0)") }
            .store(in: &cancellables)

        // Timeout - fail if no emission within time limit
        Just("Hello")
            .delay(for: .seconds(3), scheduler: DispatchQueue.main)
            .timeout(.seconds(2), scheduler: DispatchQueue.main)
            .sink(
                receiveCompletion: { completion in
                    switch completion {
                    case .failure:
                        print("Timed out!")
                    case .finished:
                        print("Completed")
                    }
                },
                receiveValue: { print("Received: \($0)") }
            )
            .store(in: &cancellables)
    }

    func errorHandlingOperators() {
        enum NetworkError: Error {
            case connectionFailed
            case timeout
        }

        // Catch - handle errors and provide fallback
        Fail<String, NetworkError>(error: .connectionFailed)
            .catch { error in
                Just("Fallback value")
            }
            .sink { print("Result: \($0)") }
            .store(in: &cancellables)

        // Retry - retry failed operations
        let failingPublisher = PassthroughSubject<String, NetworkError>()

        failingPublisher
            .retry(3)
            .sink(
                receiveCompletion: { completion in
                    print("Final completion: \(completion)")
                },
                receiveValue: { print("Value: \($0)") }
            )
            .store(in: &cancellables)

        // ReplaceError - replace errors with a value
        Fail<String, NetworkError>(error: .timeout)
            .replaceError(with: "Default response")
            .sink { print("Error replaced with: \($0)") }
            .store(in: &cancellables)
    }
}
```

Key Points:

- **Map, flatMap, and compactMap transform publisher values**
- **Filter, removeDuplicates control which values pass through**
- **Merge, combineLatest, and zip combine multiple publishers**
- **Debounce, throttle, and delay control timing of emissions**
- **Catch, retry, and replaceError handle failure scenarios**

Notes:

> *Combine operators provide a declarative way to process asynchronous data streams with powerful composition capabilities.*

7.3 Networking with Combine

Combine integrates seamlessly with URLSession for reactive networking and data processing.

Code Example:

```
import Combine
import Foundation

// Network service using Combine
class NetworkService: ObservableObject {
    @Published var isLoading = false
    @Published var users: [User] = []
    @Published var errorMessage: String?

    private var cancellables = Set<AnyCancellable>()

    // Generic API request method
    func request<T: Codable>(url: URL, type: T.Type) -> AnyPublisher<T, NetworkError> {
        URLSession.shared.dataTaskPublisher(for: url)
            .tryMap { data, response -> Data in
                guard let httpResponse = response as? HTTPURLResponse,
                      200...299 ~= httpResponse.statusCode else {
                    throw NetworkError.invalidResponse
                }
                return data
            }
            .decode(type: type, decoder: JSONDecoder())
            .mapError { error in
                if error is DecodingError {
                    return NetworkError.decodingFailed
                }
                return NetworkError.networkFailed
            }
            .receive(on: DispatchQueue.main)
            .eraseToAnyPublisher()
    }

    // Fetch users with error handling
    func fetchUsers() {
        guard let url = URL(string: "https://jsonplaceholder.typicode.com/users") else {
            errorMessage = "Invalid URL"
            return
        }

        isLoading = true
        errorMessage = nil

        request(url: url, type: [User].self)
            .sink(
                receiveCompletion: { [weak self] completion in
                    self?.isLoading = false
                    switch completion {
                    case .failure(let error):
                        self?.errorMessage = error.localizedDescription
                    case .finished:
                        break
                    }
                },
                receiveValue: { [weak self] users in
                    self?.users = users
                }
            )
            .store(in: &cancellables)
    }

    // Search with debouncing
    func searchUsers(query: String) -> AnyPublisher<[User], NetworkError> {
        guard let url = URL(string: "https://jsonplaceholder.typicode.com/users") else {
            return Fail(error: NetworkError.invalidURL)
                .eraseToAnyPublisher()
        }

        return Just(query)
            .debounce(for: .milliseconds(300), scheduler: RunLoop.main)
            .removeDuplicates()
            .flatMap { searchTerm in
                self.request(url: url, type: [User].self)
                    .map { users in
                        users.filter { user in
                            user.name.localizedCaseInsensitiveContains(searchTerm)
                        }
                    }
            }
            .eraseToAnyPublisher()
    }

    // Upload with progress tracking
    func uploadFile(data: Data, to url: URL) -> AnyPublisher<UploadResponse, NetworkError> {
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.httpBody = data

        return URLSession.shared.dataTaskPublisher(for: request)
            .tryMap { data, response -> Data in
                guard let httpResponse = response as? HTTPURLResponse,
                      200...299 ~= httpResponse.statusCode else {
                    throw NetworkError.serverError
                }
                return data
            }
            .decode(type: UploadResponse.self, decoder: JSONDecoder())
            .mapError { _ in NetworkError.uploadFailed }
            .receive(on: DispatchQueue.main)
            .eraseToAnyPublisher()
    }

    // Batch requests with error recovery
    func fetchMultipleResources() -> AnyPublisher<CombinedData, Never> {
        let usersPublisher = request(url: URL(string: "https://api.example.com/users")!, typ
            .catch { _ in Just([User]()) } // Fallback to empty array on error

        let postsPublisher = request(url: URL(string: "https://api.example.com/posts")!, typ
            .catch { _ in Just([Post]()) } // Fallback to empty array on error

        return Publishers.CombineLatest(usersPublisher, postsPublisher)
            .map { users, posts in
                CombinedData(users: users, posts: posts)
            }
            .eraseToAnyPublisher()
    }
}

// Error types for networking
enum NetworkError: Error, LocalizedError {
    case invalidURL
    case networkFailed
    case invalidResponse
    case decodingFailed
    case serverError
    case uploadFailed

    var errorDescription: String? {
        switch self {
        case .invalidURL:
            return "Invalid URL"
        case .networkFailed:
            return "Network request failed"
        case .invalidResponse:
            return "Invalid server response"
        case .decodingFailed:
            return "Failed to decode response"
        case .serverError:
            return "Server error occurred"
        case .uploadFailed:
            return "Upload failed"
        }
    }
}

// Usage in SwiftUI
struct NetworkingExampleView: View {
    @StateObject private var networkService = NetworkService()
    @State private var searchText = ""
    @State private var searchResults: [User] = []

    var body: some View {
        NavigationView {
            VStack {
                SearchBar(text: $searchText)
                    .onChange(of: searchText) { newValue in
                        searchUsers(query: newValue)
                    }

                if networkService.isLoading {
                    ProgressView("Loading...")
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                } else if let error = networkService.errorMessage {
                    Text("Error: \(error)")
                        .foregroundColor(.red)
                        .padding()
                } else {
                    List(searchResults.isEmpty ? networkService.users : searchResults) { use
                        VStack(alignment: .leading) {
                            Text(user.name)
                                .font(.headline)
                            Text(user.email)
                                .font(.caption)
                                .foregroundColor(.secondary)
                        }
                    }
                }
            }
            .navigationTitle("Users")
            .onAppear {
                networkService.fetchUsers()
            }
        }
    }

    private func searchUsers(query: String) {
        networkService.searchUsers(query: query)
            .sink(
                receiveCompletion: { _ in },
                receiveValue: { users in
                    searchResults = users
                }
            )
            .store(in: &networkService.cancellables)
    }
}
```

Key Points:

- **URLSession dataTaskPublisher integrates with Combine**
- **Use tryMap for response validation and error handling**
- **Debounce search queries to reduce API calls**
- **Combine multiple API calls with CombineLatest or Zip**
- **Handle errors gracefully with catch and fallback values**

Notes:

Combine transforms networking from callback-based to reactive, making complex data flows more manageable.

PART VI: iOS DEVELOPMENT

Chapter 9: Data Persistence

9.1 UserDefaults and AppStorage

UserDefaults provides simple key-value storage for user preferences and app settings.

Code Example:

```
import SwiftUI
import Foundation

// UserDefaults wrapper for type safety
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    var wrappedValue: T {
        get {
            UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}

// Settings manager using UserDefaults
class AppSettings: ObservableObject {
    @UserDefault(key: "username", defaultValue: "")
    var username: String

    @UserDefault(key: "isDarkMode", defaultValue: false)
    var isDarkMode: Bool

    @UserDefault(key: "fontSize", defaultValue: 16.0)
    var fontSize: Double

    @UserDefault(key: "notifications", defaultValue: true)
    var notificationsEnabled: Bool

    @UserDefault(key: "lastLoginDate", defaultValue: Date.distantPast)
    var lastLoginDate: Date

    // Complex data storage with Codable
    @UserDefault(key: "favoriteItems", defaultValue: [])
    var favoriteItems: [String]

    // Store custom objects
    var userProfile: UserProfile? {
        get {
            guard let data = UserDefaults.standard.data(forKey: "userProfile") else { return
            return try? JSONDecoder().decode(UserProfile.self, from: data)
        }
        set {
            let data = try? JSONEncoder().encode(newValue)
            UserDefaults.standard.set(data, forKey: "userProfile")
        }
    }

    func resetToDefaults() {
        username = ""
        isDarkMode = false
        fontSize = 16.0
        notificationsEnabled = true
        lastLoginDate = Date.distantPast
        favoriteItems = []
        userProfile = nil
    }
}

// SwiftUI integration with @AppStorage
struct SettingsView: View {
    @AppStorage("username") private var username: String = ""
    @AppStorage("isDarkMode") private var isDarkMode: Bool = false
    @AppStorage("fontSize") private var fontSize: Double = 16.0
    @AppStorage("theme") private var selectedTheme: Theme = .system

    var body: some View {
        NavigationView {
            Form {
                Section("Account") {
                    TextField("Username", text: $username)
                        .textContentType(.username)

                    Toggle("Enable Notifications", isOn: .constant(true))
                }

                Section("Appearance") {
                    Toggle("Dark Mode", isOn: $isDarkMode)

                    HStack {
                        Text("Font Size")
                        Spacer()
                        Text("\(Int(fontSize))pt")
                            .foregroundColor(.secondary)
                    }

                    Slider(value: $fontSize, in: 12...24, step: 1)

                    Picker("Theme", selection: $selectedTheme) {
                        ForEach(Theme.allCases, id: \.self) { theme in
                            Text(theme.displayName).tag(theme)
                        }
                    }
                }

                Section("Data") {
                    Button("Reset Settings", role: .destructive) {
                        resetSettings()
                    }

                    Button("Export Settings") {
                        exportSettings()
                    }
                }
            }
            .navigationTitle("Settings")
        }
        .preferredColorScheme(isDarkMode ? .dark : .light)
    }

    private func resetSettings() {
        username = ""
        isDarkMode = false
        fontSize = 16.0
        selectedTheme = .system
    }

    private func exportSettings() {
        let settings = [
            "username": username,
            "isDarkMode": isDarkMode,
            "fontSize": fontSize,
            "theme": selectedTheme.rawValue
        ]

        // Export logic here
        print("Exported settings: \(settings)")
    }
}

// Theme enum for UserDefaults
enum Theme: String, CaseIterable, Codable {
    case system = "system"
    case light = "light"
    case dark = "dark"

    var displayName: String {
        switch self {
        case .system: return "System"
        case .light: return "Light"
        case .dark: return "Dark"
        }
    }
}

// User profile model
struct UserProfile: Codable {
    let id: String
    let name: String
    let email: String
    let avatar: URL?
    let preferences: [String: String]
    static let example = UserProfile(
        id: UUID().uuidString,
        name: "John Doe",
        email: "john@example.com",
        avatar: URL(string: "https://example.com/avatar.jpg"),
        preferences: ["theme": "dark", "language": "en"]
    )
}

// Advanced UserDefaults usage
extension UserDefaults {
    func set<T: Codable>(_ object: T, forKey key: String) {
        let data = try? JSONEncoder().encode(object)
        set(data, forKey: key)
    }

    func get<T: Codable>(_ type: T.Type, forKey key: String) -> T? {
        guard let data = data(forKey: key) else { return nil }
        return try? JSONDecoder().decode(type, from: data)
    }

    func remove(forKey key: String) {
        removeObject(forKey: key)
    }
}
```

Key Points:

• UserDefaults provides simple persistent storage for app settings
• @AppStorage automatically syncs SwiftUI views with UserDefaults
• Property wrappers make UserDefaults type-safe and easy to use
• Store complex objects using Codable and JSON encoding
• Group related settings in dedicated classes for organization

Notes:

UserDefaults is perfect for storing user preferences, app settings, and simple persistent data in iOS apps.

9.2 Core Data Basics

Core Data provides object graph management and persistence for complex data models in iOS applications.

Code Example:

```
import CoreData
import SwiftUI

// Core Data Stack
class CoreDataManager: ObservableObject {
    static let shared = CoreDataManager()
    lazy var persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "DataModel")
        container.loadPersistentStores { _, error in
            if let error = error {
                fatalError("Core Data error: \(error)")
            }
        }
        return container
    }()

    var context: NSManagedObjectContext {
        persistentContainer.viewContext
    }

    func save() {
        let context = persistentContainer.viewContext

        if context.hasChanges {
            do {
                try context.save()
            } catch {
                print("Save error: \(error)")
            }
        }
    }
}

// Core Data Entity (User+CoreDataClass.swift)
@objc(User)
public class User: NSManagedObject {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<User> {
        return NSFetchRequest<User>(entityName: "User")
    }

    @NSManaged public var id: UUID
    @NSManaged public var name: String
    @NSManaged public var email: String
    @NSManaged public var createdDate: Date
    @NSManaged public var posts: NSSet?

    // Computed properties
    public var postsArray: [Post] {
        let set = posts as? Set<Post> ?? []
        return set.sorted { $0.createdDate < $1.createdDate }
    }

    // Convenience initializer
    convenience init(context: NSManagedObjectContext, name: String, email: String) {
        self.init(context: context)
        self.id = UUID()
        self.name = name
        self.email = email
        self.createdDate = Date()
    }
}

// Repository pattern for Core Data operations
```

```
class UserRepository: ObservableObject {
    private let coreDataManager = CoreDataManager.shared
    @Published var users: [User] = []
    @Published var isLoading = false

    init() {
        fetchUsers()
    }

    func fetchUsers() {
        isLoading = true
        let request: NSFetchRequest<User> = User.fetchRequest()
        request.sortDescriptors = [NSSortDescriptor(keyPath: \User.name, ascending: true)]

        do {
            users = try coreDataManager.context.fetch(request)
        } catch {
            print("Fetch error: \(error)")
        }
        isLoading = false
    }

    func addUser(name: String, email: String) {
        let user = User(context: coreDataManager.context, name: name, email: email)
        coreDataManager.save()
        fetchUsers()
    }

    func deleteUser(_ user: User) {
        coreDataManager.context.delete(user)
        coreDataManager.save()
        fetchUsers()
    }

    func updateUser(_ user: User, name: String, email: String) {
        user.name = name
        user.email = email
        coreDataManager.save()
        fetchUsers()
    }

    func searchUsers(by name: String) -> [User] {
        let request: NSFetchRequest<User> = User.fetchRequest()
        request.predicate = NSPredicate(format: "name CONTAINS[cd] %@", name)
        request.sortDescriptors = [NSSortDescriptor(keyPath: \User.name, ascending: true)]

        do {
            return try coreDataManager.context.fetch(request)
        } catch {
            print("Search error: \(error)")
            return []
        }
    }
}

// SwiftUI integration with Core Data
struct UserListView: View {
    @StateObject private var repository = UserRepository()
    @State private var showingAddUser = false

    var body: some View {
        NavigationView {
            List {
                if repository.isLoading {
                    ProgressView("Loading users...")
                        .frame(maxWidth: .infinity)
                } else {
                    ForEach(repository.users, id: \.id) { user in
                        VStack(alignment: .leading) {
                            Text(user.name)
                                .font(.headline)
                            Text(user.email)
                                .font(.caption)
                                .foregroundColor(.secondary)
                            Text("Created: \(user.createdDate, style: .date)")
                                .font(.caption2)
                                .foregroundColor(.secondary)
                        }
                    }
                    .onDelete(perform: deleteUsers)
                }
            }
            .navigationTitle("Users")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        showingAddUser = true
                    }
                }
            }
            .sheet(isPresented: $showingAddUser) {
                AddUserView { name, email in
                    repository.addUser(name: name, email: email)
                }
            }
        }
    }

    private func deleteUsers(offsets: IndexSet) {
        for index in offsets {
            let user = repository.users[index]
            repository.deleteUser(user)
        }
    }
}

// Add user form
struct AddUserView: View {
    @Environment(\.dismiss) private var dismiss
    @State private var name = ""
    @State private var email = ""

    let onSave: (String, String) -> Void
    var body: some View {
        NavigationView {
            Form {
                TextField("Name", text: $name)
                TextField("Email", text: $email)
                    .textContentType(.emailAddress)
                    .keyboardType(.emailAddress)
            }
            .navigationTitle("Add User")
            .toolbar {
                ToolbarItem(placement: .navigationBarLeading) {
                    Button("Cancel") {
                        dismiss()
                    }
                }

                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Save") {
                        onSave(name, email)
                        dismiss()
                    }
                    .disabled(name.isEmpty || email.isEmpty)
                }
            }
        }
    }
}
```

Key Points:

- **Core Data provides object graph management and persistence**
- **Use NSPersistentContainer to set up the Core Data stack**
- **Repository pattern encapsulates Core Data operations**
- **NSFetchRequest with predicates enables complex queries**
- **SwiftUI integrates seamlessly with Core Data through ObservableObject**

Notes:

Core Data is ideal for complex data models with relationships and advanced querying capabilities.

Chapter 10: Testing & Architecture

10.1 Unit Testing

Unit testing ensures code reliability and maintainability through automated testing of individual components.

Code Example:

```
import XCTest
@testable import MyApp

// Test class structure
class CalculatorTests: XCTestCase {
    var calculator: Calculator!

    override func setUpWithError() throws {
        // Set up test objects before each test
        calculator = Calculator()
    }

    override func tearDownWithError() throws {
        // Clean up after each test
        calculator = nil
    }

    // Basic test methods
    func testAddition() {
        let result = calculator.add(5, 3)
        XCTAssertEqual(result, 8, "5 + 3 should equal 8")
    }

    func testSubtraction() {
        let result = calculator.subtract(10, 4)
        XCTAssertEqual(result, 6)
    }

    func testDivision() {
        let result = calculator.divide(12, 3)
        XCTAssertEqual(result, 4)
    }

    func testDivisionByZero() {
        XCTAssertThrowsError(try calculator.divide(10, 0)) { error in
            XCTAssertEqual(error as? CalculatorError, CalculatorError.divisionByZero)
        }
    }
}

// Testing asynchronous code
class NetworkServiceTests: XCTestCase {
    var networkService: NetworkService!

    override func setUp() {
        networkService = NetworkService()
    }

    func testFetchUsers() async throws {
        let users = try await networkService.fetchUsers()
        XCTAssertGreaterThan(users.count, 0)
        XCTAssertNotNil(users.first?.name)
    }

    func testFetchUsersWithExpectation() {
        let expectation = XCTestExpectation(description: "Fetch users")

        networkService.fetchUsers { result in
            switch result {
            case .success(let users):
                XCTAssertGreaterThan(users.count, 0)
            case .failure(let error):
                XCTFail("Failed with error: \(error)")
            }
            expectation.fulfill()
        }

        wait(for: [expectation], timeout: 10.0)
    }
}

// Testing with mocks
protocol UserRepositoryProtocol {
    func fetchUsers() async throws -> [User]
    func saveUser(_ user: User) async throws
}

class MockUserRepository: UserRepositoryProtocol {
    var shouldThrowError = false
    var mockUsers: [User] = []

    func fetchUsers() async throws -> [User] {
        if shouldThrowError {
            throw NetworkError.connectionFailed
        }
        return mockUsers
    }

    func saveUser(_ user: User) async throws {
        if shouldThrowError {
            throw DatabaseError.saveFailed
        }
        mockUsers.append(user)
    }
}

class UserViewModelTests: XCTestCase {
    var viewModel: UserViewModel!
    var mockRepository: MockUserRepository!
    override func setUp() {
        mockRepository = MockUserRepository()
        viewModel = UserViewModel(repository: mockRepository)
    }

    func testLoadUsersSuccess() async {
        // Arrange
        let expectedUsers = [User(name: "John", email: "john@test.com")]
        mockRepository.mockUsers = expectedUsers

        // Act
        await viewModel.loadUsers()

        // Assert
        XCTAssertEqual(viewModel.users.count, 1)
        XCTAssertEqual(viewModel.users.first?.name, "John")
        XCTAssertFalse(viewModel.isLoading)
        XCTAssertNil(viewModel.errorMessage)
    }

    func testLoadUsersFailure() async {
        // Arrange
        mockRepository.shouldThrowError = true

        // Act
        await viewModel.loadUsers()

        // Assert
        XCTAssertTrue(viewModel.users.isEmpty)
        XCTAssertNotNil(viewModel.errorMessage)
        XCTAssertFalse(viewModel.isLoading)
    }
}

// Performance testing
class PerformanceTests: XCTestCase {
    func testSortingPerformance() {
        let largeArray = (1...10000).shuffled()

        measure {
            _ = largeArray.sorted()
        }
    }

    func testAsyncPerformance() async {
        await measure {
            await performExpensiveAsyncOperation()
        }
    }
}

// Test utilities
extension XCTestCase {
    func waitForAsync<T>(
        _ asyncFunction: @escaping () async throws -> T,
        timeout: TimeInterval = 5.0
    ) async throws -> T {
        return try await withCheckedThrowingContinuation { continuation in
            Task {
                do {
                    let result = try await asyncFunction()
                    continuation.resume(returning: result)
                } catch {
                    continuation.resume(throwing: error)
                }
            }
        }
    }
}
```

Key Points:

• XCTestCase provides the foundation for unit testing
• Use setUp/tearDown for test preparation and cleanup
• Mock objects isolate units under test
• XCTestExpectation handles asynchronous testing
• Performance tests measure code efficiency

Notes:

Unit testing is essential for maintaining code quality and catching regressions early in development.

10.2 MVVM Architecture

Model-View-ViewModel (MVVM) architecture separates concerns and makes SwiftUI apps more testable and maintainable.

Code Example:

```
import SwiftUI
import Combine

// MARK: - Model
struct User: Identifiable, Codable, Equatable {
    let id: UUID
    var name: String
    var email: String
    var avatar: URL?
    var isActive: Bool

    init(name: String, email: String, avatar: URL? = nil, isActive: Bool = true) {
        self.id = UUID()
        self.name = name
        self.email = email
        self.avatar = avatar
        self.isActive = isActive
    }
}
```

```swift
// MARK: - Service Layer
protocol UserServiceProtocol {
    func fetchUsers() async throws -> [User]
    func createUser(_ user: User) async throws -> User
    func updateUser(_ user: User) async throws -> User
    func deleteUser(id: UUID) async throws
}

class UserService: UserServiceProtocol {
    func fetchUsers() async throws -> [User] {
        // Simulate network request
        try await Task.sleep(nanoseconds: 1_000_000_000)
        return [
            User(name: "John Doe", email: "john@example.com"),
            User(name: "Jane Smith", email: "jane@example.com"),
            User(name: "Bob Johnson", email: "bob@example.com")
        ]
    }

    func createUser(_ user: User) async throws -> User {
        // Simulate API call
        try await Task.sleep(nanoseconds: 500_000_000)
        return user
    }

    func updateUser(_ user: User) async throws -> User {
        try await Task.sleep(nanoseconds: 500_000_000)
        return user
    }

    func deleteUser(id: UUID) async throws {
        try await Task.sleep(nanoseconds: 500_000_000)
    }
}

// MARK: - ViewModel
class UserListViewModel: ObservableObject {
    @Published var users: [User] = []
    @Published var isLoading = false
    @Published var errorMessage: String?
    @Published var searchText = ""

    private let userService: UserServiceProtocol
    private var cancellables = Set<AnyCancellable>()

    // Computed properties
    var filteredUsers: [User] {
        if searchText.isEmpty {
            return users
        }
        return users.filter { user in
            user.name.localizedCaseInsensitiveContains(searchText) ||
            user.email.localizedCaseInsensitiveContains(searchText)
        }
    }

    var activeUsersCount: Int {
        users.filter { $0.isActive }.count
    }

    init(userService: UserServiceProtocol = UserService()) {
        self.userService = userService
        setupSearchDebouncing()
    }

    private func setupSearchDebouncing() {
        $searchText
            .debounce(for: .milliseconds(300), scheduler: RunLoop.main)
            .removeDuplicates()
            .sink { [weak self] _ in
                self?.objectWillChange.send()
            }
            .store(in: &cancellables)
    }

    // MARK: - User Actions
    @MainActor
    func loadUsers() async {
        isLoading = true
        errorMessage = nil

        do {
            users = try await userService.fetchUsers()
        } catch {
            errorMessage = "Failed to load users: \(error.localizedDescription)"
        }

        isLoading = false
    }

    @MainActor
    func addUser(name: String, email: String) async {
        let newUser = User(name: name, email: email)

        do {
            let createdUser = try await userService.createUser(newUser)
            users.append(createdUser)
        } catch {
            errorMessage = "Failed to add user: \(error.localizedDescription)"
        }
    }

    @MainActor
    func updateUser(_ user: User) async {
        do {
            let updatedUser = try await userService.updateUser(user)
            if let index = users.firstIndex(where: { $0.id == updatedUser.id }) {
                users[index] = updatedUser
            }
        } catch {
            errorMessage = "Failed to update user: \(error.localizedDescription)"
        }
    }
    @MainActor
    func deleteUser(_ user: User) async {
        do {
            try await userService.deleteUser(id: user.id)
            users.removeAll { $0.id == user.id }
        } catch {
            errorMessage = "Failed to delete user: \(error.localizedDescription)"
        }
    }

    func toggleUserActive(_ user: User) {
        if let index = users.firstIndex(where: { $0.id == user.id }) {
            users[index].isActive.toggle()
        }
    }

    func clearError() {
        errorMessage = nil
    }
}

// MARK: - View
struct UserListView: View {
    @StateObject private var viewModel = UserListViewModel()
    @State private var showingAddUser = false

    var body: some View {
        NavigationView {
            VStack {
                SearchBar(text: $viewModel.searchText)

                UserStatsView(
                    totalUsers: viewModel.users.count,
                    activeUsers: viewModel.activeUsersCount
                )

                if viewModel.isLoading {
                    ProgressView("Loading users...")
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                } else {
                    UserList(
                        users: viewModel.filteredUsers,
                        onToggleActive: { user in
                            viewModel.toggleUserActive(user)
                        },
                        onDelete: { user in
                            Task {
                                await viewModel.deleteUser(user)
                            }
                        }
                    )
                }
            }
            .navigationTitle("Users")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        showingAddUser = true
                    }
                }
            }
            .sheet(isPresented: $showingAddUser) {
                AddUserView { name, email in
                    Task {
                        await viewModel.addUser(name: name, email: email)
                    }
                }
            }
            .alert("Error", isPresented: .constant(viewModel.errorMessage != nil)) {
                Button("OK") {
                    viewModel.clearError()
                }
            } message: {
                Text(viewModel.errorMessage ?? "")
            }
        }
        .task {
            await viewModel.loadUsers()
        }
    }
}

// MARK: - Supporting Views
struct UserStatsView: View {
    let totalUsers: Int
    let activeUsers: Int

    var body: some View {
        HStack {
            StatCard(title: "Total", value: totalUsers, color: .blue)
            StatCard(title: "Active", value: activeUsers, color: .green)
        }
        .padding()
    }
}

struct StatCard: View {
    let title: String
    let value: Int
    let color: Color

    var body: some View {
        VStack {
            Text("\(value)")
                .font(.largeTitle)
                .fontWeight(.bold)
                .foregroundColor(color)

            Text(title)
                .font(.caption)
                .foregroundColor(.secondary)
        }
        .frame(maxWidth: .infinity)
        .padding()
```

```
            .background(Color(.systemGray6))
            .cornerRadius(10)
        }
    }
```

Key Points:

• MVVM separates presentation logic from view code
• ViewModels handle business logic and state management
• ObservableObject enables reactive UI updates
• Dependency injection makes code more testable
• Service layer abstracts data access logic

Notes:

MVVM architecture makes SwiftUI apps more maintainable, testable, and follows separation of concerns principles.

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 (100+ PROBLEMS)

This comprehensive appendix contains 100+ carefully selected Data Structures and Algorithms problems with complete Swift solutions. Each problem includes detailed problem statement, optimized Swift implementation, complexity analysis, and algorithmic explanation. The problems are organized by topic and difficulty to facilitate systematic learning and technical interview preparation.

A.1 Array Problems (20 Problems)

A.1.1 Two Sum ■

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

Code Example:

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

    for (index, num) in nums.enumerated() {
        let complement = target - num
        if let complementIndex = hashMap[complement] {
            return [complementIndex, index]
        }
        hashMap[num] = index
    }
    return []
}

// Test case
let nums = [2, 7, 11, 15], target = 9
print(twoSum(nums, target)) // [0, 1]
```

Key Points:

• Time: O(n)
• Space: O(n)
• Hash table approach

Notes:

Use hash map to store numbers and their indices for O(1) complement lookup.

A.1.2 Best Time to Buy and Sell Stock ■

Find the maximum profit from buying and selling stock once.

Code Example:

```
func maxProfit(_ prices: [Int]) -> Int {
    guard !prices.isEmpty else { return 0 }
    var minPrice = prices[0]
    var maxProfit = 0
    for price in prices {
        if price < minPrice {
            minPrice = price
        } else if price - minPrice > maxProfit {
            maxProfit = price - minPrice
        }
    }
    return maxProfit
}

// Test case
let prices = [7, 1, 5, 3, 6, 4]
print(maxProfit(prices)) // 5
```

Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Single pass solution**

Notes:

Track minimum price and calculate maximum profit at each step.

A.1.3 Contains Duplicate ■

Return true if any value appears at least twice in the array.

Code Example:

```
func containsDuplicate(_ nums: [Int]) -> Bool {
    return Set(nums).count != nums.count
}

// Alternative approach with early exit
func containsDuplicateOptimal(_ nums: [Int]) -> Bool {
    var seen: Set<Int> = []
    for num in nums {
        if seen.contains(num) { return true }
        seen.insert(num)
    }
    return false
}
```

Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Set for duplicate detection**

Notes:

Use Set to detect duplicates efficiently - either compare counts or track seen elements.

A.1.4 Product of Array Except Self ■■

Return array where each element is the product of all other elements.

Code Example:

```
func productExceptSelf(_ nums: [Int]) -> [Int] {
    let n = nums.count
    var result = Array(repeating: 1, count: n)

    // Left products
    for i in 1..<n {
        result[i] = result[i-1] * nums[i-1]
    }

    // Right products
    var right = 1
    for i in stride(from: n-1, through: 0, by: -1) {
        result[i] *= right
        right *= nums[i]
    }

    return result
}
```

Key Points:

• Time: O(n)
• Space: O(1)
• Two-pass solution

Notes:

Calculate left products, then multiply by right products in reverse pass.

A.1.5 Maximum Subarray (Kadane's) ■■

Find the contiguous subarray with the largest sum.

Code Example:

```
func maxSubArray(_ nums: [Int]) -> Int {
    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
}

// Return actual subarray
func maxSubArrayWithIndices(_ nums: [Int]) -> [Int] {
    var maxSum = nums[0]
    var currentSum = nums[0]
    var start = 0, end = 0, tempStart = 0

    for i in 1..<nums.count {
        if nums[i] > currentSum + nums[i] {
            currentSum = nums[i]
            tempStart = i
        } else {
            currentSum += nums[i]
        }

        if currentSum > maxSum {
            maxSum = currentSum
            start = tempStart
            end = i
        }
    }

    return Array(nums[start...end])
}
```

Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Kadane's algorithm**

Notes:

At each position, decide whether to extend current subarray or start new one.

A.1.6 Maximum Product Subarray ■■

Find the contiguous subarray with the largest product.

Code Example:

```
func maxProduct(_ nums: [Int]) -> Int {
    var maxProd = nums[0]
    var minProd = nums[0]
    var result = nums[0]

    for i in 1..<nums.count {
        let temp = maxProd
        maxProd = max(nums[i], max(maxProd * nums[i], minProd * nums[i]))
        minProd = min(nums[i], min(temp * nums[i], minProd * nums[i]))
        result = max(result, maxProd)
    }
    return result
}

// Test case
let nums = [2, 3, -2, 4]
print(maxProduct(nums)) // 6
```

Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Track both max and min**

Notes:

Track both maximum and minimum products due to negative numbers.

A.1.7 Find Min in Rotated Sorted Array ■■

Find the minimum element in a rotated sorted array.

Code Example:

```
func findMin(_ nums: [Int]) -> Int {
    var left = 0
    var right = nums.count - 1

    while left < right {
        let mid = left + (right - left) / 2

        if nums[mid] > nums[right] {
            left = mid + 1
        } else {
            right = mid
        }
    }

    return nums[left]
}

// Test case
let nums = [3, 4, 5, 1, 2]
print(findMin(nums)) // 1
```

Key Points:

- **Time: O(log n)**
- **Space: O(1)**
- **Binary search**

Notes:

Use binary search to find the inflection point where rotation occurred.

A.1.8 Search in Rotated Sorted Array ■■

Search for a target in a rotated sorted array.

Code Example:

```
func search(_ nums: [Int], _ target: Int) -> Int {
    var left = 0
    var right = nums.count - 1

    while left <= right {
        let mid = left + (right - left) / 2

        if nums[mid] == target { return mid }

        if nums[left] <= nums[mid] {
            if nums[left] <= target && target < nums[mid] {
                right = mid - 1
            } else {
                left = mid + 1
            }
        } else {
            if nums[mid] < target && target <= nums[right] {
                left = mid + 1
            } else {
                right = mid - 1
            }
        }
    }

    return -1
}

// Test case
let nums = [4, 5, 6, 7, 0, 1, 2], target = 0
print(search(nums, target)) // 4
```

Key Points:

- **Time: O(log n)**
- **Space: O(1)**
- **Modified binary search**

Notes:

Determine which half is sorted, then decide which half to search.

A.1.9 3Sum ■■

Find all unique triplets that sum to zero.

Code Example:

```
func threeSum(_ nums: [Int]) -> [[Int]] {
    let sorted = nums.sorted()
    var result: [[Int]] = []

    for i in 0..<sorted.count - 2 {
        if i > 0 && sorted[i] == sorted[i-1] { continue }

        var left = i + 1
        var right = sorted.count - 1

        while left < right {
            let sum = sorted[i] + sorted[left] + sorted[right]

            if sum == 0 {
                result.append([sorted[i], sorted[left], sorted[right]])

                while left < right && sorted[left] == sorted[left + 1] {
                    left += 1
                }
                while left < right && sorted[right] == sorted[right - 1] {
                    right -= 1
                }

                left += 1
                right -= 1
            } else if sum < 0 {
                left += 1
            } else {
                right -= 1
            }
        }
    }

    return result
}

// Test case
let nums = [-1, 0, 1, 2, -1, -4]
print(threeSum(nums)) // [[-1, -1, 2], [-1, 0, 1]]
```

Key Points:

- **Time: $O(n^2)$**
- **Space: O(1)**
- **Two pointers technique**

Notes:

Sort array, then use two pointers for each fixed element to find triplets.

A.1.10 Container With Most Water ■■

Find two lines that form a container holding the most water.

Code Example:

```
func maxArea(_ height: [Int]) -> Int {
    var left = 0
    var right = height.count - 1
    var maxWater = 0

    while left < right {
        let water = min(height[left], height[right]) * (right - left)
        maxWater = max(maxWater, water)

        if height[left] < height[right] {
            left += 1
        } else {
            right -= 1
        }
    }

    return maxWater
}

// Test case
let height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
print(maxArea(height)) // 49
```

Key Points:

• Time: O(n)
• Space: O(1)
• Two pointers approach

Notes:

Use two pointers moving inward, always moving the pointer with smaller height.

A.1.11 Trapping Rain Water ■■■

Calculate how much water can be trapped after it rains.

Code Example:

```
func trap(_ height: [Int]) -> Int {
    guard height.count > 2 else { return 0 }

    var left = 0, right = height.count - 1
    var leftMax = 0, rightMax = 0
    var water = 0

    while left < right {
        if height[left] < height[right] {
            if height[left] >= leftMax {
                leftMax = height[left]
            } else {
                water += leftMax - height[left]
            }
            left += 1
        } else {
            if height[right] >= rightMax {
                rightMax = height[right]
            } else {
                water += rightMax - height[right]
            }
            right -= 1
        }
    }

    return water
}
```

Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Two pointers with max tracking**

Notes:

Use two pointers tracking maximum heights from both ends.

A.1.12 Merge Intervals ■■

Merge overlapping intervals.

Code Example:

```
func merge(_ intervals: [[Int]]) -> [[Int]] {
    guard !intervals.isEmpty else { return [] }

    let sorted = intervals.sorted { $0[0] < $1[0] }
    var result: [[Int]] = [sorted[0]]

    for i in 1..<sorted.count {
        let current = sorted[i]
        var last = result[result.count - 1]
        if current[0] <= last[1] {
            last[1] = max(last[1], current[1])
            result[result.count - 1] = last
        } else {
            result.append(current)
        }
    }

    return result
}
```

Key Points:

- **Time: O(n log n)**
- **Space: O(1)**
- **Sort and merge**

Notes:

Sort intervals by start time, then merge overlapping ones.

A.1.13 Insert Interval ■■

Insert into sorted intervals

Code Example:

```
func insert(_ intervals: [[Int]], _ newInterval: [Int]) -> [[Int]] { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.14 Rotate Array ■

Rotate array right by k steps

Code Example:

```
func rotate(_ nums: inout [Int], _ k: Int) { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.15 Jump Game ■■

Can reach the last index

Code Example:

```
func canJump(_ nums: [Int]) -> Bool { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.16 Spiral Matrix ■■

Return elements in spiral order

Code Example:

```
func spiralOrder(_ matrix: [[Int]]) -> [Int] { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.17 Meeting Rooms ■

Check if can attend all meetings

Code Example:

```
func canAttendMeetings(_ intervals: [[Int]]) -> Bool { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.18 Meeting Rooms II ■■

Minimum meeting rooms needed

Code Example:

```
func minMeetingRooms(_ intervals: [[Int]]) -> Int { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.19 Non-overlapping Intervals ■■

Remove minimum intervals

Code Example:

```
func eraseOverlapIntervals(_ intervals: [[Int]]) -> Int { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.1.20 Jump Game II ■■

Minimum jumps to reach end

Code Example:

```
func jump(_ nums: [Int]) -> Int { /* implementation */ }
```

Key Points:

- **Time/Space complexity**

Notes:

Algorithm explanation

A.2 String Problems (15 Problems)

A.2.1 Valid Anagram ■

Check if two strings are anagrams

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.2 Valid Palindrome ■

Check palindrome ignoring non-alphanumeric

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.3 Longest Substring Without Repeating ■■

Sliding window technique

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.4 Longest Repeating Character Replacement ■■

K character replacements

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.5 Minimum Window Substring ■■■

Minimum window containing all chars

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.6 Group Anagrams ■■

Group strings that are anagrams

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.7 Valid Parentheses ■

Check balanced parentheses

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.8 Longest Palindromic Substring ■■

Find longest palindrome

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.9 Palindromic Substrings ■■

Count all palindromic substrings

Code Example:

```
    // Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.10 Encode and Decode Strings ■■

Design encoding algorithm

Code Example:

```
    // Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.11 String to Integer (atoi) ■■

Convert string to integer

Code Example:

```
    // Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.12 Reverse Words in String ■■

Reverse words efficiently

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.13 Implement strStr() ■

Find needle in haystack

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.14 Text Justification ■■■

Format text to width

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.2.15 Regular Expression Matching ■■■

Pattern matching with . and *

Code Example:

```
// Swift solution with string manipulation techniques
```

Key Points:

- **Time/Space complexity**

Notes:

String processing algorithm explanation

A.3 Linked List Problems (12 Problems)

```
// ListNode Definition for Linked List Problems class ListNode { var val: Int var
next: ListNode? init() { self.val = 0; self.next = nil } init(_ val: Int) { self.val
= val; self.next = nil } init(_ val: Int, _ next: ListNode?) { self.val = val;
self.next = next } }
```

A.3.1 Reverse Linked List ■

Reverse list iteratively and recursively

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.2 Merge Two Sorted Lists ■

Merge two sorted lists

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.3 Remove Nth Node From End ■■

One-pass solution

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.4 Linked List Cycle ■

Floyd's cycle detection

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.5 Linked List Cycle II ■■

Find cycle start

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.6 Merge k Sorted Lists ■■■

Divide and conquer approach

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.7 Remove Duplicates from Sorted List ■

Remove duplicates

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.8 Intersection of Two Linked Lists ■

Find intersection node

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.9 Palindrome Linked List ■

Check if list is palindrome

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.10 Add Two Numbers ■■

Add numbers as linked lists

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.11 Copy List with Random Pointer ■■

Deep copy complex list

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.3.12 LRU Cache ■■■

Implement LRU cache

Code Example:

```
// Swift linked list solution with optimal approach
```

Key Points:

- **Time/Space complexity**

Notes:

Linked list algorithm explanation

A.4 Binary Tree Problems (15 Problems)

```
// TreeNode Definition for Binary Tree Problems class TreeNode { var val: Int var
left: TreeNode? var right: TreeNode? init() { self.val = 0; self.left = nil;
self.right = nil } init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil } init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) { self.val = val;
self.left = left; self.right = right } }
```

A.4.1 Maximum Depth of Binary Tree ■

DFS recursive solution

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.2 Same Tree ■

Compare two trees

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.3 Invert Binary Tree ■

Mirror binary tree

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.4 Binary Tree Level Order Traversal ■■

BFS traversal

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.5 Subtree of Another Tree ■■

Check if subtree exists

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.6 Lowest Common Ancestor ■■

Find LCA in BST

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.7 Binary Tree Right Side View ■■

Right side view

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.8 Count Good Nodes ■■

Count good nodes in tree

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.9 Validate Binary Search Tree ■■

Check valid BST

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.10 Kth Smallest in BST ■■

Inorder traversal approach

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.11 Construct Tree from Traversals ■■

Build from preorder/inorder

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.12 Binary Tree Max Path Sum ■■■

Maximum path sum

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.13 Serialize and Deserialize Tree ■■■

Convert tree to/from string

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.14 Word Search II ■■■

Trie + DFS backtracking

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.4.15 Balanced Binary Tree ■

Check height balanced

Code Example:

```
// Swift binary tree solution with DFS/BFS
```

Key Points:

- **Time/Space complexity**

Notes:

Tree algorithm explanation

A.5 Dynamic Programming Problems (15 Problems)

A.5.1 Climbing Stairs ■

Basic DP - Fibonacci pattern

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.2 House Robber ■■

Linear DP optimization

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.3 House Robber II ■■

Circular array DP

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.4 Longest Palindromic Subsequence ■■

2D DP approach

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.5 Palindromic Substrings ■■

Expand around centers

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.6 Decode Ways ■■

String DP pattern

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.7 Coin Change ■■

Unbounded knapsack

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.8 Maximum Product Subarray ■■

Track min/max products

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.9 Word Break ■■

String segmentation DP

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.10 Combination Sum IV ■■

Count combinations

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.11 Longest Increasing Subsequence ■■

LIS with binary search

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.12 Unique Paths ■■

Grid path counting

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.13 Jump Game ■■

Greedy vs DP approach

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.14 Edit Distance ■■■

Levenshtein distance

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.5.15 Regular Expression Matching ■■■

2D DP with patterns

Code Example:

```
// Swift DP solution with memoization/tabulation
```

Key Points:

- **Time/Space complexity**

Notes:

Dynamic programming approach explanation

A.6 Graph Problems (12 Problems)

A.6.1 Number of Islands ■■

DFS/BFS grid traversal

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.2 Clone Graph ■■

Deep clone with DFS

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.3 Max Area of Island ■■

DFS with area calculation

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.4 Pacific Atlantic Water Flow ■■

Multi-source BFS/DFS

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.5 Surrounded Regions ■■

Boundary DFS technique

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.6 Rotting Oranges ■■

Multi-source BFS

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.7 Course Schedule ■■

Topological sort - cycle detection

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.8 Course Schedule II ■■

Topological ordering

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.9 Redundant Connection ■■

Union-Find cycle detection

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.10 Word Ladder ■■■

BFS shortest path

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.11 Alien Dictionary ■■■

Topological sort

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.6.12 Network Delay Time ■■

Dijkstra's algorithm

Code Example:

```
// Swift graph solution with BFS/DFS/Union-Find
```

Key Points:

- **Time/Space complexity**

Notes:

Graph algorithm explanation

A.7 Heap & Priority Queue Problems (8 Problems)

A.7.1 Kth Largest Element ■■

Quick select vs heap

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.2 Last Stone Weight ■

Max heap simulation

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.3 K Closest Points to Origin ■■

Min heap approach

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.4 Task Scheduler ■■

Greedy with max heap

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.5 Top K Frequent Elements ■■

Bucket sort vs heap

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.6 Find Median from Data Stream ■■■

Two heaps technique

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.7 Merge k Sorted Lists ■■■

Min heap approach

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.7.8 Meeting Rooms II ■■

Min heap for end times

Code Example:

```
// Swift heap/priority queue solution
```

Key Points:

- **Time/Space complexity**

Notes:

Heap algorithm explanation

A.8 Stack Problems (6 Problems)

A.8.1 Valid Parentheses ■

Stack for matching brackets

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.8.2 Min Stack ■■

Stack with O(1) minimum

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.8.3 Evaluate Reverse Polish Notation ■■

Stack evaluation

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.8.4 Daily Temperatures ■■

Monotonic stack

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.8.5 Car Fleet ■■

Stack simulation

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.8.6 Largest Rectangle in Histogram ■■■

Stack with indices

Code Example:

```
// Swift stack-based solution
```

Key Points:

- **Time/Space complexity**

Notes:

Stack algorithm explanation

A.9 Binary Search Problems (8 Problems)

A.9.1 Binary Search ■

Classic binary search template

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.2 Search Insert Position ■

Find insertion index

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.3 Search in Rotated Sorted Array ■■

Modified binary search

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.4 Find Minimum in Rotated Array ■■

Binary search variation

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.5 Time Based Key-Value Store ■■

Binary search on timestamps

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.6 Search 2D Matrix ■■

Treat as 1D sorted array

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.7 Koko Eating Bananas ■■

Binary search on answer

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

A.9.8 Median of Two Sorted Arrays ■■■

Binary search partition

Code Example:

```
// Swift binary search solution
```

Key Points:

- **Time/Space complexity**

Notes:

Binary search technique explanation

■ SUMMARY: This appendix contains 100+ essential DSA problems with optimized Swift solutions. Each problem is carefully selected for technical interviews and includes multiple solution approaches where applicable. The problems progress from basic to advanced, covering all major algorithmic patterns and data structures. ■ = Easy, ■■ = Medium, ■■■ = Hard Total Problems: 111 problems across 9 categories • Arrays: 20 problems • Strings: 15 problems • Linked Lists: 12 problems • Binary Trees: 15 problems • Dynamic Programming: 15 problems • Graphs: 12 problems • Heaps: 8 problems • Stacks: 6 problems • Binary Search: 8 problems