

Системные аспекты и библиотеки в Swift

Презентация 5

10 октября 2025 г.

Содержание

Стандартная библиотека ввода-вывода

Конкурентность / параллелизм

Специфические конструкции языка

Управление памятью

Виртуальная машина

Заключение

Стандартная библиотека ввода-вывода

- ▶ Foundation framework для базовых операций
- ▶ Работа с файлами и директориями
- ▶ Сетевое программирование
- ▶ Сериализация и десериализация данных

Работа с файлами

```
1 import Foundation
2
3 // Создание и запись в файл
4 let documentsPath = FileManager.default.urls(for: .documentDirectory,
5                                              in: .userDomainMask)[0]
6 let fileURL = documentsPath.appendingPathComponent("data.txt")
7
8 let content = "Привет, мир!\nЭто тестовый файл."
9 do {
10     try content.write(to: fileURL, atomically: true, encoding: .utf8)
11     print("Файл создан успешно")
12 } catch {
13     print("Ошибка записи файла: \(error)")
14 }
15
16 // Чтение из файла
17 do {
18     let readContent = try String(contentsOf: fileURL, encoding: .utf8)
19     print("Содержимое файла: \(readContent)")
20 } catch {
21     print("Ошибка чтения файла: \(error)")
22 }
23
24 // Проверка существования файла
25 if FileManager.default.fileExists(atPath: fileURL.path) {
26     print("Файл существует")
27
28     // Получение информации о файле
29     let attributes = try? FileManager.default.attributesOfItem(atPath: fileURL.
30                                                                    path)
31     if let size = attributes?[.size] as? Int {
32         print("Размер файла: \(size) байт")
33     }
34 }
```

Работа с директориями

```
1 import Foundation
2
3 // Создание директории
4 let documentsPath = FileManager.default.urls(for: .documentDirectory,
5                                              in: .userDomainMask)[0]
6 let newDirectory = documentsPath.appendingPathComponent("MyApp")
7
8 do {
9     try FileManager.default.createDirectory(at: newDirectory,
10                                           withIntermediateDirectories: true)
11     print("Директория создана")
12 } catch {
13     print("Ошибка создания директории: \(error)")
14 }
15
16 // Перечисление содержимого директории
17 do {
18     let contents = try FileManager.default.contentsOfDirectory(at: documentsPath
19                                                                ,
20                                                                includingPropertiesForKeys
21                                                                : nil)
22
23     print("Содержимое директории:")
24     for item in contents {
25         print("- \(item.lastPathComponent)")
26     }
27 } catch {
28     print("Ошибка чтения директории: \(error)")
29 }
30
31 // Копирование файла
32 let sourceFile = documentsPath.appendingPathComponent("data.txt")
33 let destinationFile = newDirectory.appendingPathComponent("copied_data.txt")
34
35 do {
36     try FileManager.default.copyItem(at: sourceFile, to: destinationFile)
```

Сетевое программирование

```
1 import Foundation
2
3 // Простой HTTP запрос
4 func makeHTTPRequest(url: String, completion: @escaping (Data?, Error?) -> Void)
5 {
6     guard let url = URL(string: url) else {
7         completion(nil, NSError(domain: "Invalid URL", code: 0))
8         return
9     }
10
11     let task = URLSession.shared.dataTask(with: url) { data, response, error in
12         DispatchQueue.main.async {
13             completion(data, error)
14         }
15     }
16     task.resume()
17 }
18
19 // Использование HTTP запроса
20 makeHTTPRequest(url: "https://api.github.com/users/apple") { data, error in
21     if let error = error {
22         print("Ошибка: \(error)")
23         return
24     }
25
26     if let data = data {
27         do {
28             let json = try JSONSerialization.jsonObject(with: data)
29             print("JSON ответ: \(json)")
30         } catch {
31             print("Ошибка парсинга JSON: \(error)")
32         }
33     }
34 }
```

Конкурентность в Swift

- ▶ Grand Central Dispatch (GCD) для многопоточности
- ▶ `async/await` для асинхронного программирования
- ▶ `Operation` и `OperationQueue` для сложных задач
- ▶ Actor модель для безопасного доступа к данным

Grand Central Dispatch - Основы

```
1 import Foundation
2
3 // Основные очереди
4 DispatchQueue.main.async {
5     print("Выполняется на главной очереди")
6 }
7
8 DispatchQueue.global(qos: .background).async {
9     print("Выполняется в фоновом режиме")
10
11     // Возврат на главную очередь
12     DispatchQueue.main.async {
13         print("Вернулись на главную очередь")
14     }
15 }
16
17 // Создание собственной очереди
18 let customQueue = DispatchQueue(label: "com.example.queue",
19                                 qos: .userInitiated)
20
21 customQueue.async {
22     print("Выполняется на пользовательской очереди")
23 }
```

Grand Central Dispatch - Синхронизация

```
1 // Синхронное выполнение
2 let customQueue = DispatchQueue(label: "com.example.queue",
3                                   qos: .userInitiated)
4
5 customQueue.sync {
6     print("Синхронное выполнение")
7 }
8
9 // Группы задач
10 let group = DispatchGroup()
11
12 for i in 1...3 {
13     group.enter()
14     DispatchQueue.global().async {
15         print("Задача \(i) выполняется")
16         Thread.sleep(forTimeInterval: 1)
17         group.leave()
18     }
19 }
20
21 group.notify(queue: .main) {
22     print("Все задачи завершены")
23 }
```

async/await - Основы

```
1 import Foundation
2
3 // Асинхронная функция
4 func fetchData() async throws -> String {
5     // Симуляция сетевого запроса
6     try await Task.sleep(nanoseconds: 1_000_000_000) // 1 секунда
7     return "Данные получены"
8 }
9
10 // Использование async/await
11 Task {
12     do {
13         let data = try await fetchData()
14         print(data)
15     } catch {
16         print("Ошибка: \(error)")
17     }
18 }
```

async/await - Параллельное выполнение

```
1 // Параллельное выполнение
2 func fetchData() async -> String {
3     try? await Task.sleep(nanoseconds: 500_000_000)
4     return "Данные пользователя"
5 }
6
7 func fetchSettings() async -> String {
8     try? await Task.sleep(nanoseconds: 300_000_000)
9     return "Настройки"
10 }
11
12 Task {
13     async let userData = fetchData()
14     async let settings = fetchSettings()
15
16     let (user, config) = await (userData, settings)
17     print("Пользователь: \(user), Настройки: \(config)")
18 }
```

async/await - Отмена задач

```
1 // Отмена задач
2 let task = Task {
3     for i in 1...10 {
4         try await Task.sleep(nanoseconds: 100_000_000)
5         print("Итерация \(i)")
6     }
7 }
8
9 // Отмена через 1 секунду
10 Task {
11     try await Task.sleep(nanoseconds: 1_000_000_000)
12     task.cancel()
13 }
```

Operation и OperationQueue - Кастомные операции

```
1 import Foundation
2
3 // Кастомная операция
4 class DataProcessingOperation: Operation {
5     private let data: [Int]
6     private var _result: [Int] = []
7
8     var result: [Int] {
9         return _result
10    }
11
12    init(data: [Int]) {
13        self.data = data
14        super.init()
15    }
16
17    override func main() {
18        // Проверка отмены
19        guard !isCancelled else { return }
20
21        // Обработка данных
22        _result = data.map { $0 * 2 }.filter { $0 > 10 }
23
24        // Проверка отмены после обработки
25        guard !isCancelled else { return }
26
27        print("Обработка завершена: \(_result)")
28    }
29 }
```

Operation и OperationQueue - Использование

```
1 // Использование операций
2 let operationQueue = OperationQueue()
3 operationQueue.maxConcurrentOperationCount = 2
4
5 let operation1 = DataProcessingOperation(data: [1, 2, 3, 4, 5])
6 let operation2 = DataProcessingOperation(data: [6, 7, 8, 9, 10])
7
8 // Зависимости между операциями
9 operation2.addDependency(operation1)
10
11 operationQueue.addOperation(operation1)
12 operationQueue.addOperation(operation2)
13
14 // Ожидание завершения
15 operationQueue.waitUntilAllOperationsAreFinished()
16 print("Все операции завершены")
```

Operation и OperationQueue - Блок операции

```
1 // Блок операция
2 let blockOperation = BlockOperation {
3     print("Выполняется блок операция")
4 }
5
6 blockOperation.addExecutionBlock {
7     print("Дополнительный блок")
8 }
9
10 operationQueue.addOperation(blockOperation)
```

Специфические конструкции Swift

- ▶ Property Wrappers для инкапсуляции логики
- ▶ Result Builders для DSL
- ▶ Key Paths для типобезопасных ссылок
- ▶ Dynamic Member Lookup для динамического доступа

Property Wrappers - UserDefault

```
1 // Создание Property Wrapper
2 @propertyWrapper
3 struct UserDefault<T> {
4     let key: String
5     let defaultValue: T
6
7     var wrappedValue: T {
8         get {
9             UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
10        }
11        set {
12            UserDefaults.standard.set(newValue, forKey: key)
13        }
14    }
15 }
16
17 // Использование Property Wrapper
18 struct Settings {
19     @UserDefault(key: "username", defaultValue: "")
20     var username: String
21
22     @UserDefault(key: "isFirstLaunch", defaultValue: true)
23     var isFirstLaunch: Bool
24 }
25
26 var settings = Settings()
27 settings.username = "user123"
28 print("Username: \(settings.username)")
```

Property Wrappers - Валидация

```
1 // Property Wrapper для валидации
2 @propertyWrapper
3 struct ValidatedString {
4     private var value: String = ""
5     private let validator: (String) -> Bool
6
7     init(wrappedValue: String, validator: @escaping (String) -> Bool) {
8         self.validator = validator
9         self.wrappedValue = wrappedValue
10    }
11
12    var wrappedValue: String {
13        get { value }
14        set {
15            if validator(newValue) {
16                value = newValue
17            } else {
18                print("Неверное значение: \(newValue)")
19            }
20        }
21    }
22 }
23
24 // Использование валидации
25 struct UserSettings {
26     @ValidatedString(validator: { $0.count >= 3 })
27     var password: String = ""
28 }
29
30 var userSettings = UserSettings()
31 userSettings.password = "abc" // Валидно
32 userSettings.password = "ab"  // Невалидно - слишком короткий
```

Key Paths - Основы

```
1 // Структура для демонстрации Key Paths
2 struct Person {
3     let name: String
4     let age: Int
5     let address: Address
6 }
7
8 struct Address {
9     let street: String
10    let city: String
11 }
12
13 // Создание объектов
14 let people = [
15     Person(name: "Анна", age: 25, address: Address(street: "Ленина 1", city: "Мо
        сква")),
16     Person(name: "Петр", age: 30, address: Address(street: "Пушкина 5", city: "С
        пб")),
17     Person(name: "Мария", age: 28, address: Address(street: "Гагарина 10", city:
        "Москва"))
18 ]
```

Key Paths - Использование

```
1 // Использование Key Paths
2 let names = people.map(\.name)
3 let ages = people.map(\.age)
4 let cities = people.map(\.address.city)
5
6 print("Имена: \(names)")
7 print("Возрасты: \(ages)")
8 print("Города: \(cities)")
9
10 // Key Paths для сортировки
11 let sortedByAge = people.sorted(by: \.age)
12 let sortedByName = people.sorted(by: \.name)
13
14 // Key Paths для фильтрации
15 let moscowResidents = people.filter { $0[keyPath: \.address.city] == "Москва" }
```

Key Paths - Динамические ссылки

```
1 // Динамические Key Paths
2 let ageKeyPath = \Person.age
3 let nameKeyPath = \Person.name
4
5 for person in people {
6     print("\(person[keyPath: nameKeyPath]): \(person[keyPath: ageKeyPath]) лет")
7 }
8
9 // Использование в функциях
10 func getProperty<T>(<_ keyPath: KeyPath<Person, T>, from person: Person) -> T {
11     return person[keyPath: keyPath]
12 }
13
14 let personName = getProperty(\.name, from: people[0])
15 let personAge = getProperty(\.age, from: people[0])
```

Dynamic Member Lookup - Основы

```
1 // Dynamic Member Lookup для динамического доступа
2 @dynamicMemberLookup
3 struct DynamicDictionary {
4     private var dictionary: [String: Any] = [:]
5
6     subscript(dynamicMember member: String) -> Any? {
7         get {
8             return dictionary[member]
9         }
10        set {
11            dictionary[member] = newValue
12        }
13    }
14
15    subscript<T>(dynamicMember member: String) -> T? {
16        return dictionary[member] as? T
17    }
18 }
19
20 // Использование Dynamic Member Lookup
21 var config = DynamicDictionary()
22 config.username = "admin"
23 config.password = "secret123"
24 config.isEnabled = true
25
26 print(config.username as? String ?? "nil")
27 print(config.isEnabled as? Bool ?? false)
```

Dynamic Member Lookup - Типизированная версия

```
1 // Dynamic Member Lookup с типизацией
2 @dynamicMemberLookup
3 struct TypedConfig {
4     private var storage: [String: Any] = [:]
5
6     subscript<T>(dynamicMember member: String) -> T? {
7         return storage[member] as? T
8     }
9
10    mutating func set<T>(_ value: T, for key: String) {
11        storage[key] = value
12    }
13 }
14
15 var typedConfig = TypedConfig()
16 typedConfig.set("localhost", for: "host")
17 typedConfig.set(8080, for: "port")
18 typedConfig.set(true, for: "debug")
19
20 let host: String? = typedConfig.host
21 let port: Int? = typedConfig.port
22 let debug: Bool? = typedConfig.debug
23
24 print("Host: \(host ?? "unknown"), Port: \(port ?? 0), Debug: \(debug ?? false)"
    )
```

Управление памятью в Swift

- ▶ ARC (Automatic Reference Counting) для автоматического управления
- ▶ Weak и unowned ссылки для предотвращения циклов
- ▶ Strong reference cycles и их решение
- ▶ Memory leaks и способы их избежания

ARC и ссылочные циклы

```
1 // Проблема сильных ссылочных циклов
2 class Person {
3     let name: String
4     var apartment: Apartment?
5
6     init(name: String) {
7         self.name = name
8         print("\(name) инициализирован")
9     }
10
11     deinit {
12         print("\(name) деинициализирован")
13     }
14 }
15
16 class Apartment {
17     let unit: String
18     var tenant: Person?
19
20     init(unit: String) {
21         self.unit = unit
22         print("Квартира \(unit) инициализирована")
23     }
24
25     deinit {
26         print("Квартира \(unit) деинициализирована")
27     }
28 }
29
30 // Создание сильного цикла
31 var john: Person? = Person(name: "Джон")
32 var unit4A: Apartment? = Apartment(unit: "4A")
33
34 john!.apartment = unit4A
35 unit4A!.tenant = john
```

Решение сильных циклов

```
1 // Решение с weak ссылками
2 class Person {
3     let name: String
4     var apartment: Apartment?
5
6     init(name: String) {
7         self.name = name
8         print("\(name) инициализирован")
9     }
10
11     deinit {
12         print("\(name) деинициализирован")
13     }
14 }
15
16 class Apartment {
17     let unit: String
18     weak var tenant: Person? // Слабая ссылка
19
20     init(unit: String) {
21         self.unit = unit
22         print("Квартира \(unit) инициализирована")
23     }
24
25     deinit {
26         print("Квартира \(unit) деинициализирована")
27     }
28 }
29
30 // Теперь цикл разрывается
31 var john: Person? = Person(name: "Джон")
32 var unit4A: Apartment? = Apartment(unit: "4A")
33
34 john!.apartment = unit4A
35 unit4A!.tenant = john
```

Управление памятью в замыканиях

```
1 // Захват в замыканиях
2 class HTMLElement {
3     let name: String
4     let text: String?
5
6     lazy var asHTML: () -> String = {
7         if let text = self.text {
8             return "<\(self.name)>\(text)</\(\self.name)>"
9         } else {
10             return "<\(self.name) />"
11         }
12     }
13
14     init(name: String, text: String? = nil) {
15         self.name = name
16         self.text = text
17     }
18
19     deinit {
20         print("\(name) деинициализирован")
21     }
22 }
23
24 // Создание элемента
25 var paragraph: HTMLElement? = HTMLElement(name: "p", text: "Hello, world")
26 print(paragraph!.asHTML())
27
28 paragraph = nil
29 // Элемент деинициализирован
30
31 // Решение проблемы захвата
32 class HTMLElementFixed {
33     let name: String
34     let text: String?
```

Виртуальная машина Swift

- ▶ Swift Runtime для выполнения кода
- ▶ Objective-C Runtime совместимость
- ▶ Reflection и метапрограммирование
- ▶ Динамические возможности языка

Reflection и метапрограммирование

```
1 import Foundation
2
3 // Reflection с помощью Mirror
4 struct Person {
5     let name: String
6     let age: Int
7     let email: String
8 }
9
10 let person = Person(name: "Анна", age: 25, email: "anna@example.com")
11 let mirror = Mirror(reflecting: person)
12
13 print("Тип: \(mirror.subjectType)")
14 print("Количество детей: \(mirror.children.count)")
15
16 for child in mirror.children {
17     if let label = child.label {
18         print("\(label): \(child.value)")
19     }
20 }
21
22 // Динамическое создание объектов
23 class DynamicClass {
24     var properties: [String: Any] = [:]
25
26     func setValue(_ value: Any, forKey key: String) {
27         properties[key] = value
28     }
29
30     func getValue(forKey key: String) -> Any? {
31         return properties[key]
32     }
33 }
34
35 let dynamicObject = DynamicClass()
```

Objective-C Runtime совместимость

```
1 import Foundation
2
3 // @objc для совместимости с Objective-C
4 @objc class ObjectiveCCompatible: NSObject {
5     @objc var name: String
6     @objc var age: Int
7
8     @objc init(name: String, age: Int) {
9         self.name = name
10        self.age = age
11        super.init()
12    }
13
14    @objc func greet() -> String {
15        return "Привет, меня зовут \(name)"
16    }
17 }
18
19 // Динамическое выполнение методов
20 let obj = ObjectiveCCompatible(name: "Петр", age: 30)
21
22 // Получение информации о классе
23 let className = NSStringFromClass(type(of: obj))
24 print("Имя класса: \(className)")
25
26 // Вызов метода через селектор
27 let selector = #selector(ObjectiveCCompatible.greet)
28 if obj.responds(to: selector) {
29     let result = obj.perform(selector)?.takeUnretainedValue() as? String
30     print("Результат: \(result ?? "nil")")
31 }
32
33 // KVO (Key-Value Observing)
34 class ObservableObject: NSObject {
35     @objc dynamic var value: Int = 0
```

Динамические возможности

```
1 import Foundation
2
3 // Динамическое создание типов
4 func createDynamicType() -> Any.Type {
5     return type(of: "Hello")
6 }
7
8 let dynamicType = createDynamicType()
9 print("Динамический тип: \(dynamicType)")
10
11 // Проверка типов во время выполнения
12 func processValue(_ value: Any) {
13     switch value {
14     case let string as String:
15         print("Строка: \(string)")
16     case let int as Int:
17         print("Целое число: \(int)")
18     case let array as [Any]:
19         print("Массив с \(array.count) элементами")
20     default:
21         print("Неизвестный тип: \(type(of: value))")
22     }
23 }
24
25 processValue("Hello")
26 processValue(42)
27 processValue([1, 2, 3])
28
29 // Динамическое выполнение кода
30 func executeDynamicCode() {
31     let code = ""
32     let result = 2 + 3
33     print("Результат: \(result)")
34     ""
35 }
```

Ключевые моменты

- ▶ **Стандартная библиотека** предоставляет богатый набор инструментов
- ▶ **Конкурентность** обеспечивает производительность приложений
- ▶ **Специфические конструкции** делают код более выразительным
- ▶ **Управление памятью** автоматизировано через ARC
- ▶ **Виртуальная машина** обеспечивает выполнение и совместимость
- ▶ **Reflection** позволяет создавать динамические приложения

Практические рекомендации

- ▶ Используйте `async/await` для асинхронного программирования
- ▶ Применяйте `Property Wrappers` для инкапсуляции логики
- ▶ Избегайте сильных ссылочных циклов с помощью `weak`
- ▶ Используйте `Key Paths` для типобезопасных ссылок
- ▶ Применяйте `Operation` для сложных многопоточных задач
- ▶ Следите за управлением памятью в замыканиях