**Лабораторная работа №2-3**

**SWIFT. ПЕРЕМЕННЫЕ, ТИПЫ, МАССИВЫ, ОПЕРАТОРЫ УПРАВЛЕНИЯ**

**Простые значения**

Используйте let для создания констант и var для объявления переменных. Значение константы не обязательно должно быть известно на момент компиляции, но оно должно присваиваться строго один раз.

var myVariable = 42

myVariable = 50

let myConstant = 42

Константа или переменная должна иметь тот тип, что и значение, которое вы хотите присвоить ей, однако вам не обязательно всегда объявлять тип — компилятор сам может определить тип, если при создании константы или переменной вы указываете ее значение. В приведенном выше примере, компилятор определит, что myVariable — это целое число (integer), т.к. начальное значение — целое число.

Если начальное значение не предоставляет достаточной информации (или его нет), укажите тип вручную:

let explicitDouble: Double = 70

Значения никогда не конвертируются в другой тип неявно. Если необходимо сконвертировать значение в другой тип, вы должны явно это показать:

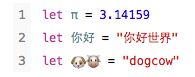
let label = "The width is "

let width = 94

let widthLabel = label + String(width)

**Имена для констант и переменных**

Вы можете использовать практически любые символы для названий констант и переменных, включая символы юникода:



Имена констант и переменных не могут содержать *математических символов, стрелок, точек и некоторых символов* Unicode (private-use or invalid Unicode code points), символов для рисования линий или прямоугольников. Они так же *не могут начинаться с цифры*.

Переменные и константы следует именовать в нижнем камелкейсстиле.

Если требуется дать переменной или константе имя, зарезервированное в Swift под какое-либо служебное ключевое слово, то следует написать его в апострофах (`), но настоятельно рекомендуется избегать этого, чтобы не нарушать читабельность кода .

Как только вы объявили константу или переменную определенного типа, вы не превратите константу в переменную или наоборот.

**Вывод значений констант и переменных на экран**

Вы можете вывести текущее значение константы или переменной, используя функцию println:

var friendlyWelcome = "Hello"

println(friendlyWelcome)

// получим на экране Hello

В XCode это сообщение будет выведено в окошке «console» (консоль). Есть другая функция — print — она делает то же самое, что и println, но не переводит текст на новую строку

При работе с playground-проектами можно вывести любое значение в окне результатов . Для этого необходимо всего лишь написать имя объявленной и проинициализированной ранее переменной или константы

var friendlyWelcome = "Hello"

friendlyWelcome

Swift использует интерполяцию строк для включения имени константы или переменной в определенное место строки — для этого, она должна быть заключена в скобки и предваряться обратным слэшем, вот так:

var friendlyWelcome = "Hello"

println("The current value of friendlyWelcome is \(friendlyWelcome)")

// печатает The current value of friendlyWelcome is Hello

**Комментарии**

Комментарии в Swift аналогичны комментариям в С. Однострочные комментарии начинаются с двух слэшей:

// это комментарий

Вы можете написать многострочный комментарий, начав его с символов /\* и закончив \*/:

/\* это тоже комментарий,

но он на нескольких строках \*/

В отличие от многострочных комментариев в С, многострочные комментарии в Swift могут быть вложенными:

/\* начало комментария

/\* а вот мы вложили комментарий и закончили это вложение \*/

и закрыли первый комментарий \*/

Вложенные комментарии позволяют вам закомментировать большие куски кода быстро и без проблем, даже если он уже содержит многострочные комментарии.

playgound-проекты в Xcode поддерживают markdown-комментарии — особый *форматированный вид комментариев*. Такой вид комментариев должен начинаться с удвоенной косой черты и двоеточия (//:), после которых и следует текст *комментария.*

*Справочные комментарии* пишутся в виде многострочного комментария, но с использованием ключевых слов

/\*\*

This func say hello to user

— parameter name:String Name of user

— returns: Absolutely nothing

— throws: Error when name is array.

— authors: Bilbo Baggins

— bug: This is very simple function

\*/

func sayHello(name: String) {

print("hello, \(name)!")

}

sayHello(name:

sayHello(name: "Abba")

При щелчке мышкой на имени этой функции при зажатой Alt отобразится справочное окно

скрин

**Точка с запятой**

В отличие от многих других языков, Swift не требует от вас писать точку с запятой после каждой строчки кода, но вы можете писать их, если захотите.

let cat = "cat";

println(cat);

//напечатает cat;

Точки с запятой обязательны, если вы хотите написать несколько утверждений в одной строке:

let cat = "cat"; println(cat);

//напечатает cat

**Типы данных**

В Swift, по аналогии с C, Objective-C, а также другими языками программирования, есть ряд предустановленных (как их называет Apple, фундаментальных) типов данных .

Все фундаментальные типы данных — это так называемые типы значения, или значимые типы (value type) . Помимо типов-значений в Swift существуют типы-ссылки, или ссылочные типы (reference type)

.

**Целые числа (тип Integer)**

Swift представляет целые числа в виде знаковых и беззнаковых целых с размерами в 8, 16, 32 и 64 бита. Эти целые числа следуют стандартному именованию в Си, то есть 8 битное беззнаковое целое число имеет тип **UInt8**, 32битное знаковое число — **Int32**. Как и все типы в Swift, эти типы целых чисел должны начинаться с большой буквы.

Вы можете получить минимальное и максимальное значение каждого числа с помощью свойств min и max:

let minValue = UInt8.min // минимальное значение равно 0 для этого типа

let maxValue = UInt8.max // максимальное значение равно 255 для этого типа

[Значения этих свойств (0 и 255 в нашем случае) имеют тип самого свойства

В большинстве случаев, вам не придется выбирать специальный размер целого числа для использования в вашем коде — в Swift есть тип Int, который имеет тот же размер, что и «родной» размер слова для текущей платформы, а именно:

// для 32битных платформ - Int имеет тот же размер, что и Int32

// для 64битных платформ - Int имеет тот же размер, что и Int64

Если вам не нужно работать с особым размером целых чисел, используйте Int для целых чисел в вашем коде — это позволяет коду быть последовательным и совместимым. Даже на 32 битных платформах, Int может хранить любое значение от —2 147 483 648 до 2 147 483 647 — что является достаточно большим для большинства задач.

Swift также предоставляет беззнаковый тип целых чисел UInt, который имеет тот же размер, что и «родной» размер слова на текущей платформе:

// для 32битных платформ - UInt имеет тот же размер, что и UInt32

// для 64битных платформ - UInt имеет тот же размер, что и UInt64

Итого в Swift есть целочисленные типы данных: Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64 и UInt64 . Они определяют диапазон хранимых значений: 8-, 16-, 32- и 64-битные числа .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.1**

1. *Определите несколько переменных с let и var с явным указанием типа и без (Int, Double, String, Float, Bool)*
2. *Выведите значения на консоль.*
3. *В строках добавьте выражения или числовые переменные объявленные ранее.*
4. *Используя тип Int создайте переменную. Выведите ее диапазон.*

**Числа с плавающей точкой**

**Double** является 64 битным числом с плавающей точкой

**Float** является 32 битным числом с плавающей точкой

**Double** имеет точность в минимум 15 десятичных знаков, тогда как Float — только 6.

**Безопасность типов и подбор(inference) типов**

Swift является языком с типовой безопасностью. Типовая безопасность позволяет вам быть точным в типах значений, с которыми работает ваш код.

Поскольку Swift безопасен, он выполняет проверки типов при компиляции кода и помечает любые несоответствующие типы как ошибки, что позволяет вам ловить их максимально рано при разработке.

Swift использует подбор (inference) типа для определения подходящего типа. Интерфейс типа позволяет компилятору делать вывод о типе по операциям в вашем коде во время компиляции.

Из-за подбора типа, Swift требует гораздо меньше объявлений, чем языки типа С или Objective-C. Константы и переменные по-прежнему явно типизированы, но большинство работы по определению типа сделано за вас.

Swift всегда выбирает Double, а не Float, если вы не указываете тип явно.

Если вы совместите в одной строке целочисленный литерал и литерал с плавающей точкой, тип будет также Double:

let anotherPi = 3 + 0.14159

// anotherPi - также Double

Литеральное значение не имеет заданного типа, зато компилятор видит присутствие плавающей точки — поэтому и назначает тип Double.

**Числовые литералы**

Целочисленные литералы могут быть записаны как:

Десятичное число без префикса.

Двоичное число с префиксом 0b

Восьмеричное число с префиксом 0o.

Шестнадцатеричное число с префиксом 0x

Запишем число 17 каждым из способов:

let decimalInteger = 17

let binaryInteger = 0b10001 // 17 в двоичном представлении

let octalInteger = 0o21 // 17 в восьмеричном представлении

let hexadecimalInteger = 0x11 // 17 в шестнадцатеричном представлении

Литералы с плавающей точкой могут быть десятичными (без префикса) или шестнадцатеричными (с префиксом 0x). Они обязаны всегда иметь число (десятичное или шестнадцатеричное) с обеих сторон от точки. Они также могут иметь экспоненту, отделяемую от числа буквой e или E для десятичных и буквой p или P для шестнадцатеричных чисел.

Для десятичных чисел с экспонентой exp, базовое число умножается на 10exp:



Для шестнадцатеричных чисел с экспонентой exp, базовое число умножается на 2exp:



Все следующие литералы с плавающей точкой имеют десятичное значение 12.1875:

let decimalDouble = 12.1875

let exponentDouble = 1.21875e1

let hexadecimalDouble = 0xC.3p0

Числовые литералы могут содержать дополнительное форматирование, чтобы их было легче читать. Как целые числа, так и числа с плавающей запятой могут быть дополнены нулями и содержать подчеркивания для повышения читаемости. Примеры:

let paddedDouble = 000.123.456

let oneMillion = 1\_000\_000

let justOVerOneMillion = 1\_000\_000.000\_000\_1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.2**

1. *Создайте числовые литералы для числа 5 в двоичном, десятичном и шестнадцатеричном представлении.*
2. *Создайте литерал для числа 0.36 в экспоненциальной форме.*
3. *Используйте дополнительное форматирование для числа 12305670.9805*

**Текстовые типы данных**

В Swift существует два типа данных, предназначенных для хранения текстовой информации:

* тип Character предназначен для хранения отдельных символов;
* тип String предназначен для хранения произвольной текстовой информации .

Character . позволяет хранить в переменной или константе текстовый литерал длиной в один символ

var char: Character = "a"

char // "a"

При попытке записать в параметр типа Character два и более символа Xcode сообщит об ошибке несоответствия типов записываемого значения и переменной. При попытке передать строковый литерал длиной более одного символа Swift рассматривает его в качестве значения типа данных String и не может записать его в переменную типа Character.

Тип данных String — это упорядоченный набор символов (типа Character), позволяющий хранить строку произвольной длины . То есть каждый отдельный элемент строки представляет собой значение типа Character .

*Пустая строка также* является строковым литералом. Вы можете передать ее в качестве значения параметру текстового типа данных

// с помощью пустого строкового литерала

var emptyString = ""

// с помощью инициализатора типа String

var anotherEmptyString = String()

//Обе строки в результате будут иметь идентичное (пустое) значение .

// объявление переменной типа String без передачи значения

var str: String

При явном указании типа String без инициализации значения в переменной или в константе не окажется пусто, то есть не будет вообще никакого значения, в том числе и пустого строкового литерала, — он также является значением.

**Многострочные строковые литералы**

В Swift есть возможность создания многострочных литералов . Для этого текст обрамляется с обеих сторон тремя двойными кавычками ("""строковый литерал""") .

let longString = """

Это многострочный

строковый литерал

"""

**Приведение к строковому типу данных**

Инициализатор String(\_:) может получать на входе не только текстовое значение, но и переменную произвольного типа данных

// переменная типа Double

var numDouble = 74.22

// строка, созданная на основе переменной типа Double

var numString = String(numDouble)

Функция String(\_:) преобразует переданное в нее значение в тип String .

**Объединение строк**

При необходимости вы можете объединять несколько строк в одну, более длинную. Для этого существует два механизма: *интерполяция и конкатенация*. При интерполяции происходит объединение строковых литералов, переменных, констант и выражений в едином строковом литерале (см. пример c print)

При конкатенации происходит объединение различных строковых значений в одно с помощью символа арифметической операции сложения

let firstText = "Мой вес "

var weight = 12.4

var text = firstText + String(weight)

**Алиасы типов**

Алиасы типов — это альтернативные имена для существующих типов. Их можно определить с помощью ключевого слова typealias:

typealias AudioSample = UInt16

Как только вы объявили алиас типа, вы можете использовать его в любом месте, где вы использовали бы настоящее имя типа:

var maxAmplitudeFound = AudioSample.min

//maxAmplitudeFound теперь 0

Для Swift обращение к псевдониму равносильно обращению к самому типу данных. Псевдоним — это ссылка на тип.

**Логический тип (boolean)**

Swift имеет базовый логический тип, называемый Bool. Swift предоставляет две соответствующие константы — true и false:

let orangesAreOrange = true

let turnipsAreDelicious = false

Типы этих переменных — Bool, поскольку они инициализируются булевыми литеральными значениями, поэтому мы не объявляли их тип явно — это позволяет коду быть более читаемым.

Типовая безопасность языка предотвращает использование не-логических значений в выражениях, требующих логическое значение. Следующий пример (нормально компилирующийся в С и ObjectiveC), в Swift вызовет ошибку:

let i = 1

if i {//тут будет ошибка}

Однако, альтернативный вариант будет корректным:

let i = 1

if i == 1 { //тут все ок }

Результат сравнения i == 1 является логическим значением, поэтому второй пример проходит проверку типов. Сравнения описываются в следующей главе «базовые операторы».

**Кортежи (tuples)**

Кортежи (tuples) группируют несколько значений в одно составное значение. Это значение внутри кортежа может иметь любой тип и значениям не обязательно всем быть одного типа.

В следующем примере, (404, «Not found») — это кортеж, который описывает код ответа HTTP.

let http404Error = (404, "Not found")

// http404Error имеет тип (Int, String) и равен (404, "Not found")

Кортеж (404, «Not found») группирует Int и String для того, чтобы вернуть код ответа HTTP, состоящего из двух частей — номера и понятного человеку описания. Мы можем описать этот тип как «кортеж типа (Int, String)», в данном примере он задан неявно. Порядок указания типов данных должен соответствовать порядку следования элементов в кортеже.

При объявлении кортежа ему можно явно задать требуемый тип . Тип кортежа указывается точно так же, как тип переменных и констант, — через двоеточие:

let http404Error : (Int, String)= (404, "Not found")

Вы можете разобрать (или декомпозировать, или разложить — decompose) компоненты кортежа в отдельные константы или переменные, чтобы использовать их:

let (statusCode, statusMessage) = http404Error

println("Код статуса \(statusCode)")

// печатает "Код статуса 404"

println("Сообщение - \(statusMessage)")

// печатает "Сообщене - Not found"

Если вам нужны лишь некоторые из значений кортежа, можно игнорировать его части, используя подчеркивание (\_) при разложении кортежа:

let (justTheStatusCode, \_) = http404Error

println("Код статуса \(justTheStatusCode)")

// печатает "Код статуса 404"

Другой возможный вариант — доступ по индексу элемента, начиная от 0:

println("Код статуса \(http404Error.0)")

// печатает "Код статуса 404"

println("Сообщение - \(http404Error.1)")

// печатает "Сообщение - Not found"

Можно также дать индивидуальные имена элементам в кортеже при его объявлении:

let http200Status = (statusCode: 200, description: "OK")

Если вы дали имена элементам в кортеже, вы можете использовать их, чтобы обращаться к этим элементам:

println("Код статуса \(http200Status.statusCode)")

// печатает "Код статуса 200"

println("Сообщение - \(hhttp200Status.description)")

// печатает "Сообщение - OK"

Если кортежи имеют один и тот же тип данных, можно присвоить значение одного кортежа другому . Помимо изменения значения кортежа целиком можно, используя индексы или имена, изменять значения отдельных элементов.

var http200Status = (200, "OK")

http200Status.0= 200

При необходимости можно сравнивать кортежи между собой . Сравнение кортежей производится последовательным сравнением элементов кортежей: вначале сравниваются первые элементы обоих кортежей; если они идентичны, то производится сравнение следующих элементов, и так далее до тех пор, пока не будут обнаружены неидентичные элементы

(1, "alpha") < (2, "beta")

// истина, так как 1 меньше 2.

Встроенные механизмы Swift позволяют сравнивать кортежи с количеством элементов менее 7. При необходимости сравнения кортежей с большим количеством элементов вам необходимо реализовать собственные механизмы.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.3**

1. *Создайте псевдоним типа String с именем Text .*
2. *Объявите три константы типа Text . Значения двух констант должны состоять только из цифр, третьей — из цифр и букв .*
3. *Из всех трех констант найдите те, которые состоят только из цифр, и выведите их на консоль .*
4. *Создайте кортеж (2019, FIT, 4).*
5. *Задайте имена элементам кортежа.*
6. *Выведите элементы двумя способами - по индексу, по имени.*
7. *Измените значения элемента кортежа*
8. *Запишите каждый элемент кортежа в три константы*
9. *Создайте второй кортеж, аналогичный первому по параметрам. Обменяйте значения в кортежах между собой.*

**Опциональные значения**

Вы можете использовать опциональные значения в ситуациях, где значение может отсутствовать. Опциональное значение говорит одно из двух:

«уменяестьзначениеионоравнох»

или

«уменянетникакогозначения»

Тип String имеет метод toInt, который пробует перевести содержимое строки в целочисленное значение типа Int. Однако, не каждая строка может быть сконвертирована в целое число. Из строки «123» получается число 123, однако из строки «hello, world» нельзя получить целочисленного значения.

Пример ниже использует метод toInt, чтобы попробовать перевести строку в Int:

let possibleNumber = "123"

let convertedNumber = possibleNumber.toInt()

// convertedNumber теперь имеет тип "Int?" (да-да, именно Int со знаком вопроса), или так называемый "опциональный Int"

Поскольку метод toInt может не сработать, он возвращает опциональный Int, а не обычный Int. Опциональный Int записывается как Int? — знак вопроса показывает, что значение является опциональным, т.е. переменная или константа типа Int? может или содержать какое-либо значение Int?, или не содержать никакого значения вообще. (Она не может содержать ничего другого, т.е. Bool, String и т.д. \_не\_ могут храниться в ней. Либо Int, либо ничего).

**nil**

Вы можете присвоить опциональной переменной состояние «без значения», присвоив ей специальное значение nil:

var serverResponseCode: Int? = 404

//serverResponseCode содержит Int со значением 404

serverResponseCode = nil

//serverResponseCode теперь не содержит значения

Если вы определите опциональную константу или переменную, не предоставив значения, она автоматически получит nil (отсутствие значения):

var surveyAnswer: String?

// surveyAnswer автоматически является nil

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.4**

1. *Создайте две строки с числовым значением и без.*
2. *Выполните ее преобразование к опциональному Int.*
3. *Объявите переменную типа Int?*
4. *Присвойте ей состояние – без значения.*

**Условные выражения и вынужденная распаковка**

**(forced unwrapping)**

Вы можете использовать if, чтобы выяснить, содержит ли опциональное значение какое-либо значение.

if convertedNumber != nil {

    println("convertedNumber имеет целочисленное значение.")

}

// prints "convertedNumber  имеет целочисленное значение."

Как только вы убедились, что опциональное значение содержит значение, вы можете обратиться к нему, используя восклицательный знак в конце имени. Он обозначает «я знаю, что это опциональное значение определенно содержит значение — пожалуйста, используйте его». Это и называется «вынужденная распаковка» (мы как бы принуждаем компилятор «распаковать» значение).

if convertedNumber != nil {

    println("convertedNumber has an integer value of \(convertedNumber!).")

}

// prints "convertedNumber has an integer value of 123."

**Опциональная связка**

Вы можете использовать опциональную связку, чтобы узнать, содержит ли опциональная переменная значение и если содержит, то сохранить его во временную константу или переменную. Опциональная связка может быть использована с if и while, чтобы проверить, что значение присутствует в опциональной переменной и извлечь его в константу или другую переменную одним действием.

Опциональные связки реализуются следующим образом:

if let constantName = someOptional {

    statements

}

if let constantName = someOptional { // если \_название константы\_ = \_некоторая опциональная переменная\_

//КОД

}

// то есть если в опциональной переменной someOptional есть значение, оно пишется в constantName и код внутри if выполняется

Наш предыдущий пример с конвертацией числа можно переписать следующим образом:

if let actualNumber = possibleNumber.toInt() {

   println("\(possibleNumber) имеет целочисленное значение \(actualNumber)")

} else {

   println("\(possibleNumber) не может быть сконвертирован в целое число")

}

// печатает на экран "123 имеет целочисленное значение 123"

Этот код можно прочитать так:

«Если опциональный Int, возвращаемый методом possibleNumber.toInt, содержит значение, то создайте новую константу actualNumber со значением, которое там содержится».

Если конвертация прошла успешно, actualNumber становится доступной в пределах первой ветки выражения if. Она уже инициализирована значением, поэтому нет нужды использовать восклицательный знак. В нашем примере мы просто выводим ее на экран.

Вы можете использовать как константы, так и переменные, для опциональной связки. Если вы хотите управлять значением actualNumber внутри вашего if, используйте var вместо let для объявления переменной, а не константы.

**Неявное развернутые опциональные значения**

**(implicit unwrapped optionals)**

Как описано выше, опциональные значения обозначают, что константа или переменная может не иметь никакого значения. Мы можем проверить это, используя if и получить значение, используя восклицательный знак или опциональную связку.

Иногда из структуры приложения видно, что опциональная переменная или константа будет всегда иметь значение после того, как оно впервые было установлено. В этих случаях, полезно каким-то образом убрать нужду в проверке и «доставании» значения каждый раз, когда мы обращаемся к этой переменной.

Такие опциональные переменные определяются как «неявно развернутые». Вы можете объявить неявно развернутую опциональную переменную, поставив восклицательный знак вместо вопросительного при объявлении типа.

НЕявно развернутые опциональные константы и переменные полезны, когда известно, что значение будет постоянно присутствовать в переменной или константе с какого-то момента.

Неявно развернутая опциональная константа или переменная — это обычная опциональная переменная в своей сути, но может быть использована как неопциональная, без необходимости «доставать» ее значение при каждом доступе. Лучше всего посмотреть на примере:

let possibleString: String? = "Опциональная строка"

println(possibleString!) // необходимо использовать восклицательный знак, чтобы получить доступ к значению

// получим на экране "Опциональная строка"

let assumedString: String! = "Неявно развернутая опциональная строка"

println(assumedString) // не требуется восклицательного знака для доступа

// получаем на экране "Неявно развернутая опциональная строка"

Для удобства, можно считать, что неявно развернутая опциональная переменная или константа — это обычная опциональная переменная/константа, которой дано разрешение показывать свое значение автоматически, когда она используется. Вместо того, чтобы помещать восклицательный знак после опциональной переменной при каждом использовании, просто поместите восклицательный знак после названия типа при объявлении.

*При попытке доступа к неявно развернутой опциональной переменной, когда она еще не получила значение, вы получите ошибку времени исполнения (runtime error). То же самое произойдет, если вы примените восклицательный знак к обычной опциональной переменной, которая не содержит значения.*

Вы по-прежнему можете обращаться с неявно развернутой опциональной переменной как с обычной опциональной переменной, например для проверки на значение:

if assumedString != nil {

println(assumedString)

}

// выводим на экран "Неявно развернутая опциональная строка"

Вы также можете использовать ее с опциональной связкой для проверки и разворачивания значения в одной строке:

if let definiteString = assumedString {

println(definiteString)

}

// выводим на экран "Неявно развернутая опциональная строка"

*Неявно развернутая опциональная строка не должна использоваться в случаях, когда есть шанс, что переменная не будет содержать значения (будет nil) в какой-то момент времени. Всегда используйте обычный опциональный тип в случаях, если нужно проверять на nil в течении жизненного цикла переменной.*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.5**

1. *Создайте опциональную связку.*

**Утверждения (они же ассерты, они же assertions)**

Опциональные типы позволяют проверять значения, которые могут существовать или отсутствовать, и писать код, который красиво обрабатывает отсутствие значения. Однако, в некоторых случаях код не может продолжить выполнение в случае, когда значение не существует или не удовлетворяет определенным условиям. В этих ситуациях, вы можете вызвать assert (trigger an assertion) — это остановит выполнение кода и даст возможность провести отладку, чтобы выяснить, откуда взялось неподходящее значение.

**Отладка с утверждениями**

Утверждение — это проверка, проходящая во время исполнения, которая убеждается, что определенное логическое условие является правдой (true). Оно в буквальном смысле «утверждает», что условие верно. Используйте утверждения, чтобы убедиться, что определенное условие обязательно соблюдается прежде, чем код исполняется. Если условие является правдой, код выполняется так, как и должен, а если нет, то приложение выключается.

Утверждение пишется, используя глобальную функцию assert. Ему передается функция или выражение, возвращающее true или false, а также сообщение, которое отображается в случае, когда результат выражения или функции — false.

let age = -3

assert( age >= 0, "Возраст человека не может быть меньше 0")

// строчка выше заставляет сработать наше утверждение, т.к. возраст указан меньше нуля

В этом примере, исполнение кода продолжится только в случае, если возраст age >= 0, т.е. age должно быть неотрицательным. В случае, если оно отрицательно (как в нашем примере), утверждение будет неверным и завершит наше приложение.

Сообщение можно пропустить:

assert(age>=0)

//При этом, сообщение для функции assert не может содержать интерполяцию строк.

**Базовые операторы**

Swift поддерживает большинство стандартных операторов языка C и улучшает несколько возможностей для устранения частых ошибок при программировании. Оператор присваивания (=) не возвращает значение, что предотвращает его ошибочное использование вместо оператора равенства (==). Арифметические операторы ( +, -, \*, /, % и т.д.) определяют и запрещают переполнение значений, чтобы избежать неожиданных результатов, когда работаешь с числами которые становятся больше или меньше того, что позволяет диапазон принимающей переменной. Вы можете изменять поведение переполнения значения, используя операторы переполнения,

В отличии от C, Swift дает возможность выполнять оператор остатка от деления (%) для чисел с плавающей запятой. Swift также предоставляет два оператора диапазона ( a..b и a…b ), которых не было в C, как упрощенный оператор для выражения диапазона значений.

**Оператор Присваивания**

Оператор присваивания ( a = b ) инициализирует или обновляет значение a значением b :

let b = 10

var a = 5

a=b

//а сейчас равно 10

Если правая сторона присваивания это tuple с множеством значений, то его значения сразу раскладываются и присваиваются каждой переменной или константе

let (x, y ) = (1, 2)

//x сейчас равен одному, а y равен 2

В отличии от оператора присваивания в C и Objective-C , оператор присваивания в Swift не возвращает значение. Следующие выражение не допустимо:

if x = y {

// это не валидно, так как x = y не возвращает значение

}

Это предотвращает использование оператора присваивания (=) как оператора сравнения равенства (==). Не допуская if x = y, Swift помогает Вам избежать эти виды ошибок.

**Арифметические операторы**

Swift поддерживает четыре стандартных арифметических оператора для всех числовых типов:

Сложение (+)

Вычитание (-)

Умножение (\*)

Деление (/)

1 + 2       // равно 3

5 - 3       // равно 2

2 \* 3       // равно 6

10.0 / 2.5  // равно 4.0

В отличие от арифметических операторов в C и Objective-C, арифметические операторы в Swift не позволяют значениям переполняться по умолчанию. Вы можете изменять поведение переполнения значения, используя операторы переполнения (вроде a &+ b).

Оператор сложения также поддерживает конкатенацию строк :

"hello," + "world" //равно «hello,world"

**Оператор Остатка**

Оператор остатка (%) еще известен как оператор модуля в других языках. Однако, в Swift это звучит как *Оператор Остатка*, поскольку он работает одинаково и с отрицательными числами .

В Swift это может быть записано как:

9 % 4 //равно 1

Тот же метод применяется, когда рассчитывается остаток для отрицательного значения a:

-9 % 4 // равно - 1

Знак b игнорируется для отрицательных значений b. Это значит, что a % b и a % -b всегда возвращают тот же ответ.

**Остаток для чисел с плавающей запятой**

В отличии от оператора остатка C и Objective-C, в Swift оператор остатка от деления также может работать с числами с плавающей запятой:

8 % 2.5 // равно 0.5

В этом примере, 8 разделенное на 2.5 равно 3 с остатком 0.5, т.е. оператор остатка возвращает Double значение 0.5

**Операторы Инкремента и Декремента**

Как и в C, в Swift есть операторы инкремента ( ++ ) и декремента ( — ) как упрощенный вариант ( shortcut ) для увеличения и уменьшения значения числовой переменной на 1. Вы можете использовать эти операторы с переменными целого типа и с плавающей запятой.

var i = 0

++i      // i теперь равно 1

--i // i теперь равно 0

Символы ++ и — могут быть использованы как префикс и постфикс.

**Тернарный условный оператор**

let contentHeight = 40

let hasHeader = true

let rowHeight = contentHeight + (hasHeader ? 50 : 20)

//rowHeight равно 90

**Оператор объединения по нулевому указателю**

Оператор объединения по нулевому указателю (nil)  (a ?? b) извлекает опционал a , если он содержит значение, или возвращает значение по умолчанию b, если a является нулевым указателем (nil). Выражение a может быть только опционалом. Выражение b должно быть такого же типа, что и значение внутри a.

Оператор объединения по нулевому указателю является краткой записью следующего кода:

<span class="color-lime">a</span> != <span class="color-magenta">nil</span> ? <span class="color-lime">a</span>! : <span class="color-lime">b</span>

В вышеприведенном коде тернарный условный оператор и принудительное извлечение (a!) используются для обращения к значению внутри a, если a не равно nil, или для возвращения b в противном случае. Оператор объединения по нулевому указателю — это более элегантный, короткий и понятный способ одновременно проверить условие и извлечь значение.

В следующем примере оператор объединения по нулевому указателю выбирает между стандартным значением цвета и пользовательским:

let defaultColorName = "red"

var userDefinedColorName: String? // по умолчанию равно nil

var colorNameToUse = userDefinedColorName ?? defaultColorName

// userDefinedColorName равен nil, поэтому colorNameToUse получит значение по умолчанию — "red"

Переменная userDefinedColorName объявлена как строковый (String) опционал и по умолчанию равна nil. Так как userDefinedColorName является опционалом, ее значение можно анализировать посредством оператора объединения по нулевому указателю. В вышеприведенном примере этот оператор задает начальное значение для строковой (String) переменной colorNameToUse. Так как userDefinedColorName равно nil, выражение userDefinedColorName ?? defaultColorName возвратит значение defaultColorName, т. е. «red».

Если переменной userDefinedColorName присвоить отличное от nil значение и снова передать ее в оператор объединения по нулевому указателю, вместо значения по умолчанию будет использовано значение внутри userDefinedColorName:

userDefinedColorName = "green"

colorNameToUse = userDefinedColorName ?? defaultColorName

// userDefinedColorName не равно nil, поэтому colorNameToUse получит значение "green"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.6**

1. *Используя оператор = выполните присваивание кортежу (yaer, specialty, gruope).*
2. *Задайте переменной name значение используя оператор объединения по нулевому указателю. Значение по умолчанию «noname».*

**Операторы Диапазона**

Swift включает два оператора диапазона, которые упрощают обозначение диапазона значений

**Оператор Закрытого Диапазона**

Оператор закрытого диапазона (a…b ) определяет диапазон который идет от a до b включая значения a и b

Оператор закрытого диапазона полезен, когда итерации в диапазоне, в котором вы хотите использовать все значения, как for-in цикл:

for index in 1...5 {

println("\(index) times 5 is \(index \* 5)")

}

// 1 times 5 is 5

// 2 times 5 is 10

// 3 times 5 is 15

// 4 times 5 is 20

// 5 times 5 is 25

**Оператор Полузакрытого Диапазона**

Оператор полузакрытого диапазона (a..b) определяет диапазон который идет от a до b, не включая b. Называется так, поскольку включает первое значение, но не последнее.

Полузакрытый диапазона особенно полезен, когда вы работаете со списками типа массив (Array), где счет начинается с 0, а значит полезно использовать значение длина - 1:

let names = ["Anna", "Alex", "Brian", "Jack"]

let count = names.count

for i in 0..<count {

println("Person \(i + 1) is called \(names[i])")

}

// Person 1 будет Anna

// Person 2 будет Alex

// Person 3 будет Brian

// Person 4 будет Jack

Заметьте, что массив содержит четыре значения, но 0 …< count считает только до 3(индекс последнего элемента в массиве), поскольку используется полузакрытый диапазон.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.7**

1. *Задайте массив строк с названиями зимних месяцев и выведите его используя оператор закрытого и полузакрытого диапазона.*

**Логические операторы**

Логические операторы изменяют или комбинируют логические значения типа Boolean (булево) — true и false. Язык Swift, как и другие C-подобные языки, поддерживает три стандартных логических оператора:

логическое НЕ (!a)

логическое И (a && b)

логическое ИЛИ (a || b)

**Оператор логического НЕ**

Оператор логического НЕ (!a) инвертирует булево значение — true меняется на false, а false становится true.

let allowedEntry = false

if !allowedEntry {

 println("Доступ запрещен")

}

// напечатает "Доступ запрещен"

**Оператор логического И**

Оператор логического И (a && b) дает на выходе true тогда и только тогда, когда оба его операнда также равны true.

let enteredDoorCode = true

let passedRetinaScan = false

if enteredDoorCode && passedRetinaScan {

     println("Добро пожаловать!")

} else {

     println("Доступ запрещен")

}

// напечатает "Доступ запрещен"

**Оператор логического ИЛИ**

Оператор логического ИЛИ (a || b) является инфиксным и записывается в виде двух вертикальных палочек без пробела. С его помощью можно создавать логические выражения, которые будут давать true, если хотя бы один из операндов равен true.

**Комбинирование логических операторов**

Можно также составлять и более сложные выражения из нескольких логических операторов:

if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {

    println("Добро пожаловать!")

} else {

    println("Доступ запрещен")

}

// prints "Добро пожаловать!"

**String**

*Тип String в Swift крепко связан с классом NSString из фреймворка Foundation. Если вы работаете с этим фреймворком в Cocoa или Cocoa Touch, то весь NSString API доступен для вызова любой созданной вами переменной или константой типа String . Так же вы можете использовать String с любой API, которая требует экземпляр NSString.*

Задать пустое значение String, в качестве отправной точки для построения более длинной строки, либо назначить пустую строку литерал переменной, или инициализировать новый экземпляр типа String с синтаксисом инициализации, используйте следующий код:

var emptyString = ""               // пустой строковый литерал

var anotherEmptyString = String()  // синтаксис инициализации

// обе строки пусты и эквиваленты друг другу

Можно узнать пустое ли String значение, воспользовавшись его Boolean свойством isEmpty:

if emptyString.isEmpty {

    println("Пустая строка")

}

// "Пустая строка

В Swift Строка (String type) представляет собой коллекцию значений символов в указанном порядке. Вы можете получить доступ к отдельным значениям символов в строке с помощью перебора этой строки в цикле

for character in "Dog!" {

    println(character)

}

// D

// o

// g

// !

Можно создать отдельную константу типа Character или переменную из отдельного строкового литерала и присвоить ему тип Character:

let exclamationMark: Character = "!"

Строковые значения могут быть построены, передавая массив значений символов в качестве аргумента к его инициализатору

let catCharacters: [Character] = ["C", "a", "t", "!", ""]

let catString = String(catCharacters)

println(catString)

// отобразит "Cat!

**Интерполяция строк**

Интерполяция, интерполирование — в вычислительной математике способ нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений.

Интерполяция строки это способ построить новое значение строки из смеси констант, переменных, литералов и выражений, включив их значения внутри строки литерала. Каждый элемент, который вы вставляете в строку надо, обернуть в круглые скобки, и указать префикс «обратную косую черту»

let multiplier = 3

let message = "(multiplier) times 2.5 is (Double(multiplier) \* 2.5)"

// message равен "3 times 2.5 is 7.5"



В примере выше значение multiplier включено в строку как (multiplier). В свою очередь (multiplier) заменяется на фактическое значение константы multiplier, когда вычисляется интерполяция строки для создания конечного варианта.

Значение multiplier — это так же часть большего выражения в будущей строке. Это выражение высчитывает значение Double(multiplier) \* 2.5 и вставляет результат 7.5 в строку. В этом случае выражение записанное в виде (Double(multiplier) \* 2.5) является строковым литералом.

Выражение внутри скобок внутри интерполируемой строки не может содержать одиночные кавычки (») или обратный слэш (), так же как и не может содержать символ начала новой строки (n) или символ возврат каретки (r).

**Специальные символы в строковых литералах**

Строковые литералы могут включать в себя следующие специальные символы:

Уцелевшие специальные символы \0 (нулевой символ);

\\ (обратная косая черта);

\t(горизонтальная табуляция);

\n (перенос строки);

\r (возврат каретки);

\" (двойные кавычки);

 \' (одинарная кавычка)

Произвольное скалярное  Unicode, записать в виде \u{ п } , где п 1-8 шестнадцатеричный номер со значением равным действительной точки кода Unicode

Код ниже показывает четыре примера этих специальных символов. В wiseWords постоянные содержит два бежал двойные кавычек. В dollarSign , blackHeart и sparklingHeart постоянные демонстрации скалярное формат Unicode:

let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"

// "Imagination is more important than knowledge" - Einstein

let dollarSign = "\u{24}" // $, Unicode scalar U+0024

let blackHeart = "\u{2665}" // ♥, Unicode scalar U+2665

let sparklingHeart = "\u{1F496}" // , Unicode scalar U+1F496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.8**

1. *Используя интерполяцию задайте строку содержащую константу, переменную и выражение.*
2. *Запишите строковый литерал со специальным символом*

**Контроль управления**

if и switch используются для условий, for-in, for, while, repeat - while — для циклов. Скобки вокруг условий опциональны. Фигурные скобки вокруг тела — обязательны.

let individualScores = [75, 43, 103, 87, 12]

var teamScore = 0

for score in individualScores {

   if score > 50 {

      teamScore += 3

   } else {

      teamScore += 1

   }

}

teamScore

В if выражение должно возвращать Boolean, поэтому код типа if score { … } — это ошибка, никакого неявного сравнения с нулем не будет.

Можно использовать if и let вместе для работы со значениями, которые могут отсутствовать — они представлены, как опциональные значения. Опциональное значение может содержать значение или nil для обозначения того, что значение отсутствует:

var optionalString: String? = "Hello"

optionalString == nil

var optionalName: String? = "Anna"

var greeting = "Hello!"

if let name = optionalName {

   greeting = "Hello, \(name)"

}

Если опциональное значение равно nil, то условие вернет false и код будет пропущен. Иначе, опциональное значение будет присвоено константе и мы сможем использовать его внутри блока кода.

Switch поддерживает любой вид данных и широкое множество операторов сравнения — он не ограничен одними лишь целыми числами и сравнениями на равенство:

let vegetable = "red pepper"

switch vegetable {

   case "celery":

      let vegetableComment = "Add some raisins and make ants on a log."

   case "cucumber", "watercress":

      let vegetableComment = "That would make a good tea sandwich."

   case let x where x.hasSuffix("pepper"):

      let vegetableComment = "Is it a spicy \(x)?"

   default:

      let vegetableComment = "everything tastes good in soup"

}

После выполнения кода внутри switch-кейса, который совпал по условию, программа выходит из switch (а не продолжает идти по остальным свитчам, поэтому нет необходимости использовать break).

var userMark = 4

// проверка значения с использованием диапазона

switch userMark {

case 1..<4:

print("Экзамен не сдан!")

case 4...10:

print("Экзамен сдан!")

default:

assert(false, "Оценка \(userMark) вне доступного диапазона")

}

Как отмечалось ранее, после выполнения кода, расположенного в блоке case, происходит завершение работы конструкции switch-case. Однако в некоторых случаях требуется не завершать работу конструкции switch-case, а перейти к выполнению кода в следующем блоке case. Для этого в конце блока case указывается ключевое слово fallthrough:

/\* переменная типа Character

содержит уровень

готовности \*/

var level: Character = "Б"

// определение уровня готовности

switch level {

case "А":

print("Выключить все электрические приборы ")

fallthrough

case "Б":

print("Закрыть входные двери и окна ")

fallthrough

case "В":

print("Соблюдать спокойствие")

default:

break

}

}

В одном блоке case можно определить даже несколько возможных значений. Однако Swift позволяет в пределах одного блока case указать не только на значение рассматриваемого параметра, но и на зависимость от какого-либо условия. Данный функционал реализуется с помощью ключевого слова where в блоке case .

Ключевое слово where ставится в блоке case после перечисления значений, за ним следует логическое выражение, которое должно вернуть true или false . Код в блоке case выполняется, когда найдено совпадающее значение и условие where вернуло true. С помощью ключевого слова where вы можете также указывать на значение проверяемого в операторе switch параметра. В этом случае вместо значения блока case используйте уже знакомый вам символ нижнего подчеркивания.

var userMark = 4

switch userMark {

case \_ where userMark>1 && userMark<4:

print("Экзамен не сдан!")

case \_ where userMark >= 4:

print("Экзамен сдан!")

default:

assert(false, "Оценка \(userMark) вне доступного диапазона")

}

Здесь мы не указываем значение после ключевого слова case, а вместо него ставим символ пропуска нижнего подчеркивания, поэтому проверяется лишь переданное после where условие .

Ключевое слово case тоже может содержать несколько значений кортежа, перечисляемых через запятую, как и значения отдельных параметров .

Существует прием, который называется *связывание значений* (по аналогии с опциональным связыванием). Суть его в том, чтобы объявить новую переменную или константу и в автоматическом режиме инициализировать ее значением проверяемого параметра (или элемента проверяемого параметра) для его проверки с помощью ключевого слова where .

var dragonCharacteristic = ("green", 2.4)

// определение загона для поступившего дракона

switch dragonCharacteristic {

case ("green ", 0..<2):

print("In 1")

case ("red", 0..<2):

print("In 2")

case let(color, weight) where (color == " green " || color == "red") && weight >= 2:

print("In 3")

default:

print("Unknown")

}

let somePoint = (1, 1)

switch somePoint {

case (0, 0):

print("\(somePoint) is at the origin")

case (\_, 0):

print("\(somePoint) is on the x-axis")

case (0, \_):

print("\(somePoint) is on the y-axis")

case (-2...2, -2...2):

print("\(somePoint) is inside the box")

default:

print("\(somePoint) is outside of the box")

}

// Prints "(1, 1) is inside the box»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.9**

1. *Определите псевдоним Operation типу кортежа, содержащему три элемента со следующими именами: operandOne, operandTwo, operation. Первые два — это числа с плавающей точкой . Они будут содержать операнды для выполняемой операции. Третий элемент — строковое значение типа Character. Оно будет содержать указатель на проводимую операцию. Всего может быть четыре вида операций: +, -, \*, /.*
2. *Создайте константу типа Operation и заполните значения ее элементов произвольным образом, например (3.1, 33, "+") .*
3. *Используя конструкцию switch-case, вычислите значение операции, указанной в элементе operation созданного кортежа для операндов в его первых двух элементах . Оператор switch должен корректно обрабатывать любую из перечисленных операций.*
4. *В созданной константе замените символ операции другим произвольным символом и проверьте правильность работы конструкции switch-case .*

Можно использовать for-in для итерации по элементам, указав пару имен для каждого ключа- значения:

let interestingNumbers = [

   "Prime" : [2, 3, 5, 7, 11, 13],

   "Fibonacci": [1, 1, 2, 3, 5, 8],

   "Square": [1, 4, 9, 16, 25],

]

var largest = 0

for (kind, numbers) in interestingNumbers {

   for number in numbers {

      if number > largest {

         largest = number

      }

   }

}

largest

Переменная (kind, numbers), которая создается в цикле for-in, является локальной для данного цикла . То есть если существует внешняя одноименная переменная или константа, то ее значение не будет пересекаться с локальной переменной и все изменения локального параметра никак не повлияют на внешний.

Возможна ситуация, когда вам необходимо пройти от большего числа к меньшему. При попытке указать диапазон от большего к меньшему, например (10…1), Xcode сообщит об ошибке. Для этого используйте метод reversed()

var totalSum = 0

for i in (1...10).reversed() {

totalSum += i // (10 times)

}

totalSum //55

Если необходимо пройти определенный числовой интервал с некоторым шагом, можно использовать два одноименных метода: stride(from:through:by:) и stride(from:to:by:) .

for i in stride(from:1, through:5, by:2) {

print(i) // 1 3 5

}

for i in stride(from:1, to:5, by:2) {

print(i) // 1 3

}

Для обработки многомерных конструкций вроде вложенных коллекций вы можете вкладывать одни циклы в другие.

**Операторы повторения while и repeat while**

Операторы while и repeat while позволяют выполнять блок кода до тех пор, пока проверяемое условие истинно. Можно перенести условие в конец и использовать repeat-while:

var n = 2

while n < 100 {

   n=n\*2

}

n

var m = 2 repeat {

   m=m\*2

} while m < 100

m

**Управление циклами**

В Swift, по аналогии с другими языками программирования, есть два оператора, способных влиять на ход работы циклов, — это операторы break и continue.

Оператор continue предназначен для перехода к очередной итерации, игнорируя следующий за ним код .

for i in 1...10 {

if i%2 == 0 {

continue

} else {

print(i)

}

}

Оператор break предназначен для досрочного завершения работы цикла. При этом весь последующий код в теле цикла игнорируется.

Может возникнуть ситуация, когда из внутреннего цикла необходимо прервать выполнение внешнего, — для этого в Swift существуют метки

mainLoop: for i in 1...5 {

for y in i...5 {

if y == 4 && i == 2{

break mainLoop

}

print("\(i) — \(y)")

}

}

**Оператор раннего выхода guard**

Оператор guard называется оператором раннего выхода. Подобно оператору if, он проверяет истинность переданного ему условия. Отличие его в том, что он выполняет блок кода, расположенный в фигурных скобках, только в том случае, если условие вернуло значение false.

guard утверждение else {

// тело оператора

}

После ключевого слова guard следует некоторое проверяемое утверждение. Если утверждение возвращает true, то тело оператора игнорируется и управление переходит следующему за guard коду Если утверждение возвращает false, то выполняется код внутри тела оператора.

Для данного оператора существует ограничение: его тело должно содержать один из следующих операторов — return, break, continue, throw.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 2.10**

1. *Задайте массив целых. Выведите его значения используя for-in,*

*Подсчитайте сумму элементов по while и repeat с диапазоном, с шагом, в обратном порядке.*

1. *Продемонстрируйте работу оператора guard*

**Вопросы**

1. Как объявить константу, переменную?
2. Обязательно ли указывать тип?
3. Какие правила именования переменных и констант?
4. Перечислите операторы вывода.
5. Перечислите и поясните все виды комментариев swift.
6. Перечислите и объясните фундаментальные типы данных swift.
7. Какой размер у Int?
8. Какое дополнительное форматирование может быть у числовых литералов? Приведите пример.
9. Как задать многострочный строковый литерал?
10. Что такое typealias?
11. Как определить кортеж (tuple)?
12. Поясните на примере как разобрать (декомпозировать) кортеж.
13. Какие правила сравнения кортежей существуют в swift?
14. Что такое опциональные значения? Как обратиться к опциональному значению?
15. Как описать опциональный String? Что он может содержать?
16. Для чего используют специальное значение nil?
17. Что такое «неявно развернутые» опциональные переменные ?
18. Пусть

let pString: String? = "234"

Как получить значение?

1. Как происходит присваивание кортежей?
2. Как выполняется оператор остатка (%) с отрицательными числами и с числами с плавающей запятой?
3. Поясните выполнение оператора объединения по нулевому указателю (??).
4. Что такое оператор закрытого и полузакрытого диапазона?
5. Приведите пример интерполяции строк.
6. Какие операторы циклов могут использоваться в swift?
7. В каких случаях указывается ключевое слово fallthrough в конструкции switch-case?
8. В каких случаях используется ключевое слово where в блоке switch-case?
9. Как задать шаг в цикле?
10. Поясните, как использовать оператор guard?