Продвинутые возможности и парадигмы в Swift

Презентация 4

10 октября 2025 г.

Содержание

Обобщенные типы (Generics)

- Generics позволяют писать гибкий и переиспользуемый код
- Обеспечивают типобезопасность во время компиляции
- Позволяют избежать дублирования кода
- Поддерживают ограничения типов (type constraints)

Базовые обобщенные функции

```swift
// Простая обобщенная функция
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}

var int1 = 5, int2 = 10
swapValues(&int1, &int2)
print("int1: \(int1), int2: \(int2)") // int1: 10, int2: 5

var str1 = "Hello", str2 = "World"
swapValues(&str1, &str2)
print("str1: \(str1), str2: \(str2)") // str1: World, str2: Hello

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

let numbers = [1, 3, 5, 7, 9]
let strings = ["apple", "banana", "cherry"]

if let index = findIndex(of: 5, in: numbers) {
    print("Найдено на позиции: \(index)")
}
```

Обобщенные типы и структуры

```swift
// Обобщенная структура стека
struct Stack<Element> {
    private var items: [Element] = []

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

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

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

    var isEmpty: Bool {
        return items.isEmpty
    }
}

// Использование обобщенного стека
var intStack = Stack<Int>()
intStack.push(1)
intStack.push(2)
intStack.push(3)

print(intStack.pop()) // 3
print(intStack.peek()) // 2

var stringStack = Stack<String>()
stringStack.push("Hello")
stringStack.push("World")
```

Ограничения типов (Type Constraints)

```swift
// Ограничение по протоколу
func findLargest<T: Comparable>(_ array: [T]) -> T? {
    guard !array.isEmpty else { return nil }

    var largest = array[0]
    for item in array {
        if item > largest {
            largest = item
        }
    }
    return largest
}

let numbers = [3, 1, 4, 1, 5, 9, 2, 6]
let largestNumber = findLargest(numbers) // 9

let words = ["apple", "banana", "cherry"]
let largestWord = findLargest(words) // "cherry"

// Множественные ограничения
protocol Drawable {
    func draw()
}

protocol Measurable {
    var area: Double { get }
}

func processShape<T: Drawable & Measurable>(_ shape: T) {
    shape.draw()
    print("Площадь: \(shape.area)")
}

// Ограничение по классу
class Animal {
```

Функциональное программирование в Swift

- Swift поддерживает функциональные парадигмы
- Функции являются объектами первого класса
- Поддержка высших функций (map, filter, reduce)
- Неизменяемость и чистые функции

Высшие функции - Map и Filter

```
1  // Map - преобразование элементов
2  let numbers = [1, 2, 3, 4, 5]
3  let doubled = numbers.map { $0 * 2 }
4  let strings = numbers.map { "Number \($0)" }
5
6  print(doubled) // [2, 4, 6, 8, 10]
7  print(strings) // ["Number 1", "Number 2", ...]
8
9  // Filter - фильтрация элементов
10 let evenNumbers = numbers.filter { $0 % 2 == 0 }
11 let largeNumbers = numbers.filter { $0 > 3 }
12
13 print(evenNumbers) // [2, 4]
14 print(largeNumbers) // [4, 5]
```

Высшие функции - Reduce и комбинирование

```
1  // Reduce - свертка массива
2  let numbers = [1, 2, 3, 4, 5]
3  let sum = numbers.reduce(0, +)
4  let product = numbers.reduce(1, *)
5  let concatenated = numbers.reduce("", { "\($0)\($1)" })
6
7  print(sum) // 15
8  print(product) // 120
9  print(concatenated) // "12345"
10
11 // Комбинирование функций
12 let result = numbers
13     .filter { $0 % 2 == 0 }
14     .map { $0 * $0 }
15     .reduce(0, +)
16 print(result) // 20 (4 + 16)
```

Функциональные композиции

```swift
// Композиция функций
func addOne(_ x: Int) -> Int { return x + 1 }
func multiplyByTwo(_ x: Int) -> Int { return x * 2 }
func square(_ x: Int) -> Int { return x * x }

// Простая композиция
let numbers = [1, 2, 3, 4, 5]
let processed = numbers.map { square(multiplyByTwo(addOne($0))) }
print(processed) // [16, 36, 64, 100, 144]

// Оператор композиции
infix operator >>>: AdditionPrecedence
func >>> <T, U, V>(f: @escaping (T) -> U, g: @escaping (U) -> V) -> (T) -> V {
    return { g(f($0)) }
}
```

Использование композиции

```swift
// Использование оператора композиции
let addOneAndMultiply = addOne >>> multiplyByTwo
let addOneMultiplyAndSquare = addOne >>> multiplyByTwo >>> square

let numbers = [1, 2, 3, 4, 5]
let result1 = numbers.map(addOneAndMultiply)
let result2 = numbers.map(addOneMultiplyAndSquare)

print(result1) // [4, 6, 8, 10, 12]
print(result2) // [16, 36, 64, 100, 144]

// Частичное применение функций
func multiply(_ a: Int, _ b: Int) -> Int { return a * b }
let multiplyByThree = { multiply(3, $0) }
let tripled = numbers.map(multiplyByThree)
print(tripled) // [3, 6, 9, 12, 15]
```

Чистые функции

```swift
// Чистые функции (без побочных эффектов)
func pureAdd(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func pureFactorial(_ n: Int) -> Int {
    if n <= 1 { return 1 }
    return n * pureFactorial(n - 1)
}

// Использование чистых функций
let sum = pureAdd(5, 3) // 8
let factorial = pureFactorial(5) // 120

// Чистые функции всегда предсказуемы
print("Сумма: \(sum)")
print("Факториал: \(factorial)")
```

Неизменяемые структуры данных

```swift
// Неизменяемые структуры данных
struct ImmutablePoint {
    let x: Int
    let y: Int

    func moved(by deltaX: Int, deltaY: Int) -> ImmutablePoint {
        return ImmutablePoint(x: x + deltaX, y: y + deltaY)
    }
}

let point = ImmutablePoint(x: 5, y: 10)
let movedPoint = point.moved(by: 3, by: -2)
print("Исходная точка: (\(point.x), \(point.y))")
print("Новая точка: (\(movedPoint.x), \(movedPoint.y))")

// Исходная точка остается неизменной
print("Исходная точка все еще: (\(point.x), \(point.y))")
```

Функциональные структуры данных

```
1  // Функциональные структуры данных
2  enum List<T> {
3      case empty
4      indirect case cons(T, List<T>)
5
6      func map<U>(_ transform: (T) -> U) -> List<U> {
7          switch self {
8          case .empty:
9              return .empty
10         case .cons(let head, let tail):
11             return .cons(transform(head), tail.map(transform))
12         }
13     }
14 }
15
16 // Использование функционального списка
17 let numbers = List.cons(1, List.cons(2, List.cons(3, List.empty)))
18 let doubled = numbers.map { $0 * 2 }
```

ООП в Swift

- Классы и наследование
- Инкапсуляция и полиморфизм
- Протоколы как интерфейсы
- Расширения для добавления функциональности

Классы и наследование

```swift
// Базовый класс
class Vehicle {
    var speed: Double = 0
    var maxSpeed: Double

    init(maxSpeed: Double) {
        self.maxSpeed = maxSpeed
    }

    func accelerate(by amount: Double) {
        speed = min(speed + amount, maxSpeed)
    }

    func brake(by amount: Double) {
        speed = max(speed - amount, 0)
    }

    func describe() -> String {
        return "Транспортное средство со скоростью \(speed) км/ч"
    }
}

// Наследование
class Car: Vehicle {
    var numberOfDoors: Int

    init(maxSpeed: Double, numberOfDoors: Int) {
        self.numberOfDoors = numberOfDoors
        super.init(maxSpeed: maxSpeed)
    }

    override func describe() -> String {
        return "Автомобиль с \(numberOfDoors) дверями, скорость \(speed) км/ч"
    }
```

Полиморфизм и виртуальные методы

```swift
// Полиморфизм через наследование
class Animal {
    func makeSound() {
        print("Some generic animal sound")
    }

    func move() {
        print("Moving...")
    }
}

class Dog: Animal {
    override func makeSound() {
        print("Woof!")
    }

    override func move() {
        print("Running on four legs")
    }
}

class Bird: Animal {
    override func makeSound() {
        print("Tweet!")
    }

    override func move() {
        print("Flying")
    }
}

// Использование полиморфизма
let animals: [Animal] = [Dog(), Bird(), Animal()]

for animal in animals {
```

Инкапсуляция и доступ

```swift
// Инкапсуляция через уровни доступа
class BankAccount {
    private var balance: Double = 0
    private let accountNumber: String

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

    // Публичные методы для взаимодействия
    func deposit(amount: Double) {
        guard amount > 0 else {
            print("Сумма должна быть положительной")
            return
        }
        balance += amount
        print("Внесено \(amount). Баланс: \(balance)")
    }

    func withdraw(amount: Double) -> Bool {
        guard amount > 0 else {
            print("Сумма должна быть положительной")
            return false
        }

        guard amount <= balance else {
            print("Недостаточно средств")
            return false
        }

        balance -= amount
        print("Снято \(amount). Баланс: \(balance)")
        return true
    }
```

Протокольно-ориентированное программирование

- Протоколы определяют контракты
- Расширения добавляют реализацию по умолчанию
- Композиция вместо наследования
- Гибкость и переиспользуемость кода

Базовые протоколы

```swift
// Определение протокола
protocol Drawable {
    func draw()
}

protocol Measurable {
    var area: Double { get }
    var perimeter: Double { get }
}

// Реализация протокола
struct Circle: Drawable, Measurable {
    let radius: Double

    func draw() {
        print("Рисуем круг радиусом \(radius)")
    }

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

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

struct Rectangle: Drawable, Measurable {
    let width: Double
    let height: Double

    func draw() {
        print("Рисуем прямоугольник \(width)x\(height)")
    }
```

Расширения протоколов

```swift
// Расширение протокола с реализацией по умолчанию
protocol Flyable {
    func fly()
}

extension Flyable {
    func fly() {
        print("Летит...")
    }

    func takeOff() {
        print("Взлетает")
    }
}

// Протокол с ассоциированными типами
protocol Container {
    associatedtype Item

    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

// Реализация контейнера
struct Stack<Element>: Container {
    typealias Item = Element

    private var items: [Element] = []

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

    var count: Int {
```

Композиция протоколов

```
1  // Композиция протоколов
2  protocol Engine {
3      func start()
4      func stop()
5  }
6
7  protocol Wheels {
8      func rotate()
9  }
10
11 protocol VehicleProtocol: Engine, Wheels {
12     func move()
13 }
14
15 // Реализация через композицию
16 class Car: VehicleProtocol {
17     func start() {
18         print("Двигатель запущен")
19     }
20
21     func stop() {
22         print("Двигатель остановлен")
23     }
24
25     func rotate() {
26         print("Колеса вращаются")
27     }
28
29     func move() {
30         start()
31         rotate()
32         print("Автомобиль движется")
33     }
34 }
35
```

Ключевые моменты

- **Generics** обеспечивают типобезопасность и переиспользуемость
- **Функциональное программирование** упрощает обработку данных
- **ООП** предоставляет классические паттерны проектирования
- **Протокольно-ориентированное программирование** обеспечивает гибкость
- **Композиция** часто предпочтительнее наследования
- **Расширения** позволяют добавлять функциональность к существующим типам

Практические рекомендации

- Используйте `generics` для создания переиспользуемого кода
- Предпочитайте `map`, `filter`, `reduce` циклам
- Применяйте протоколы для определения контрактов
- Используйте расширения для добавления функциональности
- Композиция протоколов гибче наследования классов
- Следуйте принципам SOLID при проектировании архитектуры