



Урок 5

Code-style и качественный код

Правила чистого кода. Линтеры.

[Чистый код](#)

[Признаки плохого кода](#)

[Переменные](#)

[Время связывания переменной](#)

[Имена переменных](#)

[Форматирование](#)

[Самодокументируемый код](#)

[Комментарии](#)

[Линтеры](#)

[SwiftLint](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Чистый код

Программный код пишется людьми и для людей. Компилятору все равно, как выглядит код — он корректно его соберет, и на выходе получится программа, вполне понятная пользователям. Но они не видят код, который мы создаем, зато его рассматриваем мы, программисты. Поэтому он должен быть понятен именно нам. Понятность — критерий сугубо человеческий. Только человек может из названия метода представить себе, что именно тот делает. Программист читает название переменной и сразу видит суть хранимых в ней данных. Особо остро проблема чистого кода проявляется в больших проектах, объединяющих команду программистов, разделенных в офисе и территориально. Требования к качеству кода в таких условиях возрастают в разы.

Существует множество известных правил чистого кода и книг, разъясняющих их. Пример — известный труд Стива Макконнелла «Совершенный код». Об этих правилах рано или поздно узнает практически любой программист. Ведь программный код — это реализация его идей, направленных на решение задач (будь то создание игровых приложений или корпоративного софта). Важно изначально принять правила чистого кода и применять их. Такой подход будет работать на программиста тем лучше, чем больших масштабов будет достигать его проект.

Основные признаки чистого кода:

- **Код, который легко понять.** Этому способствуют логичное именование переменных, методов и других сущностей, хороший стиль интерфейса и реализации.
- **Код, который сам рассказывает, что он делает.** Не надо куда-то смотреть, кроме кода, или переходить, чтобы узнать тип и переменные. Смотришь на код — и понимаешь, как он работает.
- **Код, на который приятно смотреть.** Все мы немного эстеты. И приятнее разбираться в хорошо организованном и структурированном коде.
- **Код, который легко менять.** Нужно упрощать процесс внесения дальнейших изменений. Так легче добавлять новые свойства, повышать быстродействие и изменять архитектуру.

Признаки плохого кода

Исходя из основных качеств чистого кода, выделим признаки плохого:

- **Если нельзя с первого взгляда понять, что делает код.** Это первопричина долгого встраивания новой функциональности (модификации) в уже существующий код. Точнее — долгого определения самого места встраивания. Предположим, вы — новый сотрудник, которому поручена подобная задача. Проект большой. Код сложный. И чтобы понять, что делает тот или иной файл или класс, который соответствует этому критерию, нужно затратить значительное время. А если представить, что таких мест для анализа множество, встраивание новой функциональности может стать тихим ужасом. Тогда внести изменения под силу только автору данной функциональности. Например, простая функция возврата последнего элемента в строковом массиве имеет не только непонятное название, но и вызывает сложность понимания ее реализации:

```
func element(array: [String]?) -> String {
    var x = ""
    if let a = array {
        for i in a {
            x = i
        }
    }
    return x
}
```

Лучше:

```
func last(array: [String]?) -> String {
    var result = ""
    if let array = array {
        for item in array {
            result = item
        }
    }
    return result
}
```

- **Очень просто внести дефект.** В непонятном коде со «вторым дном» легко вынести неправильное суждение и принять неверное решение. На его исправление потребуется время.

```
func element(array: [String]?) -> String {
    var x = ""
    if let a = array {
        for i in a {
            x = i
        }
    }
    return x
}

// Ошибочно предположили, что функция element() выводит первый элемент
// И в поле с именем ошибочно вывели отчество.
let firstElement = element(array: ["Иван", "Иванов", "Иванович"])
nameLabel.text = "Имя - \(firstElement)"
```

- **Много лишнего кода, без которого не обойтись.** Уменьшая количество кода, используемого для реализации нужного нам функционала, делаем его не только гораздо более читаемым и надежным, но и легким в модификации. Лишний код мешает этому — лежит мертвым грузом, требуя писать дополнительные строки, обслуживающие его.

Вместо:

```
func last(array: [String]?) -> String {
    var result = ""
    if let array = array {
        for item in array {
            result = item
        }
    }
    return result
}
```

Лучше:

```
func last(array: [String]?) -> String {
    if let array = array, let result = array.last {
        return result
    } else {
        return ""
    }
}
```

- **Непонятные имена.** Опасайтесь бессмысленных именований (например, **data_cv**, **NewSuperPuperMethod**, **oldClass**). Это касается не только переменных, но и методов, классов, файлов. Подобные наименования только мешают другим участникам команды. Не спасают даже многочисленные комментарии, так как при встрече с такими названиями нужно перейти в место их объявления, прочесть сопутствующие комментарии и попытаться понять, что имел в виду автор. Это трата времени разработки. Правильное название позволяет избавиться от комментариев и прояснить цели и предназначение компонентов. Получится самодокументируемый код — в котором переменные и функции именуются так, что при взгляде на код понятно, как он работает.

Поэтому всегда давайте «говорящие» имена:

```
override func viewDidLoad() {
    super.viewDidLoad()

    //oldConfigureAction()
    newConfigureAction()
    start()
}
```

Лучше:

```
override func viewDidLoad() {
    super.viewDidLoad()

    configureActionUIComponents()
    startAnimation()
}
```

- **Разный стиль кода в участках программы.** Разный стиль кода усложняет навигацию и его осмысление. Проходя по таким разношерстным фрагментам, будто попадаешь в другую вселенную, где действуют иные законы и правила, на понимание которых потребуется время. На крупных предприятиях чаще всего есть свод правил оформления кода в проекте

(соглашение по Code Style). И это не прихоть работодателя, а признак серьезного подхода к разработке.

Рассмотрим пример смешения различных стилей кода:

```
func configureUI() {
    self.title = "Стикеры"

    switch style {
    case .black:
        self.view.backgroundColor = UIColor.black
        self.separatorView.backgroundColor = UIColor.lightGray
    case .blue:
        self.view.backgroundColor = UIColor.blue
        self.separatorView.backgroundColor = UIColor.white
    }
}

func load_stickers()
{
    for i in 1...5
    {
        if let url = Bundle.main.url(forResource: "image\(i)", withExtension:
"png")
        {
            do
            {
                let sticker = try MSSticker(contentsOfFileURL: url,
localizedDescription: "")
                stickers.append(sticker)
            }
            catch
            {
                print(error)
            }
        }
    }
}
```

Как видим, функции **configureUI** и **load_stickers** имеют не только разные стили написания составных слов (**CamelCase** и **snake_case**), но и разные варианты группировки при помощи скобок. И ориентироваться в таком коде очень сложно.

Лучше:

```
override func viewDidLoad() {
    super.viewDidLoad()

    configureUI()
    loadStickers()
}

func configureUI() {
    self.title = "Стикеры"

    switch style {
    case .black:
        self.view.backgroundColor = UIColor.black
```

```

        self.separatorView.backgroundColor = UIColor.lightGray
    case .blue:
        self.view.backgroundColor = UIColor.blue
        self.separatorView.backgroundColor = UIColor.white
    }
}

func loadStickers() {
    for i in 1...5 {
        if let url = Bundle.main.url(forResource: "image\(i)", withExtension:
"png") {
            do {
                let sticker = try MSSticker(contentsOfFileURL: url,
localizedDescription: "")
                stickers.append(sticker)
            } catch {
                print(error)
            }
        }
    }
}
}

```

Переменные

Мы постоянно имеем дело с переменными. Настало время разобраться, какие требования выставляются к переменным по принципам чистого кода.

Старайтесь всегда объявлять все переменные как константы (`let`). Любая возможность изменить переменную — это потенциальный риск совершить ошибку, которой не случилось бы, объяви вы переменную как константу.

```

func getContext(dictionary: [String: String]?) -> String {
    let result: String
    if let dictionary = dictionary, let value = dictionary["context"] {
        result = value
    } else {
        result = ""
    }
    return result
}

```

Но не нужно фанатично создавать все константами. Просто посмотрите на код и определите — не совершаете ли вы изменение, без которого можно обойтись. Если это так — сделайте эту переменную `let`. Также можно применить такой подход: все переменные объявляем как `let` и далее, если потребуется изменить какую-нибудь переменную и мы точно осознаем, что без изменения никак, переделываем ее в `var`.

```

class ViewController: UIViewController {

    @IBOutlet weak private var loginTextField: UITextField!

    var login = ""

    @IBAction func loginButtonTap(_ sender: Any) {

```

```

        login = loginTextField.text ?? ""
    }
}

```

Объявляем переменную как можно ближе к тому месту, где она будет использоваться. Если этого не сделать, переменная может оказаться вовсе не нужной.

```

func loadCityNames() -> [String] {

    var result = [String]()

    guard let realm = try? Realm() else { return [String]() }

    let cities = realm.objects(City.self)
    for city in cities {
        result.append(city.name)
    }

    return result
}

```

В приведенном выше примере переменная **result** может никогда и не понадобиться, если при создании объекта **Realm** возникнут исключения.

Если переменная расположена далеко от места ее применения, есть риск совершить с ней ошибочные действия (например, присвоить `nil` опциональной переменной). В результате к моменту прихода к блоку ее непосредственного использования переменная модифицировалась, и код применения отработал некорректно.

Если есть вероятность, что переменная получит неверное значение к моменту использования, проверяйте ее корректность перед этим. Например, в уже использованной нами функции загрузки городов перед возвратом из функции захотим сделать ее сортировку в отдельном методе, который будет возвращать опционал. И совершим в нем ошибку. Но так как метод **loadCityNames()** возвращает не опционал, — предположим, что мы были на 100% уверены в функции и сделали принудительное извлечение (!) — получим runtime-ошибку.

```

func loadCityNames() -> [String] {
    guard let realm = try? Realm() else { return [String]() }

    var result = [String]()
    let cities = realm.objects(City.self)
    for city in cities {
        result.append(city.name)
    }

    result = sortCityNames(names: result)!

    return result
}

func sortCityNames(names: [String]) -> [String]? {
    return nil
}

```


Этого не случилось бы, если бы сделали дополнительную проверку.

```
func loadCityNames() -> [String] {
    guard let realm = try? Realm() else { return [String]() }

    var result = [String]()
    let cities = realm.objects(City.self)
    for city in cities {
        result.append(city.name)
    }

    if let sortResult = sortCityNames(names: result) {
        result = sortResult
    }

    return result
}

func sortCityNames(names: [String]) -> [String]? {
    return nil
}
```

Время жизни переменной делаем как можно короче, грамотно выбирая области видимости. Под каждую переменную выделяется область памяти. И если переменная живет ненужное время, это ведет к потере памяти. Кроме того, чем дольше живет переменная, тем сложнее с ней работать. Большее время жизни — это более объемный контекст, в котором используется переменная. При работе с ней его придется держать в голове. Работать с такими переменными тяжелее, а риск допустить ошибку выше. Так что при любой возможности старайтесь уменьшать область видимости.

Например, реализация метода загрузки списка городов могла бы выглядеть так:

```
class DataManager {

    var cityNames = [String]()

    func loadCityNames() {
        guard let realm = try? Realm() else { return }

        let cities = realm.objects(City.self)
        for city in cities {
            cityNames.append(city.name)
        }
        return
    }
}
```

Но далее по коду метод **loadCityNames()** использовался бы только для получения имени первого города с последующим его отображением — и все. А переменная все еще висит в памяти и занимает драгоценный ресурс. Всегда грамотно определяйте время жизни переменной.

Переменные должны всегда преследовать только одну цель и не иметь скрытого смысла. Вспомним пример с сохранением логина — код мог бы иметь плохой стиль:

```
class ViewController: UIViewController {

    @IBOutlet weak private var loginTextField: UITextField!
    @IBOutlet weak private var passwordTextField: UITextField!
```

```

var inputData = ""

// ...
// множество кода
// ...

@IBAction func loginButtonTap(_ sender: Any) {
    inputData = loginTextField.text ?? ""
}
}

```

Что скрывает в себе переменная **inputData**? Логин или пароль? Без детального просмотра и анализа метода **loginButtonTap** невозможно это понять.

Время связывания переменной

Время связывания — это момент, когда переменная и ее значение связываются вместе. Существует несколько вариантов времени связывания:

- **Во время написания кода** — например, обычные строковые литералы. Это вариант раннего связывания.

```
private let baseUrl = "https://api.vk.com/method/"
```

- **Во время компиляции** — например, вычисляемые переменные.

```
private let baseUrl = "https://api.vk.com/method/"
var friendsUrl: String {
    return "\(baseUrl)/friends.get"
}

```

- **Во время загрузки** — это позднее связывание. Фактически, это процесс связывания вызова и тела внешней функции. Предоставляется внешними библиотеками, выполняется при компоновке, откладывается до момента загрузки.
- **Во время выполнения** — модифицируя предыдущий пример, предположим, что у нас есть функция, возвращающая **url** для получения друзей. И привязка результата работы этой функции с переменной **url** и будет связыванием во время выполнения.

```
let url = getFriendsUrl()
```

- **При создании объекта.**

```
let value = try JSONDecoder().decode(T.self, from: data)
```

- **По требованию** — например, значение текстового поля, куда пользователь вбивает данные, которые нам нужны от пользователя, и мы не можем их прописать во время кодирования. Еще пример — ленивое свойство хранения (*lazy*). Свойство, начальное

значение которого не вычисляется до первого использования. Ленивые свойства полезны, когда их значение зависит от внешних факторов, значения которых неизвестны до окончания инициализации. Пример использования lazy-свойства — **persistentContainer** при подключении фреймворка **CoreData**.

Всегда желательно поднимать время связывания как можно выше — к моменту написания кода или к компиляции.

Имена переменных

Выпишем правила именования переменных:

- **Имя точно должно отражать суть и назначение переменной.** Переменные не должны называться в духе **k**, **i**, **m**. Если **i** — это индекс в цикле **for**, переменную лучше назвать **index**. Не должно быть сокращений в именах переменных. **ViewController** не надо называть **vk**, лучше **controller**. Если понадобится создать базовый класс **VK**, что это будет — базовый класс контроллера или одноименный сервис «ВКонтакте»?

Не создавайте рядом в коде переменные, похожие по смыслу, например **data** и **strData**, — в них легко запутаться и совершить ошибку. Лучше **data** и **body**.

```
Alamofire.request("https://xxx", method: .post, parameters: parameters).response
{ response in
    if let data = response.data, let strData = String(data: data, encoding:
.utf8) {
        print("String data: \(strData)")
    }
}
```

Применим правило:

```
Alamofire.request("https://xxx", method: .post, parameters: parameters).response
{ response in
    if let data = response.data, let body = String(data: data, encoding: .utf8)
{
        print("Response body: \(body)")
    }
}
```

- **При выборе нужно ориентироваться на проблему, а не на способ решения.** Надо именовать то, что будет в этой переменной, а не способ, как мы это получили.

```
class ViewController: UIViewController {

    @IBOutlet weak private var loginTextField: UITextField!
    @IBOutlet weak private var passwordTextField: UITextField!

    var login = ""

    // ...
    // множество кода
    // ...

    @IBAction func loginButtonTap(_ sender: Any) {
```

```

        login = loginTextField.text ?? ""
    }
}

```

- **Как правило, идеальное имя переменной содержит от 5 до 16 символов, но это не точно.** Это рекомендация, а фактически имя зависит от ваших реалий. Даже если возьмем стандартный пример с координатами и обозначим наши свойства **x** и **y** соответственно, все равно можно найти наименование лучше — **pointX** и **pointY**.
- **Чем меньше область видимости переменной, тем короче может быть имя.** Есть такое правило — чем меньше живет переменная, тем короче может быть ее имя. Наши названия **x** и **y** уместно использовать во внутренней функции расчета центра треугольника, но как внешние свойства они должны иметь более длинные наименования. Иначе их можно перепутать с координатами вершин треугольника:

```

class Triangle {
    var centerX: Int?
    var centerY: Int?

    func calculate() {
        guard let x = centerX, let y = centerY else {
            return
        }
        /*
         * Вычисления с 'x' и 'y'
         */
    }
}

```

- **Не бойтесь добавлять в имена переменных глаголы и предлоги.** В приведенной выше функции могла появиться переменная с именем **calculatedValue**, и этого не надо бояться:

```

func calculate() {
    guard let x = centerX, let y = centerY else {
        return
    }

    let calculatedValue = x * y

    /*
     * Вычисления с 'x' и 'y'
     */
}

```

- **Добавляйте к булевым переменным is, has и так далее.** Если может появиться переменная, проверяющая что-то, добавляем **is** (например, если треугольник прямоугольный). Если же переменная показывает, что содержится, добавляем префикс **has** (например, что треугольник содержит данную точку).

```

let isRightTriangle = true
let hasPoint = false

```

- **Константы и переменные, методы и функции принято писать с маленькой буквы.** Всегда обращайтесь на это внимание. Иначе подобные именованья могут вводить в заблуждение.

```
let isRightTriangle = true
var hasPoint = false

func calculate() { }
```

- **Типы (классы, структуры, перечисления) принято писать с большой буквы.**

```
class Triangle {
    var centerX: Int?
    var centerY: Int?
}

struct Location {
    let latitude: Double
    let longitude: Double
}

enum HTTPStatusCodes: Int {
    case Continue = 100
    case OK = 200
}
```

- **Использовать стиль CamelCase для написания составных слов.** CamelCase — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы начинается с заглавной буквы. Это правило только для iOS-разработчиков. На Ruby, Perl пишут в стиле **snake_case**.

Примеры CamelCase-написания: BackColor, ЧерныйЦвет, backColor, черныйЦвет, CamelCase, ВерблюжийРегистр.

Форматирование

Файл с кодом должен быть похож на печатную статью. Представьте, что вы читаете газету. В ней есть абзацы, колонки — логические блоки. Такие же законченные мысли должны формироваться и в приложении.

В начале должен быть заголовок (имя класса, комментарий), который говорит, что происходит в этом файле. Он не должен транслировать очевидные вещи, но давать четкое понятие о том, для чего предназначен код.

```
/*
Класс, описывающий геометрическую фигуру — треугольник.
Производит вычисления радиуса...
*/
class Triangle {

    var centerX: Int?
    var centerY: Int?
```

```

func calculate() {
    guard let x = centerX, let y = centerY else {
        return
    }

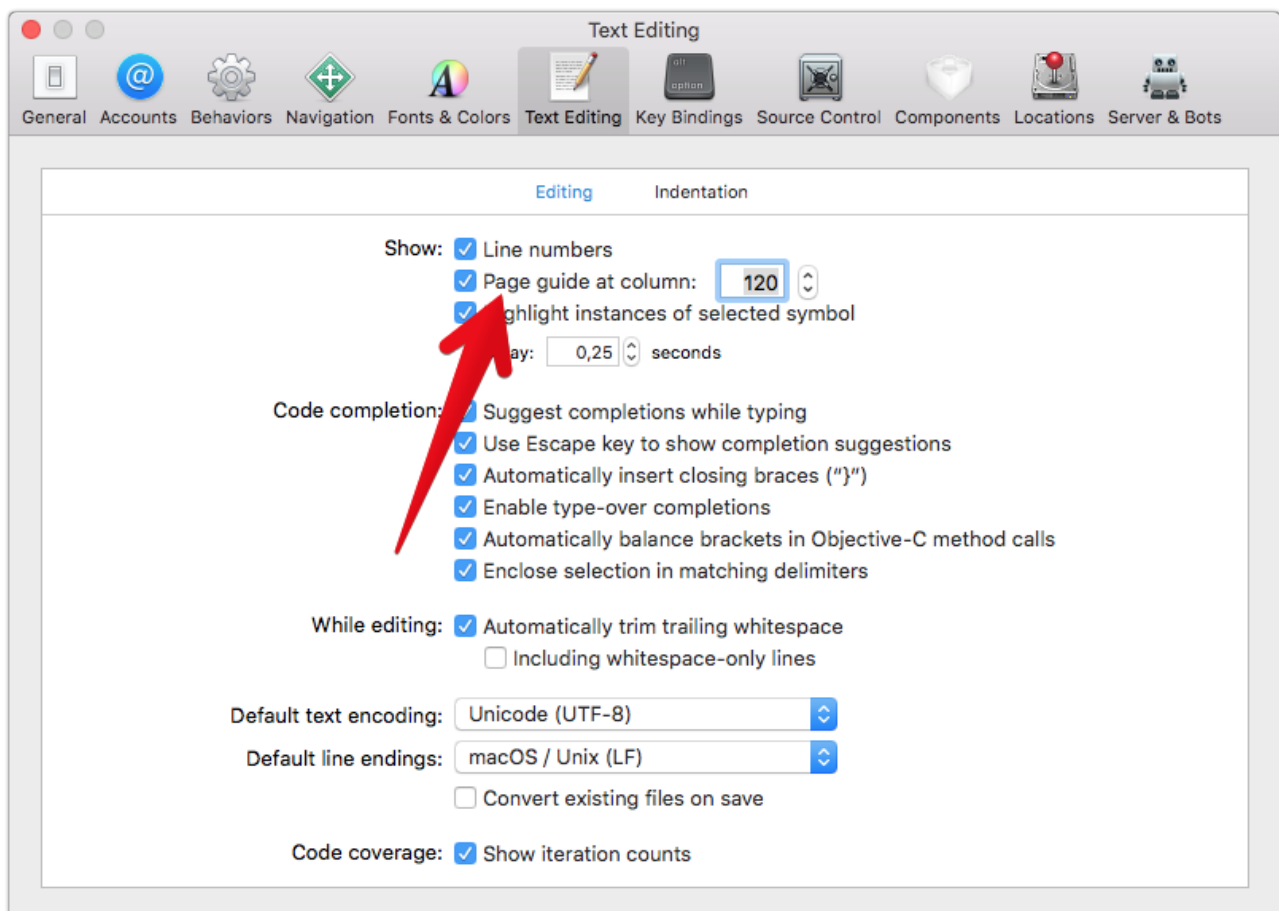
    let calculatedValue = x * y // Умножение координат (не нужно, это
очевидно!)

    /*
    Вычисления с 'x' и 'y'
    */
}
}

```

Для нас выделением мыслей являются пустые строки. Строка — это выражение. Группа строк — это законченная мысль (абзац). Пустые строки — мощный инструмент, который делает код более выразительным.

Строки не должны быть длиннее 120 символов. Это рекомендация, но ей не надо пренебрегать. Предположим, у вас хороший монитор, в который помещается 300 символов. А у коллег возможности могут быть хуже, и ваша строка в 300 символов у них будет резаться. Уважайте людей, которые работают на маленьких ноутбуках. XCode выделяет для этого дополнительную настройку — **Page guide at column:**



Следите за пробелами и скобками. Все должно быть выровнено. Не допускайте разных стилей.

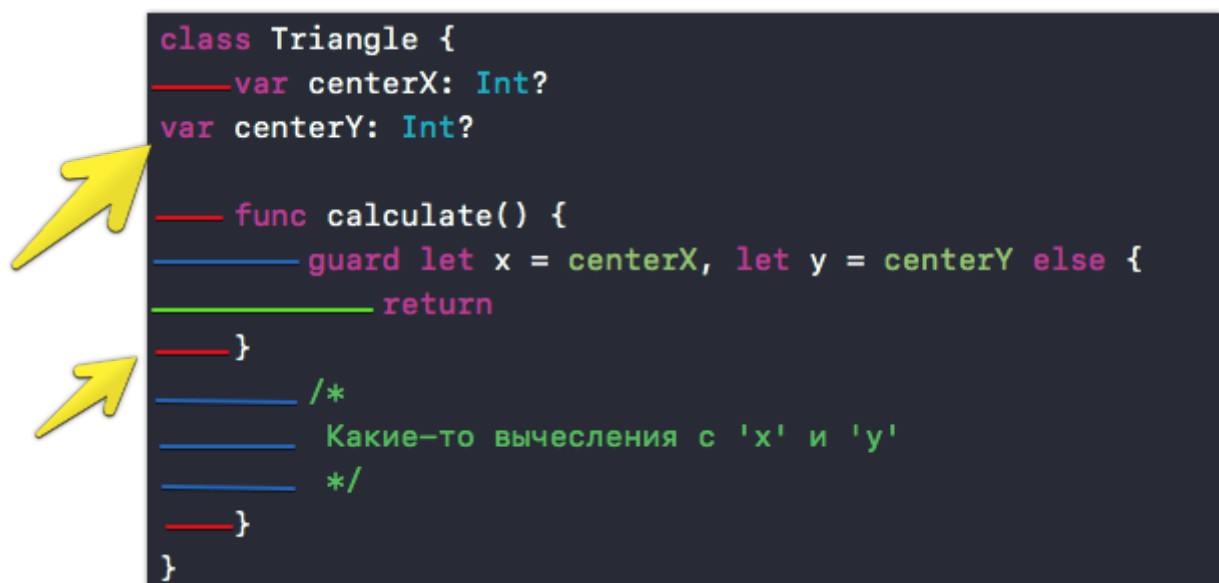
Плохой пример:

```
func calculate()
{
    if let x = centerX    {return}
    if let y = centerY{
        return}
    /*
    Какие-то вычисления с 'x' и 'y'
    */
}
```

Вставляйте пробелы для отделения скобок от кода. Открывающая скобка — на той же строке, что она открывает. Не вставляйте лишних пробелов.

Код одного уровня вложенности должен иметь одинаковый отступ.

Плохой пример:

The image shows a code snippet for a `Triangle` class. The code is as follows:

```
class Triangle {
    var centerX: Int?
    var centerY: Int?

    func calculate() {
        guard let x = centerX, let y = centerY else {
            return
        }
        /*
        Какие-то вычисления с 'x' и 'y'
        */
    }
}
```

Two yellow arrows point to the opening curly braces of the `calculate()` function and the `guard` statement. The first arrow points to the brace on the line `func calculate() {`, which is correctly aligned with the class opening brace. The second arrow points to the brace on the line `return`, which is indented further than the `guard` statement's opening brace, indicating an incorrect indentation level.

Здесь желтыми стрелками обозначены неправильные отступы. И наш код должен выглядеть следующим образом:

```
func calculate() {
    guard let x = centerX, let y = centerY else {
        return
    }

    /*
    Какие-то вычисления с 'x' и 'y'
    */
}
```

Самодокументируемый код

Самодокументируемый код — тот, в котором переменные и функции именуются таким образом, что при взгляде на код понятно, как он работает.

Основные принципы самодокументируемого кода:

- Следовать озвученным правилам, SOLID и другим принципам. Методы и классы должны следовать одной цели;
- Давать хорошие имена переменным, методам, классам;
- Делать небольшие методы;

Тогда код получается ясным и требует намного меньше документации.

Рекомендуется соблюдать баланс и разумно использовать и комментарии, и «говорящие» имена переменных, и возможности самодокументируемого кода — там, где это оправданно.

Код, приведенный ниже, не нуждается в документировании: из названия функции видно, какую работу он выполняет — находит последний элемент строкового массива с проверкой:

```
func last(array: [String]?) -> String {
    if let array = array, let result = array.last {
        return result
    } else {
        return ""
    }
}
```

А этот код очень нуждается в документировании:

```
// Функция нахождения последнего элемента строкового массива с проверкой
func element(array: [String]?) -> String {
    var x = ""
    if let a = array {
        for i in a {
            x = i
        }
    }
    return x
}
```

Комментарии

Комментарии — пояснения к исходному тексту программы, находящиеся непосредственно внутри кода.

Принципы написания комментариев:

- Комментарии должны быть полезными;

- Комментарии не должны пояснять очевидные вещи;
- Комментарии должны объяснять, не что выполняет код, а зачем он это делает;
- В комментариях стоит учитывать, что очевидное вам может не показаться таким другому человеку.

Пример плохого комментария:

```
// Контролер стикеров
class StickersController: MSMessagesAppViewController {
    ...
}
```

Пример хороших комментариев:

```
// Контролер выбора и отображения набора стикеров
// для расширения приложения 'Сообщения'
class StickersController: MSMessagesAppViewController {
    var stickers = [MSSticker]()

    override func viewDidLoad() {
        super.viewDidLoad()

        loadStickers()
        createStickerBrowser()
    }

    // Загрузка изображений стикеров из ресурсов проекта
    func loadStickers() {
        for i in 1...5 {
            if let url = Bundle.main.url(forResource: "image\(i)",
withExtension: "png") {
                do {
                    let sticker = try MSSticker(contentsOfFileURL: url,
localizedDescription: "")
                    stickers.append(sticker)
                } catch {
                    print(error)
                }
            }
        }
    }

    // Подготовка графических компонентов
    func createStickerBrowser() {
        let controller = MSStickerBrowserViewController(stickerSize: .large)
        addChildViewController(controller)
        view.addSubview(controller.view)

        controller.stickerBrowserView.backgroundColor = UIColor.green
        controller.stickerBrowserView.dataSource = self

        view.topAnchor.constraint(equalTo: controller.view.topAnchor).isActive =
true
        view.bottomAnchor.constraint(equalTo:
controller.view.bottomAnchor).isActive = true
        view.leftAnchor.constraint(equalTo: controller.view.leftAnchor).isActive
```

```
= true
    view.rightAnchor.constraint(equalTo:
controller.view.rightAnchor).isActive = true
    }
}
```

Линтеры

Зачастую небрежность в написании кода может повлечь серьезные последствия. А иногда — это всего лишь чересчур запутанный код. Некоторые из опечаток поначалу кажутся незначительными, но чем дальше растет проект, тем серьезней они становятся. Все больше разработчиков используют плохо написанный код, что заставляет их писать так же. Чтобы такой кошмар не нарастал снежным комом, на помощь приходят специальные утилиты, проверяющие код на соответствие гайдлайнам. Они позволяют отлавливать ошибки качества кода.

Один из ярких представителей линтеров для языка Swift — **SwiftLint**.

SwiftLint

SwiftLint — это утилита от разработчиков Realm для автоматической проверки качества Swift-кода. Утилита содержит набор правил, основанных на [Swift Style Guide](#) с возможностью добавления своего набора правил.

Утилита может быть установлена как через терминал с использованием утилиты **brew**, так и с помощью **cocoapods**.

SwiftLint встраивается непосредственно в XCode и проверяет код в конце каждой сборки проекта. После первого запуска можем наблюдать:



Это нормально, большинство ошибок и предупреждений достаточно просты и их легко исправить. В SwiftLint есть замечательная функция автокоррекции. Для ее использования в терминале заходим в каталог проекта и выполняем команду:

```
swiftlint autocorrect
```

После повторной сборки проекта количество ошибок заметно уменьшится, но оставшиеся придется исправлять вручную.

Со списком правил можно ознакомиться [по ссылке](#). Для просмотра предустановленных правил нужно выполнить команду в терминале:

```
swiftlint rules
```

Рекомендация: лучше включать правила по одному, а не все сразу. В крупных компаниях данный список дополняют своими пунктами.

Настройка SwiftLint осуществляется путем добавления файла **.swiftlint.yml** в корень проекта. Можно настроить следующие параметры:

Включение правила:

- **disabled_rules** — отключить правила из установленного по умолчанию набора;
- **opt_in_rules** — включить правила не по умолчанию;
- **whitelist_rules** — только правила, указанные в этом списке, будут включены.

Правила также можно отключать непосредственно в коде:

```
func printName() {  
    // swiftlint:disable force_cast  
    let name = loadName() as! String  
    // swiftlint:enable force_cast  
    print(name)  
}
```

Практическое задание

1. Реализовать получение списка отзывов о товаре.
2. Реализовать добавление отзыва о товаре.
3. Реализовать удаление отзыва.

Дополнительные материалы

1. [SwiftLint](#).

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [SwiftLint](#).
2. [Правила SwiftLint](#).
3. [Swift Style Guide](#).