

ОСНОВЫ языка Swift

Методичка к уроку 1

Синтаксис языка Swift





Оглавление

Термины, используемые в лекции	5
Введение в Swift	6
Создание playground-a	6
Константы и переменные	13
Основные типы данных	14
Статическая типизация	15
Вывод типов	15
Арифметические операторы	16
Составные выражение	18
Операторы сравнения	18
Булева алгебра	20
Операторы диапазона	21
Условный оператор if	22
Тернарный условный оператор	27
Конструкция switch	29
Guard	33
For-in	34
Where	35
While	36
Repeat-while	37
Операторы передачи управления	38
Заключение	40
Что можно почитать еще?	40
Используемая литература	40
Домашнее задание	40



Введение

Меня зовут Высоцкая Анастасия Алексеевна, опыт в программировании больше 5 лет, опыт разработки на языке Swift более 3 лет.

В рамках данного курса мы познакомимся с языком Swift, изучим его синтаксис и напишем свою первую программу, а также узнаем что такое перечисления, опционалы, классы, структуры и ARC.

В рамках данного урока мы изучим:

- Как написать свою первую программу на языке Swift и что для этого нужно
- Чем отличаются константы от переменных
- Какие есть типы данных в Swift
- Что такое статическая типизация и какая она еще бывает
- Что такое “Вывод типов” и как он упрощает жизнь
- Какие есть арифметические операции и как написать пять плюс семь на языке Swift
- Как на языке Swift сравнить два числа
- Что такое “Булева алгебра” и как она может пригодиться
- Что такое диапазоны, какие бывают и как задать их
- Как на языке Swift задать конструкцию “если, то”
- Что такое тернарный оператор и как он помогает уменьшить количество строк кода
- Что такое switch
- Что такое циклы и какие они бывают в языке Swift



Термины, используемые в лекции

Замкнутый диапазон - это диапазон значений от a до b , при условии, что a не превышает b , и a и b включены в диапазон.

Полузамкнутый диапазон - это диапазон значений от a до b , при условии, что a меньше b и включено только значение a .

Односторонний диапазон - это диапазон, который имеет ограничение только с одной стороны.

Тело цикла - действия, которые необходимо выполнить в цикле.



Введение в Swift

Swift - это язык программирования от Apple, с его помощью можно разрабатывать приложения под iOS, Apple TV, Mac, Apple Watch.

Компания Apple начала разрабатывать Swift в 2010 году. 2 июня 2014 года этот язык был официально представлен на всемирной конференции разработчиков на платформах Apple (WWDC). Бесплатное 500-страничное руководство по его использованию было доступно на сервисе iBook Store.

В Swift есть такие возможности, как

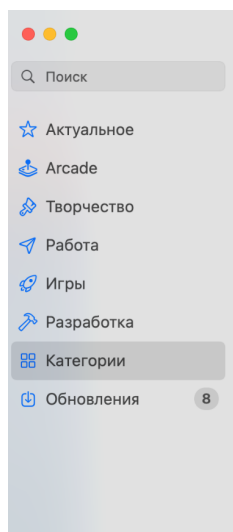
1. Кортежи, они группируют несколько значений в одно составное значение.
2. Дженерики – универсальные шаблоны
3. Протокол – описание свойств и методов без реализации.
4. Автоматический подсчет ссылок (ARC) используется в Swift для управления памятью приложения.
5. Вычисляемые свойства – свойства, которые не хранят какого-либо значения, но вычисляют его при обращении и могут обработать установку нового значения.

Создание playground-a

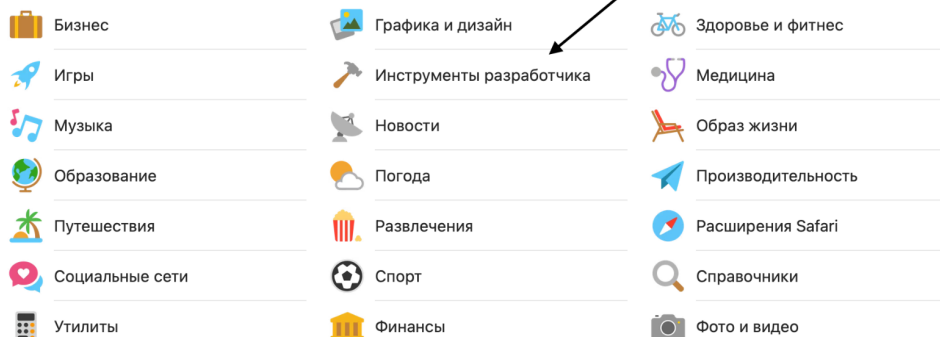
Мы узнали откуда взялся Swift и какие возможности может предоставить, пришло время написать свою первую программу, а именно вывести в консоль “Hello^world!” и первое, что нам необходимо сделать - это скачать Xcode

Xcode - это среда для разработки приложений, которую предоставляет Apple.

1. Откройте App Store
2. Перейдите в “Категории”, а затем в “Инструменты разработчика”



Категории



3. Выберите Xcode



1 Xcode

Инструменты разработчика

4. Нажмите “Загрузить”

5. После загрузки Xcode необходимо его открыть, появится следующее окно



×



Welcome to Xcode

Version 13.2.1 (13C100)



Create a new Xcode project

Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.



Clone an existing project

Start working on something from a Git repository.



Open a project or file

Open an existing project or file on your Mac.

1. Создать новый проект
2. Склонировать существующий проект
3. Открыть существующий проект или файл

Теперь мы можем перейти непосредственно к созданию своей первой программы, для этого создадим playground (или песочница).

Playground - это интерактивная среда разработки, где можно тестировать код и сразу видеть результат, “песочница” прекрасно подходит для обучения. Чтобы создать playground и написать свою первую программу необходимо:

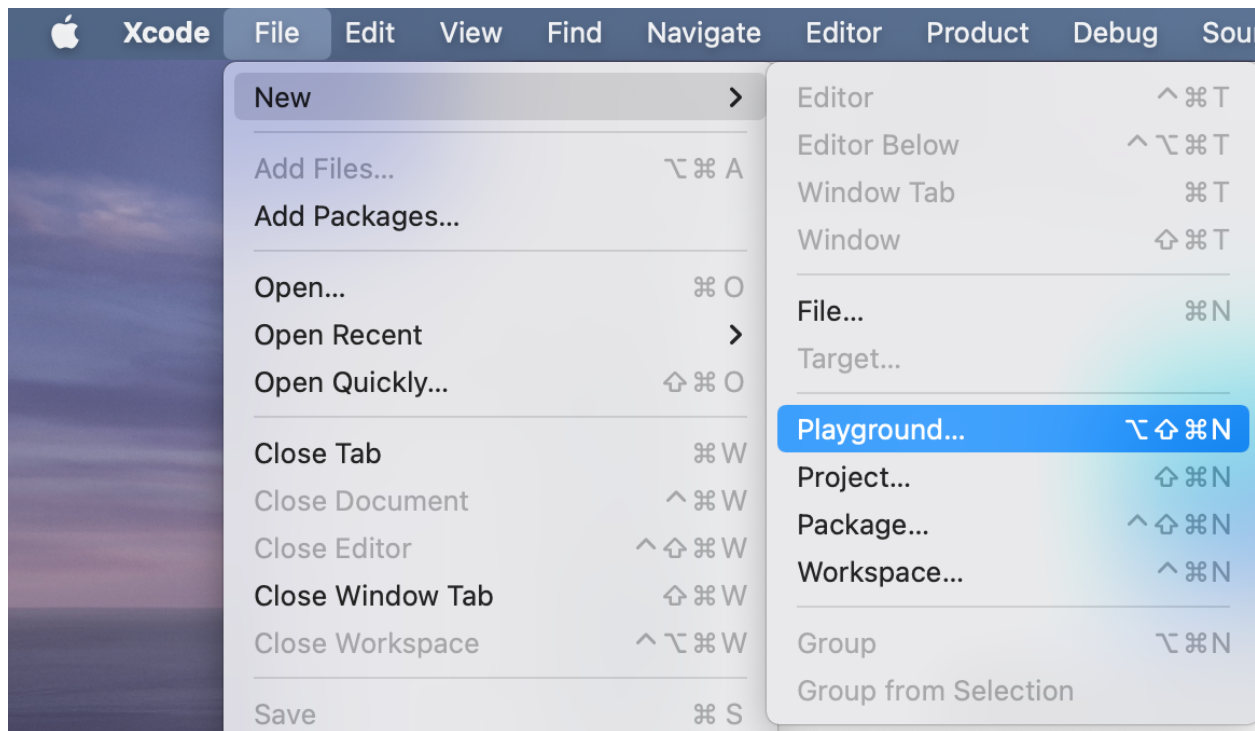
1. Открыть Xcode
2. В левом верхнем углу выбрать “File”



3. Выбрать “New”



4. Выбрать Playground



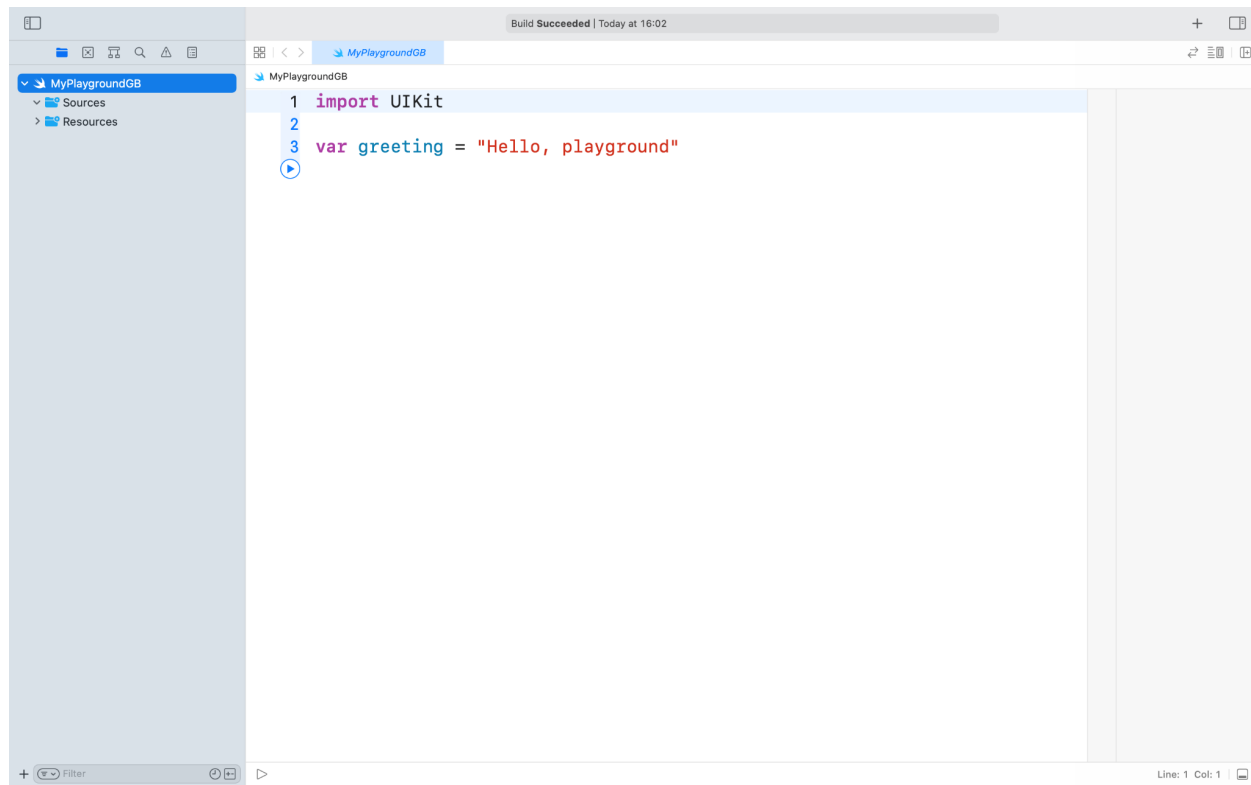
5. Выбрать “Blank” и нажать “Next”



6. Ввести название для файла и нажать “Create”

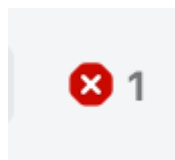


7. Появится окно песочницы с уже написанным кодом



В центре мы увидим основную область, в которой будет вестись разработка, справа выводятся значения переменных, внизу поле для вывода логов и ошибок.

В самом верху есть строка состояния, в которой будет отображаться текущий статус проекта, например, готов к сборке, собирается, запущен. Также в этой строке справа могут появляться восклицательный знак или крестик.



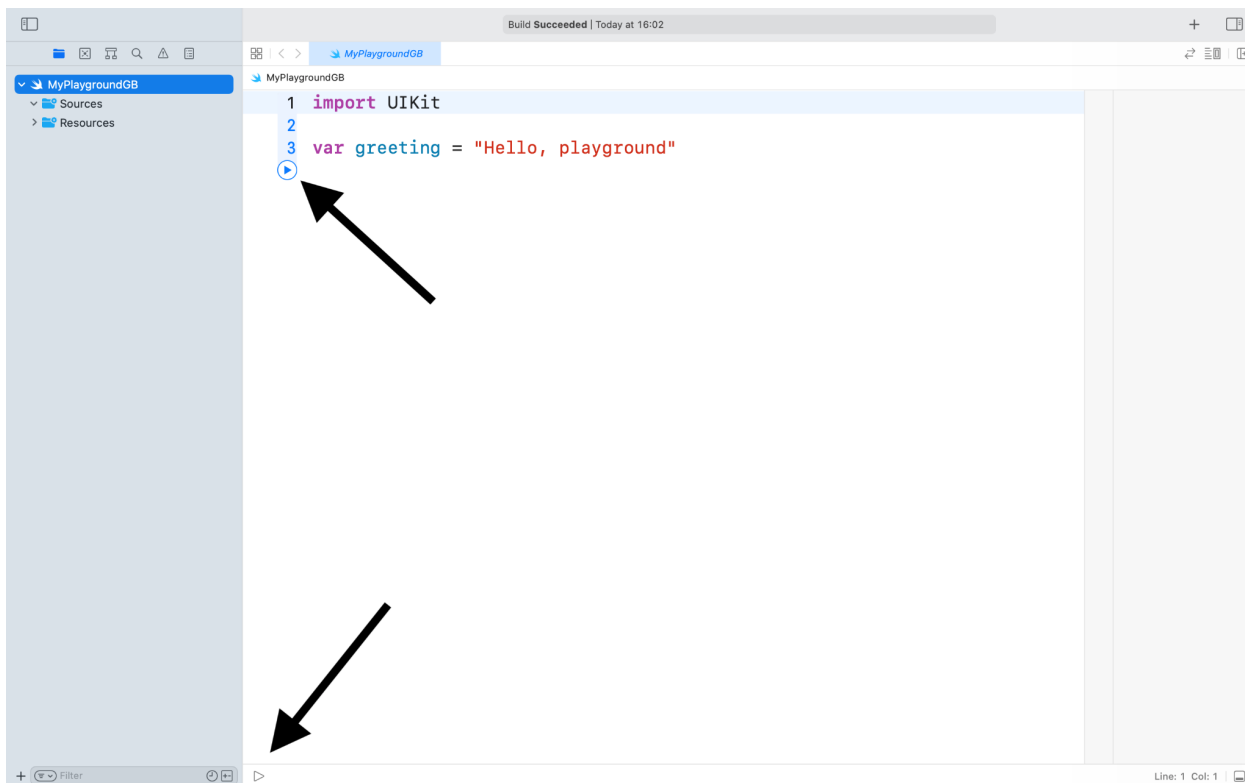
- ошибка, с ней проект не скомпилируется



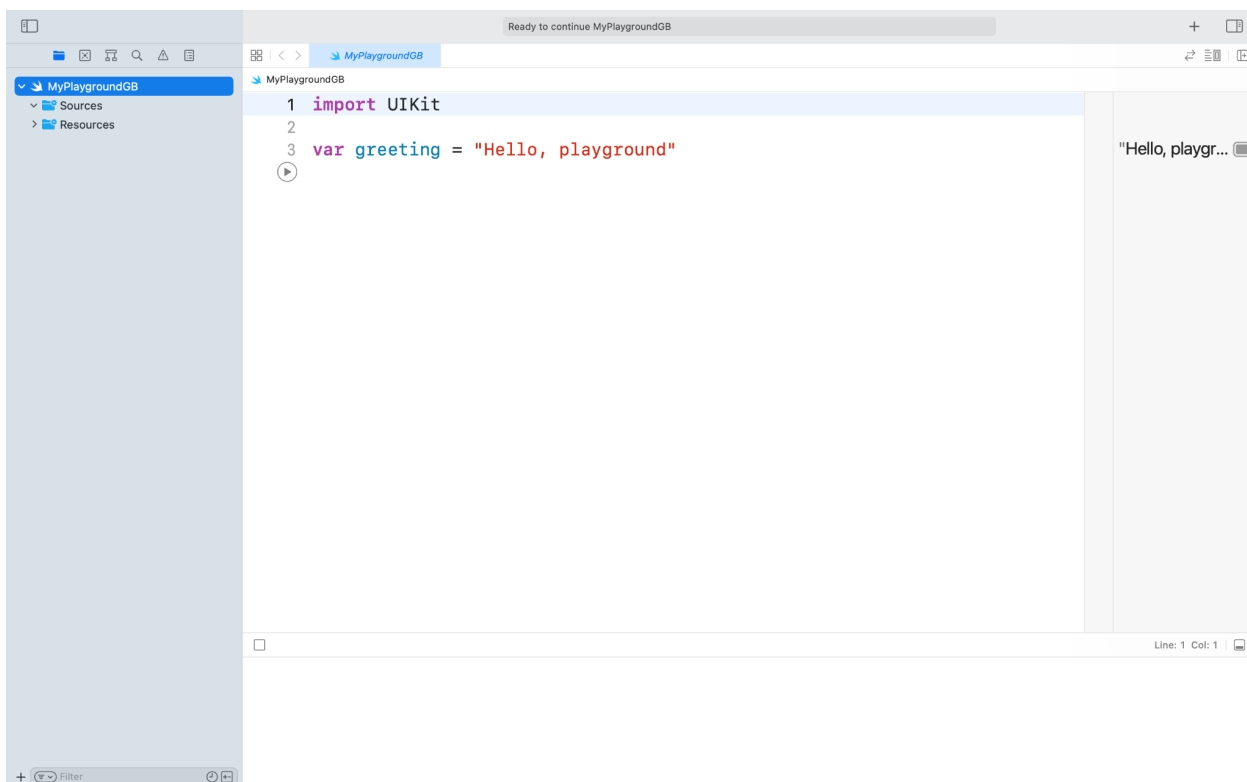
- предупреждение, что что-то не так, с ним проект скомпилируется



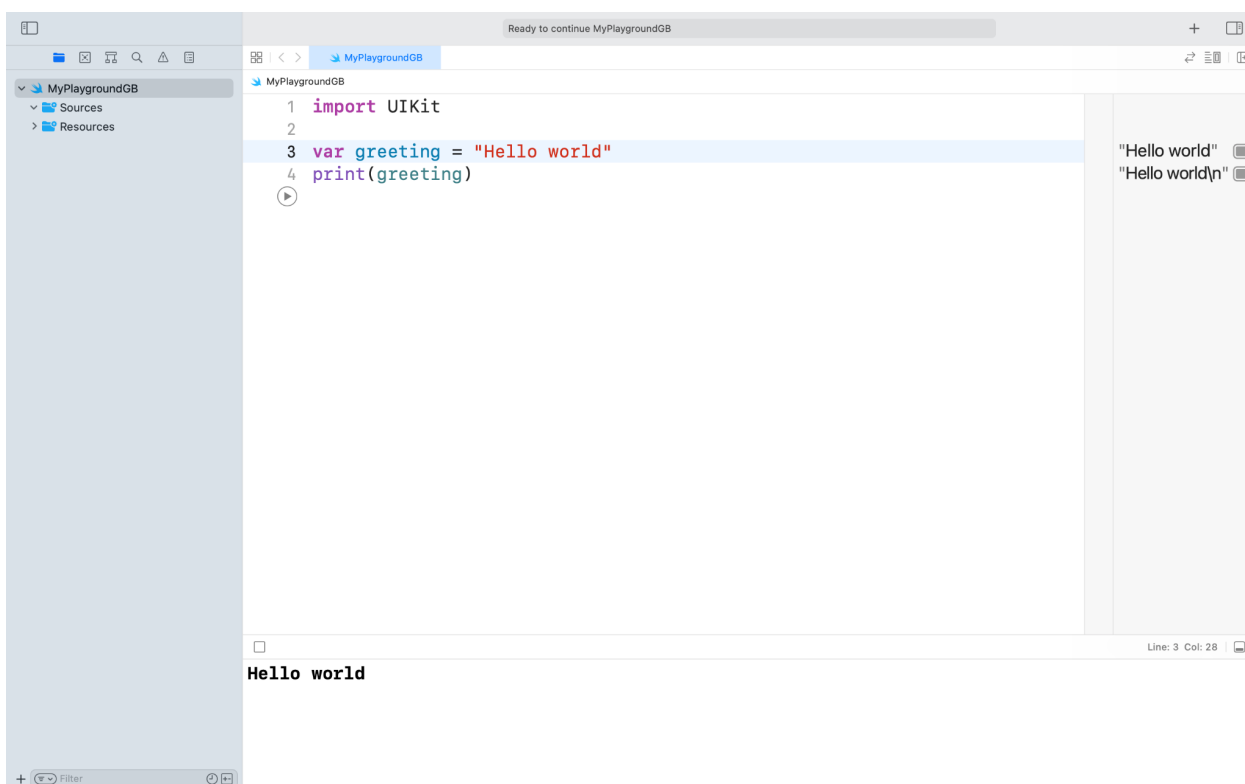
Чтобы запустить код необходимо нажать на “play” или внизу экрана или после кода



После запуска программы справа можно увидеть значения переменных, в данном случае “Hello, playground”



Чтобы вывести в консоль “Hello world!” необходимо сначала изменить значение `greeting` на необходимое, а затем строкой ниже вызвать `print` и в скобках указать, какую переменную необходимо вывести и нажать на запуск. Внизу в консоли будет выведен текст. Первая программа готова!



Константы и переменные

В своей первой программе мы использовали `greeting`, перед которым указано ключевое слово `var`. Зачем же оно нужно и как может пригодиться? Здесь мы переходим к понятиям “константы” и “переменные”.

Переменная – это область в памяти, которая проименована, её используют для доступа к данным. В Swift для объявления переменной используется ключевое слово «`var`». Данные, находящиеся в переменной называются значением переменной, а значение переменной может быть изменено.

Константа – переменная, значение которой не может быть изменено, для её объявления используется ключевое слово «`let`».

Таким образом, необходимо использовать `var`, когда мы точно знаем, что переменная будет изменяться, а `let`, когда значение не подлежит изменению.

Например, есть переменная, хранящая в себе пароль и заранее известно, что пароль можно изменять, значит переменная с паролем должна быть `var`

А длина пароля должна быть обязательно 8 и изменять длину нельзя, в этом случае длина пароля это константа, `let`




```
var password = "some"  
let passwordLength = 8
```

```
1 let a: Int = 5
```

```
2 a = 6
```

Cannot assign to value: 'a' is a 'let' constant

 Компилятор (Программы, которые преобразуют исходный код, написанный на языке программирования высокого уровня, в программу на машинном языке, то есть языке, понятном компьютеру) следит за объявлением констант и переменных. Если попробовать переопределить константу, то выскочит ошибка. Если используется переменная, которая далее в коде не изменяется, то выскочит предупреждение с рекомендацией поменять «var» на «let».

Основные типы данных

Мы узнали что такое константы и переменные и чем они отличаются, но нам же нужно что-то хранить в них? Для этого нам необходимо узнать про типы данных, которые предоставляет Swift

Integer -- это число, которое не содержит дробной части, например, 5. Число может быть как положительным, так и отрицательным.

```
let a: Int = 5  
let b: Int = -7
```

Double – это 64-битное число с плавающей точкой, например, 6,52.

```
let a: Double = 6.52
```

Bool – логический тип данных. Принимает значения true или false.

```
let a: Bool = true  
let b: Bool = false
```

String – это строка, например, “hello”.



```
let a: String = "Hello!"
```

Character -- это символ от строки, например, если «hello» это String, то «h» это character.

```
let a: Character = "H"
```

Статическая типизация

Мы познакомились с типами данных, а нужно ли нам всегда указывать его для переменной? Забегая вперед - да, нужно, а чтобы понять почему, давайте познакомимся с видами типизации, которые существуют.

Динамическая типизация – не требуется явное объявление типов, тип присваивается переменной во время выполнения.

Статическая типизация – все типы переменных должны быть указаны, так как эта информация используется во время компиляции.

Swift – язык со статической типизацией, а это значит, что если какая-то часть кода ждет Double, то нельзя передать туда String.

```
1 var a: Double = 6.7
```

```
2 a = "a"
```

Cannot assign value of type 'String' to type 'Double'

В языке Swift для указания типа используется “:”.

```
var a: Double = 6.7  
var b: Int = 6  
var c: String = "Hello!"
```

Вывод типов

Swift - язык со статической типизации, поэтому тип переменной должен быть указан заранее, до компиляции, но неужели при создании переменной или константы каждый раз нужно писать двоеточие и тип, это ведь отнимает время. Здесь на помощь приходит Вывод типов.

Вывод типов (или type inference) – это возможность компилятора самому логическим образом вывести тип. Таким образом, несмотря на то, что Swift – язык со строгой типизацией, нет необходимости каждый раз указывать тип переменной. Если нужный тип не указан, то компилятор сам выведет необходимый тип, исходя из значения, которое ему было передано.



```
var a: Double = 6.7 // Double
var b: Int = 6 // Int
var c: String = "Hello!" // String

var d = 7 // Int
var e = 7.2 // Double
var f = "Hi" // String
```



Нужно быть осторожным, не всегда компилятор укажет именно тот тип, который задумывался заранее.

Допустим, изначально есть значение 5, но известно, что далее оно будет 6.5. Если изначально не указать нужный тип, то будет ошибка, так как компилятор считает, что у переменной тип Int.

```
1 var a = 5
```

```
2 a = 6.5
```



Cannot assign value of type 'Double' to type 'Int'

Для того, чтобы ошибки не было, необходимо явно указать тип.

```
1 var a: Double = 5
```

```
2 a = 6.5
```

Арифметические операторы

Мы узнали про типы данных в Swift и как их указывать, пришло время узнать, как же можно написать пять плюс семь и присвоить результат в переменную? Для этого познакомимся, что такое арифметические операторы и какие они бывают.

Арифметические операторы - это операторы, которые принимают в качестве операторов некоторые значения и производят с ними математические действия. К таким операторам, например, относятся сложение, вычитание, деление, умножение.

Присваивание используется для инициализации или изменения значения переменной, строится следующим образом:

<переменная, которой необходимо присвоить значение> <оператор присваивания> <значение, которое необходимо присвоить>.

В качестве значения, которое необходимо присвоить может быть не только число, строка, логическое значение, но и целые выражения, например, 5+6.



```
var a = 5 // а теперь равно 5
```

Оператор сложения складывает два значения, расположенных по разные стороны от «+», строится следующим образом:

<переменная для сложения> <оператор сложения> <переменная для сложения>.

```
var a = 5 + 7 // а теперь равно 12
```

Оператор вычитания вычитает два значения, расположенных по разные стороны от «-», строится следующим образом:

<переменная для вычитания> <оператор вычитания> <переменная для вычитания>.

```
var a = 7 - 5 // а теперь равно 2
```

Оператор умножения перемножает два значения, расположенных по разные стороны от «*», строится следующим образом:

<переменная для умножения> <оператор умножения> <переменная для умножения>.

```
var a = 7 * 5 // а теперь равно 35
```

Оператор деления делит значение, расположенное справа от «/» на значение, указанное слева от «/», строится следующим образом:

<Делимое> <оператор деления> <делитель>.

```
var a = 10 / 5 // а теперь равно 2
```

Для вычисления остатка от деления используется «%». Слева указывается значение, от которого необходимо посчитать остаток от деления, справа указывается число, на которое необходимо делить, строится следующим образом:

<переменная от которой необходимо рассчитать остаток от деления> <оператор деления> <делитель>.

```
var a = 10 % 5 // Результат 0  
a = 15 % 10 // Результат 5
```




```
a = 14 % 3 // Результат 2
```

$$\begin{array}{r} 10 \overline{) 5} \\ \underline{10} \\ 0 \end{array} \quad \begin{array}{r} 14 \overline{) 3} \\ \underline{12} \\ 2 \end{array} \quad \begin{array}{r} 15 \overline{) 10} \\ \underline{10} \\ 5 \end{array}$$

Составные выражение

Пять плюс семь это простое выражение с одним действием. А можно ли использовать одновременно несколько арифметических операторов? Да, можно!

Несколько операторов можно комбинировать в одном выражении. В выражении операторы выполняются в порядке приоритета, согласно правилам математики. Порядок выполнения также можно менять при помощи скобок - «()»

```
var a = 7 + 5 - 9 // Результат 3
var b = 7 - 9 * 2 // Результат -11
var c = (7 - 9) * 2 // Результат -4
var d = (2 * 4 + 7 / 2 - 8) % 2 // Результат 1
```

В Swift есть составные операторы присваивания, они совмещают оператор присваивания с другим арифметическим оператором, например, если нужно переменной a присвоить значение $a * 5$.

```
var a = 5
a += 5 // Результат 10, a именно 5 + 5
a *= 2 // Результат 20, a именно 10 * 2
a -= 4 // Результат 16, a именно 20 - 4
a /= 2 // Результат 8, a именно 16 / 2
```

Операторы сравнения



Мы научились складывать и вычитать выражения, а что, если нужно их сравнить? Здесь помогут операторы сравнения.

Операторы сравнения используются для того, чтобы сравнить два значения. Результатом данной операции будет значение типа Bool, то есть true или false.

< - меньше

```
var a = 4 < 5 // Результат true
var b = 7 < 5 // Результат false
```

<= - меньше или равно

```
var a = 4 <= 5 // Результат true
var b = 7 <= 5 // Результат false
var c = 5 <= 5 // Результат true
```

> - больше

```
var a = 4 > 5 // Результат false
var b = 7 > 5 // Результат true
```

>= - больше или равно

```
var a = 4 >= 5 // Результат false
var b = 7 >= 5 // Результат true
var c = 5 >= 5 // Результат true
```

== - равно

```
var a = 4 == 5 // Результат false
var b = 7 == 5 // Результат false
var c = 5 == 5 // Результат true
```

!= - не равно

```
var a = 4 != 5 // Результат true
var b = 7 != 5 // Результат true
var c = 5 != 5 // Результат false
```



Булева алгебра

Мы уже знаем про арифметические операторы и как объединять их в составные выражения. Точно так же можно объединять и операторы сравнения. Чтобы разобраться как это делать, рассмотрим что такое Булева алгебра

Булева алгебра (или алгебра логики) – раздел математики, который изучает высказывания с точки зрения их логических значений и операций над ними. Используется, когда, например, нужно проверить несколько условий и привести к одному значению, типа Bool, то есть true или false, например, если длина пароля больше 5 и пароль не является “qwerty”, в данном случае мы используем сразу несколько логических операторов, это && - и, а также ! - не.

```
if passwordLength > 5 && password != "qwerty" {  
    // некие действия  
}
```

Логическое И (или конъюнкция) возвращает true только если оба оператора true. Для обозначения используется «&&».

```
true && true // Результат true  
true && false // Результат false  
false && true // Результат false  
false && false // Результат false
```

Логическое ИЛИ (или дизъюнкция) возвращает false только если оба оператора false. Для обозначения используется «||».

```
true || true // Результат true  
true || false // Результат true  
false || true // Результат true  
false || false // Результат false
```

Логическое НЕ (или отрицание) инвертирует значение, то есть true станет false, а false true.

```
let a = true  
let b = a // Результат true  
let c = !a // Результат false  
  
let d = false  
let e = d // Результат false
```



```
let f = !d // Результат true
```

В одном выражении может быть несколько логических операторов.

```
let a = true
let b = false
let c = true

let d = !a || b && c // Результат false
let e = (a || b) && c // Результат true
let f = a && b && c // Результат false
let g = a || c // Результат true
let h = (a && b && c) || (a || c) // Результат true
let i = (a && b && c) && (a || c) // Результат false
let j = a && b && c || a || c // Результат true
```

Операторы диапазона

Мы знаем, как задать для переменной значение 5. А что делать, если значений должно быть несколько, например 5, 6, 7 или 12, 13, 14, 15? Тут нам пригодятся диапазоны.

Операторы диапазона - это операторы для обозначения диапазона, то есть, например, чтобы сказать переменной, что она не просто 5, а все значение от 5 до 10. Диапазоны часто применяются в цикле `for-in` для итерации по определенным значениям, например, от 0, до количества символов в строке

```
var range = 0..password.count
```

Замкнутый диапазон - это диапазон значений от `a` до `b`, при условии, что `a` не превышает `b`, и `a` и `b` включены в диапазон.

<значение, от которого начинается диапазон, включая значение> ... <значение, на котором заканчивается диапазон, включая это значение>

```
var a = 4...7 // Диапазон от 4 до 7, включая 4 и 7
```

Полузакнутый диапазон - это диапазон значений от `a` до `b`, при условии, что `a` меньше `b` и включено только значение `a`.



<значение, от которого начинается диапазон, включая значение> ..< <значение, на котором заканчивается диапазон, исключая это значение>

```
var a = 4..7 // Диапазон от 4 до 7, включая 4 и исключая 7
```



Если значение a будет больше b, то проект скомпилируется, но в процессе выполнения произойдет краш. Значение a должно быть обязательно меньше значения b.

```
9 var a = 10
10 var b = -5
11 var c = a..b  error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code...
```

```
9 var a = 10
10 var b = -5
11 var c = a..b  error: Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code...
```

Односторонний диапазон - это диапазон, который имеет ограничение только с одной стороны.

... <значение, на котором заканчивается диапазон, включая это значение>

<значение, от которого начинается диапазон, включая значение> ...

..**<** <значение, на котором заканчивается диапазон, исключая это значение>

```
var a = ...7 // Диапазон до 7, включая 7 и включая отрицательные значения
var b = 7... // Диапазон от 7, включая 7 и далее
var c = ..<7 // Диапазон до 7, исключая 7 и включая отрицательные значения
```

Условный оператор if



Мы разобрались с константами и переменными, типами данных, арифметическими операторами и операторами сравнения, теперь рассмотрим задачу посложнее. Допустим есть переменная `b`, которая равна некоему целому числу и, если это значение больше 10, то нужно увеличить его в два раза. В этом случае на помощь приходит оператор `if`.

```
if <условие, которое подлежит проверке> {  
    <действия, которое необходимо выполнить, если условие  
    верно>  
}
```

```
var b = 15  
if b > 10 {  
    b *= 2 // Действие выполнится, так как b больше 10 и b  
    станет равно 30  
}  
  
if b > 40 {  
    b *= 4 // Действие не выполнится, так как b меньше 40  
}
```

Условие может содержать не только один оператор сравнения, но и быть целым составным выражением.

```
var b = 15  
if b > 10 && b < 14 {  
    b *= 2 // Действие не выполнится, так как условие - false,  
    b больше 10 - true, b < 14 - false, true && false = false  
}  
  
if b > 4 || b == 10 {  
    b *= 4 // Действие выполнится, так как условие - true, b  
    больше 4 - true, b равно 10 - false, true || false = true, b  
    станет равно 60  
}
```



Также составное выражение может иметь внутри арифметические операторы при проверке условия.

```
var b = 15
var a = 7
if b > (a - 4) || b <= (a * 4) {
    b *= 2 // Действие выполнится, так как условие - true, b
    больше 11 - true, b <= 28 - true, true || true = true, b станет
    30
}

if b > (a + b / 5) && b < (a + 5) {
    b *= 4 // Действие не выполнится, так как условие - false,
    b больше 10 - true, b меньше 12 - false, true && false = false
}
```

Выше представлены ситуации, когда после условия выполняется только манипуляция с b, тем не менее, после условного оператора можно совершить сразу несколько действий.

```
var b = 15
var a = 7
if b > (a - 4) || b <= (a * 4) {
    b *= 2 // Действие выполнится, так как условие - true, b
    больше 11 - true, b <= 28 - true, true || true = true, b станет
    30
    a = (b + 4) * 7 // В результате a станет 238
}

if b > (a + b / 5) && b < (a + 5) {
    b *= 4 // Действие не выполнится, так как условие - false,
    b больше 10 - true, b меньше 12 - false, true && false = false
    a = b / 5 // a не изменится, так как условие не выполнено
}
```



Условие, которое подлежит проверке обязательно должны быть типа Bool.



```
10 if b + 5 {  
11     b += 5  
12 }
```

Type 'Int' cannot be used as a boolean; test for '!= 0' instead

Далее рассмотрим оператор if/else, он имеет следующую конструкцию:

```
if <условие, которое подлежит проверке> {  
    <действия, которое необходимо выполнить, если условие истинно>  
} else {  
    <действия, которое необходимо выполнить, если условие ложно>  
}
```

В примере ниже рассматривается, что, если $b < 10$, то b будет равно $b * 2$, в противном случае b будет равно $b - 5$

```
var b = 7  
if b < 10 {  
    b *= 2 // Выполнится, так как 7 < 10, b станет равно 14  
} else {  
    b -= 5 // Не выполнится, так как первое условие уже истинно  
}
```

Конструкция if/else подходит, когда при истинности условия сделать одно действие, а если оно ложное, то другое, но бывают ситуации, когда эта конструкция не подходит, например, если b равно 10 выполнить одно действие, а если b меньше 8, то другое. В таком случае пригодится if/if else, который имеет следующую конструкцию:

```
if <условие, которое подлежит проверке> {  
    <действия, которое необходимо выполнить, если условие истинно>  
} else if <условие, которое подлежит проверке, если предыдущее условие ложно> {  
    <действия, которое необходимо выполнить, если условие истинно>  
}
```




В примере ниже рассматривается, что, если `b` равно 10, то `b` будет равно `b * 2`, в противном случае, если `b` меньше 8, то `b` будет равно `b - 5`, во всех остальных случаях `b` не изменится

```
var b = 7
if b == 10 {
    b *= 2 // Не выполнится, так как 7 не равно 10
} else if b < 8 {
    b -= 5 // Выполнится, так как и предыдущее условие ложно, и 7 < 8, b станет равно 2
}
```

Кроме того, подряд может идти сразу несколько `if else`.

```
var b = 15
if b == 10 {
    b *= 2 // Не выполнится, так как 15 не равно 10
} else if b < 8 {
    b -= 5 // Не выполнится, так как 15 не меньше 8
} else if b > 12 {
    b += 4 // Выполнится, так как и предыдущие два условия ложны, и 15 больше 12
}
```

Также `if else` можно комбинировать с `else`. Действие после `else` будет выполнено, если все остальные условия ложны.

```
if <условие, которое подлежит проверке> {
    <действия, которое необходимо выполнить, если условие верно>
} else if <условие, которое подлежит проверке, если предыдущее условие ложно> {
    <действия, которое необходимо выполнить, если условие верно>
} else {
    <действия, которое необходимо выполнить, если все условия ложны>
}
```

```
var b = 9
```



```
if b == 10 {  
    b *= 2 // Не выполнится, так как 9 не равно 10  
} else if b < 8 {  
    b -= 5 // Не выполнится, так как 9 не меньше 8  
} else if b > 12 {  
    b += 4 // Не выполнится, так как 9 меньше 12  
} else {  
    b = 100 // Выполнится, так как все остальные условия ложны,  
    b станет равно 100  
}
```

Тернарный условный оператор

Помимо if/else задать условие “если, то, иначе” может и тернарный оператор. Давайте рассмотрим что это и чем отличается от if/else.

Тернарный условный оператор - это операция, которая возвращает свой второй или третий операнд, в зависимости от логического значения, которое задано в начале. Часто используется вместо if/else, когда нужно произвести всего одно действие, например, если значение одной переменной больше 5, то новая переменная равна 10, а в противном случае - 10.

При использовании if/ else код бы выглядел следующим образом:

```
let c: Int  
if passwordLength > 5 {  
    c = 10  
} else {  
    c = -10  
}
```

А при использовании тернарного оператора, весь этот код можно свести к одной строке

```
let c = passwordLength > 5 ? 10 : -10
```

Тернарный условный оператор очень похож на if/else, но имеет другой синтаксис:

<условие> ? <действие, которое выполняется, если условие истинно> :
<действие, которое выполняется, если условие ложно>



```
let a = 15
let b = a > 6 ? 4 : 12 // b станет равно 4, так как 15 больше 6
и выполнится первое действие, то есть присвоение b значения 4
let c = a == 10 ? 5 : 20 // c станет равно 20, так как 15 не
равно 10 и выполнится второе действие, то есть присвоение c
значения 20
```

Как и в `if`, условие может состоять не из одного оператора сравнения, а быть составным выражением.

```
let a = 15
let b = (a > 20) || (a < 16) ? 4 : 12 // b станет равно 4, так
как 15 меньше 20, а значит первое выражение false, 15 меньше
16, а значит второе выражение true, false || true = true, таким
образом выполнится первое действие, то есть присвоение b
значения 4
let c = (a > 20) && (a < 16) ? 5 : 20 // c станет равно 20, так
как 15 меньше 20, а значит первое выражение false, 15 меньше
16, а значит второе выражение true, false && true = false,
таким образом выполнится второе действие, то есть присвоение c
значения 20
```



Не нужно писать тернарные операторы следующего вида, так как это “захламление” кода:

```
let a = 15
let b = a > 10 ? true : false
```

Такой тернарный оператор заменяется более коротким выражением, а результат такой же.

```
let a = 15
let b = a > 10
```



Конструкция switch

Мы узнали про if/else и тернарный оператор, а можно ли как-то еще в языке Swift реализовать конструкцию “если, то”? Да, можно, давайте познакомимся с конструкцией switch.

Конструкция switch напоминает if/else и имеет следующий синтаксис:

```
switch <сравниваемое значение> {  
    case <проверяемое значение 1> : <действие, если сравнение  
прошло успешно>  
    case <проверяемое значение 2> : <действие, если сравнение  
прошло успешно>  
    case <проверяемое значение 3> : <действие, если сравнение  
прошло успешно>  
    default: <действие, если ни одно из предыдущих сравнений не  
прошло успешно>  
}
```

```
var a = 15  
switch a { // Сравниваем значение a  
case 20: a -= 5 // Не выполнится, так как 15 не равно 20  
case 15: a -= 10 // Выполнится, так как 15 равно 15, а будет  
равно 5  
case 5: a += 5 // Не выполнится, так как одно из предыдущих  
сравнений прошло успешно  
default: a += 15 // Не выполнится, так как одно из предыдущих  
сравнений прошло успешно  
}
```

Действие, которое описано в default будет выполнено только в том случае, если ни одно из предыдущих сравнений не было успешно.

```
var a = 14  
switch a { // Сравниваем значение a  
case 20: a -= 5 // Не выполнится, так как 14 не равно 20  
case 15: a -= 10 // Не выполнится, так как 14 не равно 15  
case 5: a += 5 // Не выполнится, так как 14 не равно 5  
default: a += 15 // Выполнится, так как ни одно из предыдущих
```



```
сравнений не было успешно, а станет равно 29  
}
```



Если описаны все значения для сравнения, то указывать “default” не нужно, так как действие после него никогда не будет выполнено. Если же оно будет указано, то высветится предупреждение.

```
1 var a = 14  
2 switch a > 14 {  
3 case true: a += 10  
4 case false: a -= 10  
5 default: a *= 2 // Никогда не будет вызвано, так как сравниваемое  
                // значение имеет тип Bool, а он может быть только true или  
                // false.  
6 }
```

⚠ Default will never be executed

Помимо default, действие для значений, не перечисленных ранее, можно задать при помощи “_”.

```
var a = 14  
switch a {  
case 10: a += 10 // Не вызовется, так как 14 не равно 10  
case 12: a -= 10 // Не вызовется, так как 14 не равно 12  
case _: a *= 2 // Вызовется, так как это case для всех  
              // остальных значений, а станет равно 28  
              // default не указан, так как все варианты уже перечислены  
              // ранее  
}
```

В качестве проверяемых значений может быть несколько значений, в этом случае они перечисляются через запятую.

```
switch <сравниваемое значение> {  
    case <проверяемое значение 1>, <проверяемое значение 2> :  
        <действие, если сравнение хоть с одним проверяемым значением  
        прошло успешно>
```



```
    case <проверяемое значение 3>, <проверяемое значение 4>,
    <проверяемое значение 5> : <действие, если сравнение хоть с
    одним проверяемым значением прошло успешно>
    case <проверяемое значение 6> : <действие, если сравнение
    прошло успешно>
    default: <действие, если ни одно из предыдущих сравнений
    не прошло успешно>
}
```

```
var a = 14
switch a {
case 10: a += 10 // Не вызовется, так как 14 не равно 10
case 12, 14: a -= 10 // Вызовется, так как среди проверяемых
    значений указано 14 и 14 равно 14, а станет равно 4
default: a *= 2 // Не вызовется, так как одно из предыдущих
    сравнений прошло успешно
}
```

Также в качестве проверяемых значений могут быть диапазоны.

```
switch <сравниваемое значение> {
    case <диапазон проверяемых значений 1> : <действие, если
    сравнение прошло успешно>
    case <проверяемое значение 2> : <действие, если сравнение
    прошло успешно>
    case <диапазон проверяемых значений 2> : <действие, если
    сравнение прошло успешно>
    default: <действие, если ни одно из предыдущих сравнений
    не прошло успешно>
}
```

```
var a = 14
switch a {
case 10..<14: a += 10 // Не вызовется, так как диапазон от 10
    до 14, исключая 14, а значит 14 не входит в диапазон
case 14...20: a -= 10 // Вызовется, так как диапазон от 14 до
    20, включая 14, а значит 14 входит в диапазон, а станет равно 4
default: a *= 2 // Не вызовется, так как одно из предыдущих
```



```
сравнений прошло успешно  
}
```

Как в case может быть несколько проверяемых значений, так и выполняемых действий может быть несколько.

```
var a = 14  
var b = 3  
var c = 5  
switch a {  
case 10:  
    a += 10 // Не вызовется, так как 14 не равно 10  
    b = c * a // Не вызовется, так как 14 не равно 10  
case 12, 14:  
    a -= 10 // Вызовется, так как среди проверяемых значений  
указано 14 и 14 равно 14, a станет равно 4  
    b += a + c // Вызовется, так как среди проверяемых значений  
указано 14 и 14 равно 14, b станет равно 12 (значение получено  
следующим образом: 3 + 4 + 5)  
    c = 20 - a * 2 // Вызовется, так как среди проверяемых  
значений указано 14 и 14 равно 14, c станет равно 12  
default: a *= 2 // Не вызовется, так как одно из предыдущих  
сравнений прошло успешно  
}
```

В языке Swift при выполнении switch как только находится нужный case, то есть в котором проверяемое значение совпадает с сравниваемым значением - выполнение конструкции сразу прекращается. Однако, во многих языках программирования при использовании switch происходит проваливание из одного case в другой, то есть, даже если проверяемое значение совпадает с сравниваемым значением, выполнение конструкции не прекращается, а происходит проверка следующего case. Чтобы сделать такое поведение в Swift необходимо использовать fallthrough.

```
var b = 1  
switch b {  
case 1: b += 1 // Выполнится, так как b равно 1, b станет равно  
2
```



```
    fallthrough
case 2: b += 2 // Выполнится, так как в предыдущем case
              присутствует оператор передачи управления "fallthrough", b
              станет равно 4
    fallthrough
case 3: b += 3 // Выполнится, даже не смотря на то, что 4 не
              равно 3, так вышло потому, что в предыдущем case присутствует
              оператор передачи управления "fallthrough", b станет равно 7
case 4: b += 4 // Не выполнится, так как одно из предыдущих
              сравнений прошло успешно и не был вызван "fallthrough"
default: b *= 2 // Не выполнится, так как одно из предыдущих
              сравнений прошло успешно и не был вызван "fallthrough"
}
```

Guard

if/else, тернарный оператор, switch: как много способов задать конструкцию “если, то”! А есть еще? Да, есть и это guard.

Guard - это оператор, схожий с if/else. То есть если для if/else используется синтаксис:

```
if <условие, которое подлежит проверке> {
    <действия, которое необходимо выполнить, если условие
    истинно>
} else {
    <действия, которое необходимо выполнить, если условие
    ложно>
}
```

То для guard используется следующая конструкция:

```
guard <условие> else {
    <действие, которое выполняется, если условие ложно>
    return
}
<действие, которое выполняется, если условие истинно>
```

Ранее для if/else приводился следующий пример:



```
var b = 7
if b < 10 {
    b *= 2 // Выполнится, так как 7 < 10, b станет равно 14
} else {
    b -= 5 // Не выполнится, так как первое условие уже истинно
}
```

Ниже представлен этот же пример, но уже с использованием guard

```
guard b < 10 else {
    b -= 5 // Не выполнится, так как условие истинно, то
    есть 7 действительно меньше 10, а данное действие может быть
    выполнено только если b больше или равно 10
    return // Обязательно указывать при использовании guard
}
b *= 14 // Выполнится, так как условие истинно, то есть 7
действительно меньше 10, b станет равно 2
```

For-in

Мы рассмотрели способы задавать конструкцию “если, то”, пришло время перейти на следующий уровень и узнать, что такое циклы, какие они бывают и какие задачи помогают решить.

For-in - это циклический оператор, позволяющий выполнить одно и то же действие несколько раз. Имеет следующую конструкцию:

```
for <имя переменной из списка значений> in <список значений> {
    <действие>
}
```

```
var k = 0
for num in 0...3 {
    k += num
}
```

Цикл будет выполняться от 0 до 3, включая 3. num - имя для значения из списка значений, то есть в первой итерации оно будет равно 0, во второй 1 и так далее.



1 итерация: num = 0, k станет равно 0

2 итерация: num = 1, k станет равно 1

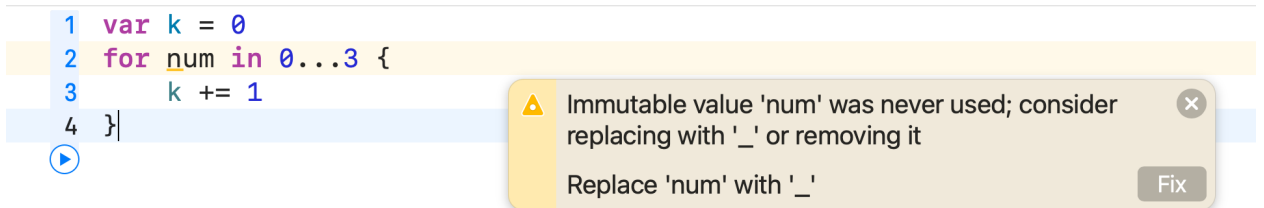
3 итерация: num = 2, k станет равно 1 + 2 = 3

4 итерация: num = 3, k станет равно 3 + 3 = 6

Также, бывают случаи, когда значение из списка значений не нужно, а необходимо только выполнить какое-то действие несколько раз, тогда вместо num необходимо указать “_”.

```
var k = 0
for _ in 0...3 {
    k += 1
}
```

💡 Если указать имя для значения из списка значений, но не использовать эту переменную в теле цикла, то выскочит предупреждение и предложение изменить имя на “_”.



Where

Where - ключевое слово для указания особых условий для списка значений, например, необходимо проходить только по четным индексам.

```
var k = 0
for num in 0...3 where num % 2 == 0 {
```



```
k += num  
}
```

Условие может быть не одно, например значение должно быть не только четным, но и больше 4.

```
var k = 0  
for num in 0...3 where num % 2 == 0 && num > 4 {  
    k += num  
}
```

While

While - это циклический оператор, который позволяет выполнять действие, пока условие не будет равно false. Имеет следующую конструкцию:

```
while <условие> {  
    <действие>  
}
```

В коде ниже происходит следующее: пока k не равно 4 увеличиваем увеличиваем переменную на 1.

```
var k = 0  
while k != 4 {  
    k += 1  
}
```

1 итерация: k = 0, значит необходимо зайти в цикл, k становится равным 1

2 итерация: k = 1, значит необходимо зайти в цикл, k становится равным 2

3 итерация: k = 2, значит необходимо зайти в цикл, k становится равным 3

4 итерация: k = 3, значит необходимо зайти в цикл, k становится равным 4

5 итерация: k = 4, 4 равно 4, значит действие выполнено не будет, цикл заканчивается



В цикле может выполняться не одно действие, а сразу несколько

```
var k = 0
var a = 5
var b = 7
while k != 4 {
    a = b + 3
    b = k - 2
    k += 1
}
```

Repeat-while

Repeat-while - это, как и while, циклический оператор, который позволяет выполнять действие, пока какое-то условие не будет равно false, но while оценивает условие в начале, а repeat-while в конце. Имеет следующую конструкцию:

```
repeat {
    <действие>
} while <условие>
```

Отличие repeat-while от while:

```
var k = 0
while k < 0 { // 0 не меньше 0, а значит цикл не выполнится
    k += 1 // Не выполнится
}
```

```
var k = 0
repeat {
    k += 1 // Сначала k увеличивается на 1
} while k < 0 // Проверяется условие
```

В первом случае k останется равно 0, а во втором случае k станет единицей.



Таким образом, из-за проверки в конце, цикл `repeat-while` обязательно выполнится хотя бы один раз, в отличие от цикла `while`.

Действий в цикле, так же, как и в `while`, может быть не одно.

```
var k = 0
var a = 5
var b = 7
repeat {
    a = b + 3
    b = k - 2
    k += 1
} while k < 4
```

Операторы передачи управления

Мы узнали про циклы и их виды, а что, если нам нужно завершить цикл раньше времени? Давайте познакомимся

Операторы передачи управления - это операторы, которые передают управления от одного куска кода к другому, изменяя при этом последовательность

1. `Continue` - говорит циклу, что необходимо прекратить выполнение итерации и перейти к следующей.

```
var k = 0
for num in 0...3 {
    if num < 2 {
        continue
    } else {
        k += 1
    }
}
```

После выполнения этого кода `k` будет равно 2:



1 итерация: num = 0, $0 < 2$, а значит происходит переход к следующей итерации

2 итерация: num = 1, $1 < 2$, а значит происходит переход к следующей итерации

3 итерация: num = 2, 2 не меньше 2, а значит k увеличивается на 1 и становится равно 1

4 итерация: num = 3, 3 не меньше 2, а значит k увеличивается на 1 и становится равно 2

2. Throw - используется при генерации ошибок
3. Fallthrough - используется в switch для “проваливания” к следующему условию
4. Break - останавливает выполнение цикла

```
var k = 0
for num in 0...3 {
    if num < 2 {
        break
    } else {
        k += 1
    }
}
```

После выполнения кода k останется равно 0:

1 итерация: $0 < 2$, а значит будет вызван оператор break, который останавливает выполнение цикла и перехода к следующей итерации не будет

5. Return - используется для выхода из функции



Заключение

Итак, в рамках сегодняшнего занятия мы узнали, как создать свой первый playground, отличие констант от переменных, познакомились с арифметическими операторами и операторами сравнения, а также научились собирать их в составные выражения. Мы рассмотрели множество способов задать конструкцию “если, то”, а также узнали про разные виды циклов. Теперь вы можете смело открыть Xcode, создать песочницу и запустить свою первую программу!

Что можно почитать еще?

1. [SwiftBook. Основы](#)
2. [SwiftBook. Базовые операторы](#)
3. [SwiftBook. Инструкции](#)
4. [SwiftBook. Базовые операторы](#)
5. [SwiftBook. Циклы](#)
6. <https://testing.swiftbook.ru/documentation/control-flow/control-transfer-statements/>

Используемая литература

1. <https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html>
2. <https://docs.swift.org/swift-book/ReferenceManual/Statements.html>
3. <https://developer.apple.com/documentation/swift/range>

Домашнее задание

1. Установить Xcode
2. Создать playground