Профессиональное программирование на Swift

Урок 6

Чистый код

Правила использования различных типов данных, методов, циклов. Табличные методы.

Типы данных

Прочитав тему этого раздела, вы можете задать логичный вопрос о его необходимости. Ведь мы уже проходили типы данных языка Swift, знаем про фундаментальные типы: **Int** — для целых чисел, **Double** и **Float** — для значений с плавающей точкой, **Bool** — для булевых значений, **String** — для представления строковых данных. Но в этом уроке поговорим, как правильно их использовать и описывать.

Целые числа

Самая распространенная ошибка — **использование «магических» чисел**. Это те самые числовые данные, смысл которых неясен. Рассмотрим пример:

```
class SomeCell: UITableViewCell {

    @IBOutlet weak var textLabel: UILabel!
    @IBOutlet weak var applyButton: UIButton!

    func calculateCellHeight(text:String, font:UIFont, width:CGFloat) -> CGFloat
{
        let label:UILabel = UILabel(frame: CGRect(x: 0, y: 0, width: width,
height: 300))
        label.numberOfLines = 0
        label.font = font
        label.text = text
        label.sizeToFit()

        return label.frame.height + 20 + 2 + 20
    }
}
```

Это функция расчета высоты ячейки, в которой находится метка. Вроде функция полезная, но что это за непонятные константы — ‘300’, ‘20 + 2 + 20’? Не зная ответа на этот вопрос, страшно менять код, так как нет уверенности — если, например, внести изменения в **constraints** на **storyboard**, как они скажутся на коде?

Решение — замена магических чисел на константы с читаемыми названиями, объясняющими их смысл и назначение:

```
class SomeCell: UITableViewCell {

    @IBOutlet weak var textLabel: UILabel!
    @IBOutlet weak var applyButton: UIButton!

    func calculateCellHeight(text:String, font:UIFont, width:CGFloat) -> CGFloat
{
        let maxHeight = 300
        let label:UILabel = UILabel(frame: CGRect(x: 0, y: 0, width: width,
height: maxHeight))
        label.numberOfLines = 0
        label.font = font
        label.text = text
        label.sizeToFit()

        let defaultMargin = 20
```

```
        let defaultSeparatorHeight = 2
        return label.frame.height + defaultMargin * 2 + defaultSeparatorHeight
    }
}
```

При делении всегда делайте защиту от деления на 0. Это очевидная рекомендация, но иногда о ней забывают — и тогда приложение падает. Всегда держите в голове это правило и при любом делении добавляйте эту проверку в код.

Для целых чисел используйте понятное преобразование типов. Хотя для Swift это не столь актуально, так как этот язык — типобезопасный. Swift всегда помогает понять, с какими типами значений код может работать. Если код ожидает **String**, безопасность типов не даст передать ему **Int** по ошибке.

```
let one: Int = Int.max + 1
// В Int нельзя записать число больше его максимального значения
// Здесь будет ошибка компиляции

let two: UInt8 = -1
// В UInt8 нельзя записать отрицательное значение
// Здесь будет ошибка компиляции
```

Даже подобный код вызовет ошибку компиляции:

```
let one: Int = 2
let two: UInt8 = 1

let sum = one + two
// Нельзя производить сложение разных числовых типов
// Здесь будет ошибка компиляции
```

Чтобы сложение из приведенного примера заработало, нужно сделать преобразование одного числового типа в другой. Для этого необходимо создать новое число желаемого типа из существующего значения.

```
let one: Int = 2
let two: UInt8 = 1

let sum = one + Int(two)
```

Но подобные преобразования могут таить «подводные камни», о которых нужно помнить в таких случаях:

```
let one: Int = -1
// Внимание -1!
let two: UInt8 = 1

let sum = two + UInt8(one)
// Компиляция пройдет
// Но в процессе выполнения возникнет runtime-ошибка(
```

При небрежных преобразованиях можно потерять точность числа с плавающей точкой:

```swift
let one: Int = 2
let two: Double = 1.3

let result: Double = Double(one * Int(two))
// Ожидается, что в переменной result значение будет 2.6
// Но это не так — значение равно 2!
// Виновник этому — преобразование Int(two), которое на выходе дало значение 1
```

Числа с плавающей точкой

Число с плавающей точкой — это число с дробной частью. В Swift представлено два типа с плавающей точкой: **Double** и **Float**.

Double — это 64-битное число с плавающей точкой, **Float** — 32-битное. Из данного определения следуют и их области применения: Double — когда число может быть очень большим и точным, иначе используем **Float** (но предпочтительнее все равно **Double**).

Но даже этих типов бывает недостаточно. Например, для работы с денежными суммами. В этих случаях используют тип **Decimal**, который использует специальное представление, где число хранится в форме мантиссы и показателя степени.

При работе с данными типами надо помнить о контроле за преобразованиями типов:

```swift
let one: Double = 0.23
let two = Decimal(one)

print("Double: '\(one)' vs Decimal: '\(two)'")
// Вывод: Double: '0.23' vs Decimal: '0.2300000000000000512'
```

В этом примере показана неточность **Double** в представлении чисел в десятичной форме. **Decimal** в этом случае уместнее.

Для сравнения чисел с плавающей точкой рекомендуется использование **isEqual**.

Строки

При работе со строками старайтесь избегать магических символов и строк. Рассмотрим код, который можно с помощью этого правила сделать не только понятнее, но и сократить его логику работы:

```swift
func someFunc() {
    var result = parse(data: "1,2,3,4")
    print(result)

    result = parse(data: "a;b;c;d")
    print(result)
}

func parse(data: String) -> [String] {
    if data.range(of: ";") != nil {
        return data.components(separatedBy: ";")
    } else if data.range(of: ",") != nil {
        return data.components(separatedBy: ",")
    } else if data.range(of: ":") != nil {
        return data.components(separatedBy: ":")
```

```
    }
    return [String]()
}

someFunc()
// ["1", "2", "3", "4"]
// ["a", "b", "c", "d"]
```

После преобразования:

```
func someFunc() {
    var result = parse(data: "1,2,3,4")
    print(result)

    result = parse(data: "a;b;c;d")
    print(result)
}

func parse(data: String) -> [String] {
    let separators = [";", ":", ","]
    for separator in separators {
        if data.range(of: separator) != nil {
            return data.components(separatedBy: separator)
        }
    }
    return [String]()
}

someFunc()
// ["1", "2", "3", "4"]
// ["a", "b", "c", "d"]
```

При работе со строковыми данными обращайте внимание на кодировки. Особенно — при работе с файлами, данные в которых могут быть представлены в различной кодировке. Для Swift рекомендована UTF-8.

```
let buffer = "Какой-то текст"
if let data = buffer.data(using: .utf8) {

    /*
     Манипуляции с кодом,
     В которых мы забыли о кодировке
     */

    let encBuffer = String(data: data, encoding: .ascii)

    print("Buffer: \(buffer)")
    print("Encoding buffer: \(encBuffer ?? "")")
}

// Buffer: Какой-то текст
// Encoding buffer: ÐÐ°Ð°Ð¾Ð¹-ÑÐ¾ ÑÐµÐ°ÑÑ
```

Логические переменные

Рекомендуется вместо проверок условий в **if** присваивать результат проверки в переменную. Дадим ей понятное имя — сделаем код яснее.

```
func parse(data: String) -> [String] {
    let separators = [";", ":", ","]
    for separator in separators {

        let isExistsSeparator = data.range(of: separator) != nil

        /*
         Код
         */

        if isExistsSeparator {
            return data.components(separatedBy: separator)
        }
    }
    return [String]()
}
```

В этом примере переменная рассчитывается раньше, чем будет использоваться. «Код» может привести к раннему выходу из тела функции, и переменная **isExistsSeparator** может вовсе не понадобиться.

Есть способ отложить расчет данной переменной:

```
func parse(data: String) -> [String] {
    let separators = [";", ":", ","]
    for separator in separators {

        var isExistsSeparator: Bool {
            return (data.range(of: separator) != nil)
        }

        /*
         Код
         */

        if isExistsSeparator {
            return data.components(separatedBy: separator)
        }
    }
    return [String]()
}
```

Старайтесь разбивать сложные условия на несколько логических переменных. Преобразуем пример кода ниже в соответствии с этим правилом:

```
enum DefaultAuthData {
    static let login = "login"
    static let password = "password"
}

func auth(login: String, password: String) -> Bool {
    if (login == DefaultAuthData.login) && (password ==
DefaultAuthData.password) {
        return true
    } else {
        return false
    }
}
```

Применим правило:

```
enum DefaultAuthData {
    static let login = "login"
    static let password = "password"
}

func auth(login: String, password: String) -> Bool {
    var isValidLogin: Bool = {
        return login == DefaultAuthData.login
    }
    var isValidPassword: Bool = {
        return password == DefaultAuthData.password
    }
    if isValidLogin && isValidPassword {
        return true
    } else {
        return false
    }
}
```

Собственные типы данных (typealias)

Собственные типы данных задают альтернативное имя для существующего типа, а сами задаются с помощью ключевого слова **typealias**.

```
typealias <# type name #> = <# type expression #>

typealias Buffer = String
typealias Separator = String
typealias SeparatorHeight = CGFloat
```

Не бойтесь создавать псевдонимы для типов. Они полезны, когда вы хотите обратиться к существующему типу по имени, которое больше подходит по смыслу.

```
typealias Data = String
typealias Separator = String
typealias IgnoreValue = String
```

```
typealias IsUpper = Bool

func parse(data: Data, separator: Separator, ignoreValue: IgnoreValue, isUpper:
IsUpper) -> [String] {
    var result = [String]()
    if data.range(of: separator) != nil {
        for item in data.components(separatedBy: separator) {
            if item != ignoreValue {
                result.append(isUpper ? item.uppercased() : item)
            }
        }
    }
    return result
}


func someFunc() {
    let result = parse(data: "a,b,c,d", separator: ",", ignoreValue: "a",
isUpper: true)
    print(result)
}

someFunc()
// ["B", "C", "D"]
```

Пример использования псевдонимов типов данных — замыкания. Представленная выше функция **parse** может быть замыканием и без использования **typealias**. Ее код выглядел бы так:

```
func someFunc(parseClosure: (_ name: String, String, String, Bool) -> [String])
{

}
```

Но применяя подобное замыкание, можно ошибочно вставить значения не в те входные параметры. Нет подсказки от XCode — даже именование параметров в полной мере не поддерживается.

С применением псевдонимов все встает на свои места.

```
typealias Data = String
typealias Separator = String
typealias IgnoreValue = String
typealias IsUpper = Bool

func someFunc(parseClosure: (Data, Separator, IgnoreValue, IsUpper) -> [String])
{
```

```
}
```

```
func someFunc(parseClosure: (Data, Separator, IgnoreValue, IsUpper) -> [String]) {

}

func configure () {
    someFunc(
}     Void (parseClosure: (Data, Separator, IgnoreValue, IsUpper) -> [String])
```

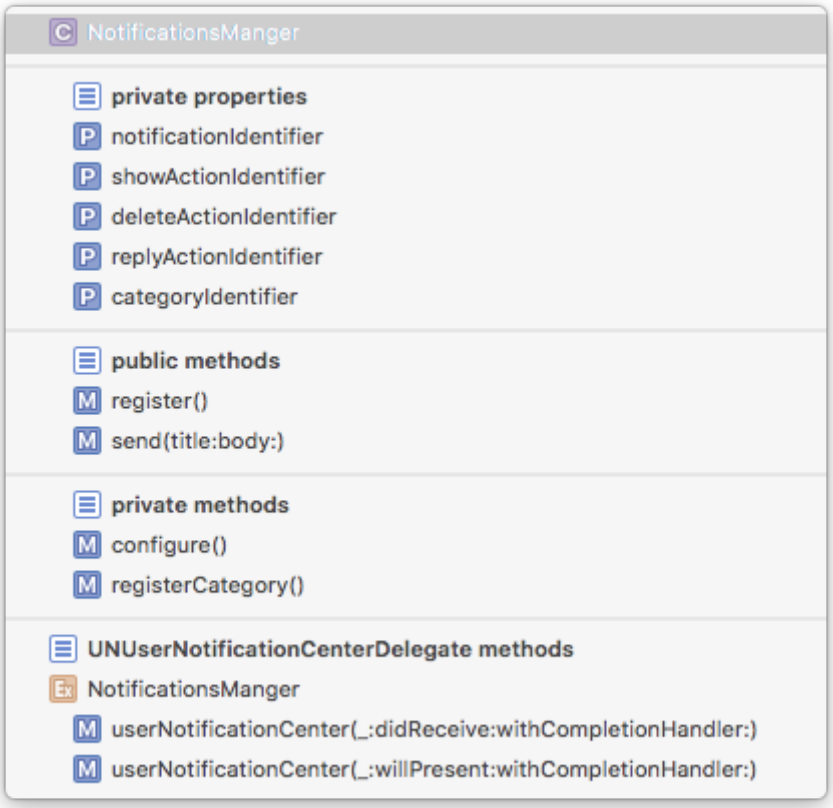
Организация методов

Старайтесь писать методы, да и код в целом, сверху вниз — аналогично их выполнению, и группировать взаимосвязанные методы. Такое объединение неплохо дополнять комментарием MARK — это организует дополнительную информацию о вашем коде на панели быстрого перехода.

```
import UIKit
import UserNotifications

class NotificationsManger: NSObject {

    // MARK: - private properties

    private let notificationIdentifier = "Notification"
    private let showActionIdentifier = "ShowAction"
    private let deleteActionIdentifier = "DeleteAction"
    private let replyActionIdentifier = "ReplyAction"
    private let categoryIdentifier = "MyCategory"

    // MARK: - public methods

    func register() {
        UNUserNotificationCenter.current().getNotificationSettings { [unowned
self] (settings) in
            if settings.authorizationStatus != .authorized {
                UNUserNotificationCenter.current().requestAuthorization(options:
[.alert, .sound, .badge]) { [unowned self] (granted, error) in
                    guard granted else {
                        print("requestAuthorization() error:
\(error?.localizedDescription ?? "")")
                        return
                    }
                    self.configure()
                }
            } else {
                self.configure()
            }
        }
    }

    func send(title: String, body: String) {
        let trigger = UNTimeIntervalNotificationTrigger.init(timeInterval: 5,
```

```
repeats: false)

        let content = UNMutableNotificationContent()
        content.title = title
        content.body = body

        let request = UNNotificationRequest(identifier: notificationIdentifier,
content: content, trigger: trigger)
        UNUserNotificationCenter.current().add(request) {(error) in
            if let error = error {
                print("Send error - \(error)")
            }
        }

    }

    // MARK: - private methods

    private func configure() {
        UNUserNotificationCenter.current().delegate = self
        registerCategory()
    }

    private func registerCategory() {
        let replyAction = UNTextInputNotificationAction(identifier:
replyActionIdentifier,
                                                        title: "Ответ",
                                                        options: [],
                                                        textInputButtonTitle:
"Отправить",
                                                        textInputPlaceholder:
"Ваше сообщение")
        let category = UNNotificationCategory(identifier: categoryIdentifier,
                                              actions: [replyAction, showAction,
deleteAction],
                                              intentIdentifiers: [],
                                              options: [])
        UNUserNotificationCenter.current().setNotificationCategories([category])
    }
}

// MARK: - UNUserNotificationCenterDelegate methods
extension NotificationsManger: UNUserNotificationCenterDelegate {
    func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive
response: UNNotificationResponse, withCompletionHandler completionHandler:
@escaping () -> Void) {
        let userInfo = response.notification.request.content.userInfo
        print("Recieve notification:\n \(userInfo)")
        completionHandler()
    }

    func userNotificationCenter(_ center: UNUserNotificationCenter, willPresent
notification: UNNotification, withCompletionHandler completionHandler: @escaping
(UNNotificationPresentationOptions) -> Void) {
        completionHandler([.alert, .sound])
    }
}
```

Результат работы комментария MARK:



Если вы вызываете несколько методов и для них важен порядок, используйте входные и возвращаемые аргументы. Приведенный выше метод **register** класса **NotificationsManger** мог выглядеть так:

```
class NotificationsManger: NSObject {

    var isRegisterNotifications = false

    func register() {
        registerNotifications()
        configure()
    }

    private func registerNotifications() {
        /*
         Код
         */
        isRegisterNotifications = true
    }

    private func configure() {
        if isRegisterNotifications {
            UNUserNotificationCenter.current().delegate = self
            registerCategory()
        }
    }
}
```

Как видим, метод **configure()** полностью зависит от работы **registerNotifications()**. Но эта зависимость зафиксирована только в порядке их вызова. После манипуляций код пришел в такой вид:

```
class NotificationsManger: NSObject {

    var isRegisterNotifications = false

    func register() {
        configure()
        registerNotifications()
    }

    private func registerNotifications() {
        /*
         Код
         */
        isRegisterNotifications = true
    }

    private func configure() {
        if isRegisterNotifications {
            UNUserNotificationCenter.current().delegate = self
            registerCategory()
        }
    }
}
```

Изменился порядок вызова методов **configure()** и **registerNotifications()**. Код метода **configure()** никогда не будет выполнен. В этом случае мы не в силах повлиять на изменение кода. Но если допишем наши методы в соответствии с озвученным правилом, подобной ситуации просто не случится.

```
class NotificationsManger: NSObject {

    func register() {
        if registerNotifications() {
            configure()
        }
    }

    private func registerNotifications() -> Bool {
        /*
         Код
         */
        return true
    }

    private func configure() {
        UNUserNotificationCenter.current().delegate = self
        registerCategory()
    }
}
```

Циклы

Вспомним работу в Swift с циклами. В языке присутствуют знакомые операторы управления циклами: **for-in** и **while**. Оба направлены на многократное выполнение задач (итераций). Используйте **while**, когда неизвестно, сколько итераций необходимо выполнить. Например, при чтении из файла с неизвестным количеством строк. Применяйте **for-in**, когда известно количество итераций или необходимо перебрать массив или словарь. Весь код подготовки к циклу размещайте непосредственно перед ним.

Старайтесь избегать раннего выхода — не изменять последовательности выполнения кода, передавая управление от одного фрагмента другому. Нежелателен и пропуск итераций — это риск скрытых ошибок. В Swift ранний выход представлен операторами передачи управления. Основные — **continue** и **break**. Если приходится их использовать, старайтесь выполнять этот код в начале тела цикла.

```
func parse(data: Data, separator: String, ignoreValue: String, isUpper: Bool) ->
[String] {
    let result = [String]()
    if data.range(of: separator) != nil {
        for item in data.components(separatedBy: separator) {
            if item == ignoreValue {
                continue
            }

            /*
             Какой-то код
             */
```

```
        }
    }
    return result
}
```

Табличные методы

Табличные методы — замена логическим условиям. Подробно эта тема освещена в книге «Совершенный код» Стива Макконнелла.

Рассмотрим пример использования табличных методов:

```
func getDaysCount(month: Int) -> Int {
    if month == 1 {
        return 31
    } else if month == 2 {
        return 28
    } else if month == 3 {
        return 31
    } else if month == 4 {
        return 30
    } else if month == 5 {
        return 31
    } else if month == 6 {
        return 30
    } else if month == 7 {
        return 31
    } else if month == 8 {
        return 31
    } else if month == 9 {
        return 30
    } else if month == 10 {
        return 31
    } else if month == 11 {
        return 30
    } else if month == 12 {
        return 31
    }
    return 0
}
```

Подобного вида гигантские функции элегантно упрощаются при использовании табличных методов:

```
func getDaysCount(month: Int) -> Int {
    if 0...11 ~= month {
        let daysCountInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
        return daysCountInMonth[month];
    } else {
        return 0
    }
}
```

Используйте такие словари для связывания операций с условием их выполнения. Помните, что можно вкладывать одни словари в другие, чтобы иметь к данным доступ в два шага вместо сложного поиска.

Практическое задание

1. Реализовать добавление товара в корзину.
2. Реализовать удаление товара из корзины.
3. Реализовать оплату заказа.

Дополнительные материалы

1. typealias.
2. Стив Макконнелл. Совершенный код.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. typealias.
2. Стив Макконнелл. Совершенный код.