# **№ 19 Введение в swift**

Следует предварительно ознакомиться с приведенным здесь материалом (до лабораторной работы изучить теорию). Он включает теорию и задания для выполнения. На лабораторной работе необходимо выполнить задания (выделены желтым цветом) и показать преподавателю.

## Теория + Задания

**Swift**  — мультипарадигменный объектно-ориентированный язык программирования, созданный компанией Apple для разработчиков iOS и OS X.

Swift работает с фреймворками  Cocoa и Cocoa Touch и совместим с основной кодовой базой Apple, написанной на Objective-C. Язык поддерживается в среде программирования Xcode 6; программы на нем компилируются при помощи Apple LLVM и используют рантайм Objective-C, что делает возможным использование обоих языков (а также pure С и С++) в рамках одной программы.

Особенность Swift состоит в поддержке инновационной функции playground. Она делает возможным проводить любые эксперименты, видеть моментально их результат, не требуя запуска приложения.

1. Не нужно импортировать библиотеку для работы с вводом/выводом или работой со строками.
2. Код, написанный в глобальной области видимости (global scope), используется как входная точка для приложения, поэтому функция main() не нужна.
3. Также не нужно ставить точки с запятой в конце каждого оператора.

Задание 1

Все последующие примеры необходимо будет проверять и вводить в IDE XCode - <https://developer.apple.com/download/> (он уже у вас установлен).

Далее надо завести себе Apple ID на <https://www.apple.com/> - но в лаборатории он тоже установлен.

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

В нем содержится:

1) UIKit — фреймворк для построения пользовательского интерфейса. Он содержит в себе различные элементы управления (кнопки, текстовые поля, панель навигации и другие).

2) Cocoa Touch — это коллекция библиотек Apple, которая содержит множество функциональных фреймворков iOS, например UIKit для построения пользовательского интерфейса, SpriteKit для создания 2D-игр, MapKit для работы с картами, Core Graphics для рисования, Core Animation для анимации объектов и т. д.

3) NeXTSTEP — ОС компании NeXT.

4) iOS Simulator — инструмент, который входит в состав среды Xcode. С его помощью можно имитировать запуск приложений на виртуальных копиях iPhone и iPad.

5) Playgrounds — миниатюрные тестовые программы Swift, в рабочей области которых можно вводить код и моментально видеть результаты его выполнения. Они не используются в рабочих продуктах, но подходят для обучения.

**Playground**

В переводе на русский «песочница», представляет из себя окружение для того, чтобы изучить Swift. Выглядит это так: вы вводите строку кода и моментально видите результат его выполнения. После того, как вы убедитесь, что написанный вами код дает нужный результат, вы можете просто взять и перенести его в свой проект.

Например, при помощи Playground вы можете решить следующие задачи:

• освоить синтаксис Swift

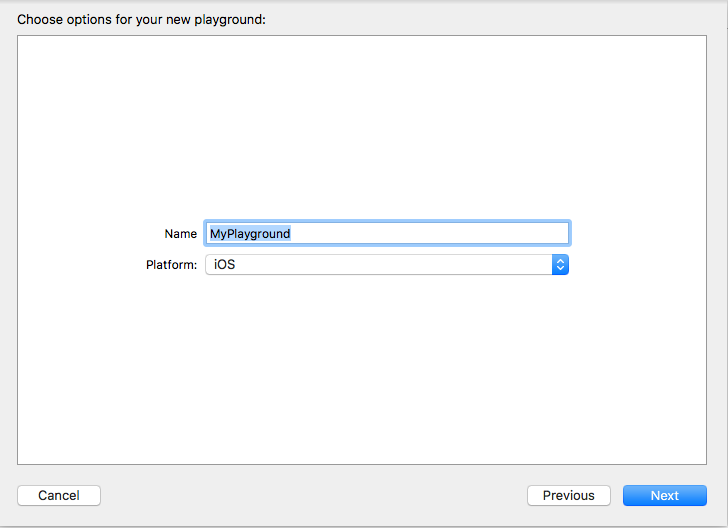
• поэкспериментировать с новыми API

• разработать новый алгоритм и понаблюдать за каждым шагом его выполнения

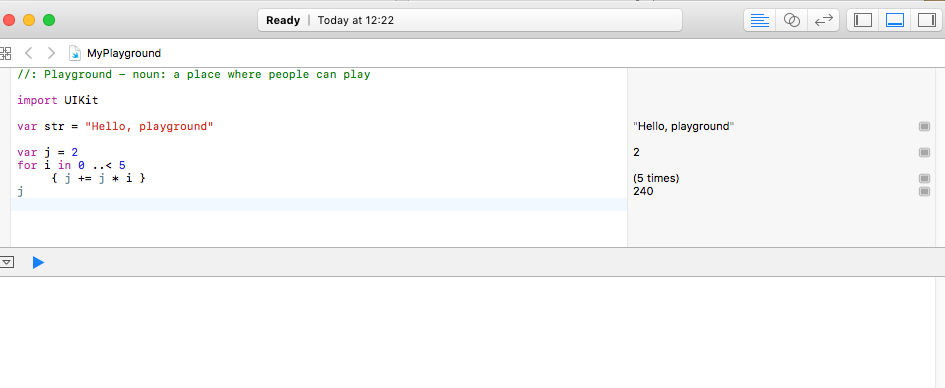
• провести простые арифметические вычисления

Задание 2

Создайте Playground



Теперь вам нужно придумать имя для своего Playground, нажать кнопку Next и сохранить его куда-нибудь.

Наберите небольшой код и посмотрите результаты

Правый столбец называется панелью результатов (results sidebar) и для каждой строки в нем показано значение, которое будет получено в результате ее выполнения. Например, для строки 10 в столбце результата будет значение этой переменной.

Обратите внимание на строку 8, в ней панель результатов содержит не результат выражения, а то, сколько раз это выражение было посчитано (5 раз). Мы можем увидеть, какие значения были получены для этой строки. Чтобы это сделать, надо нажать на маленький прямоугольник рядом с текстом (5 times). После чего Playground станет выглядеть так:

Теперь внизу Playground отображается панель Timeline. Еще ее можно открыть, выбрав в меню пункт View->Assistant Editor->Show Assistant Editor.

Timeline позволяет увидеть, как менялось значение выбранного выражение во времени. Чтобы это сделать, используйте слайдер внизу. Попробуйте поперетаскивать его.

Кроме того, в Timeline показывает «Console Output». Это тот текст, который бы вывела ваша программа в консоль устройства или терминал.

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

**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)

Есть простой способ включить значения в строки:

let apples = 3

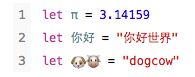
let oranges = 5

let appleSummary = "I have \(apples) apples."

let fruitSummary = "U have \(apples + oranges) pieces of fruit."

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.

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 = "Bonjour"

println(friendlyWelcome)

// получим на экране Bonjour!

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

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 = "Bonjour"

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

// печатает The current value of friendlyWelcome is Bonjour!&nbsp;

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.

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

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.

**UInt**

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

Задание 3

Определите несколько переменных с let и var с явным указанием типа и без (Int, Double, String, Float, Bool)

Выведите значения на консоль. В строках добавьте выражения или числовые переменные объявленные ранее.

Используя тип Int создайте переменную. Выведите ее диапазон. Проинициализируйте.

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

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 4

Создайте числовые литералы для числа 5 в двоичном, десятичном и шестнадцатеричном представлении.

Создайте литерал для числа 0.36 в экспоненциальной форме.

Используйте дополнительное форматирование для числа 12305670.9805

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

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

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)».

Вы можете создавать кортежи из любой перестановки типов и они могут содержать сколько угодно нужных вам типов. Можно создать кортежи (Int, Int, Int) или (String, Bool), в общем — любой, какой вам нужно.

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 "Сообщене - 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 "Сообщене - OK"

Задание 5

Создайте кортеж (2017, “ISIT”, 1).

Задайте имена элементам кортежа.

Выведите элементы двумя способами - по индексу, по имени.

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

Опциональные значения

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

**«у меня есть значение и оно равно х»**

или

**«у меня нет никакого значения»**

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 6

Создайте две строки с числовым значением и без. Выполните преобразование к опциональному Int.

Объявите переменную типа Int?

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 — прим. переводчика).

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 этот ад 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 7

Создайте опциональную связку.

Создайте НЕявно развернутую опциональную переменную.

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

**Массивы**

Полная форма указания типа массива в языке Swift:

Array<SomeType>

, где SomeType является типом значений, которые могут содержаться в массиве. Вы также можете написать тип массива в сокращенном виде как [SomeType] . Хотя обе формы функционально идентичны, но в сокращенном виде запись, является предпочтительной.

Создание и инициализация массива

Вы можете создать пустой массив определенного типа, используя синтаксис инициализации:

var someInts = [ Int ]()

println ( "someInts имеет тип [Int] и содержит \( someInts . count ) элементов." )

// prints "someInts элементов [Int] и содержит 0 элементов."

Если контекст уже обеспечивает информацию о типе, как аргумент функции или типизированная переменная или константа, то вы можете создать пустой массив с пустым литералом массива, который пишется как [] (пустая пара квадратных скобок).

someInts . append ( 3 )

// someInts теперь содержит 1 значение типа Int

someInts = []

// someInts теперь пустой массив, но по-прежнему остается типом [Int]

Тип Array в языке Swift так же имеет свой инициализатор для создания массива определенного размера с присвоением значения по умолчанию всем исходным элементам. Вы передаете инициализатору число элементов, которое должно быть добавлено массиву (count) и значение по умолчанию с подходящим типом (repeatedValue):

var threeDoubles = [ Double ]( count : 3 , repeatedValue : 0.0 )

// threeDoubles имеет тип [Double], и равен [0.0, 0.0, 0.0]

Вы можете создать новый массив путем сложения двух существующих массивов с совместимыми типами при помощи оператора сложения ( + ). Тип нового массива выводится из типа двух складываемых массивов:

var anotherThreeDoubles = [Double](count: 3, repeatedValue: 2.5)

// anotherThreeDoubles выведен тип [Double], и равен [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles

// sixDoubles выведен тип [Double], и равен [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]

Литералы массива

Вы можете инициализировать массив литералом массива, что является сокращенной формой для записи одной или нескольких величин в коллекцию массива. Литерал массива записывается в виде списка значений, разделенных запятыми, в окружении пары квадратных скобок:

[Значение 1, значение 2, значение 3]

Пример ниже создает массив с именем shoppingList для хранения String значений:

var shoppingList: [String] = ["Eggs", "Milk"]

// shoppingList был инициализирован двумя исходными элементами

shoppingList переменная объявлена как «массив строковых значений», записанная, как [String]. Поскольку для этого массива указан тип хранимых значений как String, то только String значения ему и разрешено хранить.. Здесь shoppingList массив инициализируется с двумя Sting значениями ( «Eggs» и «Milk» ), записанных в  литерал массива .

Благодаря выведению типов Swift, вы не обязаны писать типа массива, если вы инициализируете его литералом, содержащим элементы одного типа. Значит инициализация shoppingList может быть записано вот так:

var shoppingList = [ "Eggs" , "Milk" ]

Поскольку все значения в литерале массива имеют один и тот же тип, Swift может сделать вывод, что [String] правильный тип для переменной shoppingList.

Доступ и изменение массива

Вы можете получать значения и изменять массив через его методы и свойства, или с помощью синтаксиса index.

Чтобы получить количество элементов в массиве воспользуемся read-only свойством count:

println ( "Список покупок содержит \( shoppingList . count ) элемента." )

// Выведет "Список покупок содержит 2 элемента."

Используйте свойство isEmpty типа Bool для быстрой проверки равенства свойства count нулю (0):

if shoppingList . isEmpty {

println ( "Список покупок пустой." )

} else {

println ( "Список покупок не пустой." )

}

// выведет "Список покупок не пустой."

Вы можете добавить новый элемент в конец массива путем вызова метода append(\_:):

shoppingList . append ( "Flour" )

// shoppingList теперь сдержит 3 элемента

Как альтернатива методу append, вы можете использовать оператор (+=) для добавления нового элемента в конец массива:

shoppingList += [ "Baking Powder" ]

// shoppingList теперь содержит 4 элемента

shoppingList += [ "Chocolate Spread" , "Cheese" , "Butter" ]

// shoppingList теперь содержит 7 элементов

Чтобы получить значение из массива, используйте синтаксис index, передавая значения индекса в квадратных скобках сразу после имени массива:

var firstItem = shoppingList [ 0 ]

// firstItem имеет значение "Eggs"

Массивы в Swift всегда нумеруются с нуля (0).

Вы можете использовать синтаксис index для изменения существующего значения в данном индексе:

shoppingList [ 0 ] = "Six eggs"

// теперь первое значение списка имеет значение "Six eggs" в место "Eggs"

Вы также можете использовать синтаксис index для изменения сразу диапазона значений, даже если набор значений имеет разную длину, чем в указанном диапазоне. Следующий пример заменяет «Chocolate Spread» , «Cheese» , и «Butter» на значение «Bananas» и «Apples» :

CshoppingList [ 4 ... 6 ] = [ "Bananas" , "Apples" ]

// shoppingList содержит 6 элементов

*Вы не можете использовать синтаксис Index, чтобы добавить новый элемент в конец массива.*

Для того чтобы вставить элемент на определенную позицию в массиве, нужно использовать метод insert(atIndex:):

shoppingList.insert("Maple Syrup", atIndex: 0)

// shoppingList теперь содержит 7 элементов

// "Maple Syrup" теперь первый элемент списка

Вызов метода insert вставляет новый элемент со значением «Maple Syrup» в самое начало массива.

Тоже самое происходит и при удалении элементов из массива с помощью метода removeAtIndex.

let mapleSyrup = shoppingList.removeAtIndex(0)

// элемент с индексом 0 был удален

// shoppingList теперь содержит 6 элементов, Maple Syrup удален

// константа mapleSyrup теперь равна удаленной строке "Maple Syrup"

*Если вы попробуете использовать синтаксис index, чтобы получить или назначить новое значение индексу, который находиться за пределами существующего массива, то сработает runtime ошибка.*

Все пробелы внутри массива закрываются существующими элементами, то есть происходит сдвиг элементов по индексу. Теперь новый элемент с индексом 0 будет «Six eggs»:

firstItem = shoppingList [ 0 ]

// firstItem теперь равно "Six eggs"

 Если нужно удалить последний элемент массива, то используйте метод removeLast() вместо removeAtIndex( для избежание дополнительного вызова свойства count. Как и removeAtIndex(:), метод removeLast() так же возвращает значение удаленного элемента:

let apples = shoppingList.removeLast()

// последний элемент массива был удален

// shoppingList теперь содержит 5 элементов, и нет "Apples"

// константа apples теперь равна удаленной строке "Apples"

Перебор массива

Вы можете перебрать весь набор значений в массиве циклом (for — in):

for item in shoppingList {

println ( item )

}

// Six eggs

// Milk

// Flour

// Baking Powder

// Bananas

Если вам нужно значение каждого элемента массива и его индекс, то необходимо использовать глобальную функцию enumerate для итерации по массиву. Функция enumerate возвращает кортеж, состоящий из индекса и значения элемента массива. Вы можете разложить кортеж во временные константы или переменные, как часть итерации:

for ( index , value ) in enumerate ( shoppingList ) {

println ( "Item \( index + 1 ) : \( value ) " )

}

// Item 1: Six eggs

// Item 2: Milk

// Item 3: Flour

// Item 4: Baking Powder

// Item 5: Bananas

Задание 8

Создайте массив строк. Добавьте два элемента. Удалите один с начала. Выведите значения массива c for. Вставьте два элемента в конец. Выведите массив с использованием enumerate (кортеж – индекс и значение).

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

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 ассерт (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 равенста (==). 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 .

Чтобы вычислить 9 % 4, сначала посчитайте сколько «4» поместится в «9»

Вы можете поместить две «4» внутри «9» и у вас останется 1.

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 9

Используя оператор = выполните присваивание кортежу (yaer, specialty, gruope).

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 умножить на 5 будет 5

// 2 умножить на 5 будет 10

// 3 умножить на 5 будет 15

// 4 умножить на 5 будет 20

// 5 умножить на 5 будет 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 10

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:

Cif 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")

}

// prints "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!

Интерполя́ция строк

Интерполя́ция, интерполи́рование — 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 литералax

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 11

Используя интерполяцию задайте строку содержащую константу, переменную и выражение.

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

Контроль управления

if и switch используются для условий, for-in, for, while, do-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. Для создания «опционального значения» используется знак вопроса:

var optionalString: String? = "Hello"

optionalString == nil

var optionalName: String? = "John Appleseed"

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).

Можно использовать for-in для итерации по элементам в словаре, 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

While используется для повторения блока кода, пока не изменится условие. Можно перенести условие в конец и использовать do-while:

var n = 2

while n < 100 {

   n=n\*2

}

n

var m = 2 do {

   m=m\*2

} while m < 100

m

Можно хранить индекс в цикле — или используя .. для создания диапазона индексов, или написав явную инициализацию, условие и инкремент.

Следующие два цикла делают одно и то же:

var firstForLoop = 0

for i in 0..3 {

   firstForLoop += 1

}

firstForLoop

var secondForLoop = 0;

for var i = 0; i < 3; ++i {

   secondForLoop += 1

}

secondForLoop

Задание 12

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

Подсчитайте сумму элементов по do while с диапазоном

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

**Функции и замыкания (closures)**

Используйте func для объявления функции. Функция вызывается по ее имени и списку параметров в скобках. Используйте -> для отделения имен параметров и типов от возвращаемого значения функции:

func greet(name: String, day: String) -> String {

return "Hello \(name), today is \(day)."

}

greet("Bob", "Tuesday")

Используйте кортеж (tuple) для возврата нескольких значений из функции. Элементы набора могут быть отнесены либо по имени или по номеру.

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

    var min = scores[0]

    var max = scores[0]

    var sum = 0

    for score in scores {

        if score > max {

            max = score

        } else if score < min {

            min = score

        }

        sum += score

    }

    return (min, max, sum)

}

let statistics = calculateStatistics([5, 3, 100, 3, 9])

statistics.sum

statistics.2

Функции могут использовать разное число аргументов, собирая их в массив:

func sumOf(numbers: Int...) -> Int {

    var sum = 0

    for number in numbers {

        sum += number

    }

    return sum

}

sumOf()

sumOf(42, 597, 12)

Функции могут быть вложенными. Вложенные функции имеют доступ к переменным, объявленным во внешнем функции. Используйте вложенные функции для организации кода длинной и сложной функции.

func returnFifteen() -> Int {

    var y = 10

    func add() {

        y += 5

    }

    add()

    return y

}

returnFifteen()

Функции по своей сути являются типом — то есть вы можете возвращать функцию из функции:

func makeIncrementer() -> (Int -> Int) {

    func addOne(number: Int) -> Int {

        return 1 + number

    }

    return addOne

}

var increment = makeIncrementer()

increment(7)

Функция может принимать другую функцию, как ее аргумент:

func hasAnyMatches(list: [Int], condition: Int -> Bool) -> Bool {

    for item in list {

        if condition(item) {

            return true

        }

    }

    return false

}

func lessThanTen(number: Int) -> Bool {

    return number < 10

}

var numbers = [20, 19, 7, 12]

hasAnyMatches(numbers, lessThanTen)

Функции — это на самом деле специальный случай замыканий. Вы можете написать замыкание без имени, окружив код фигурными скобками. Используйте in для разделения аргументов и типа возвращаемого значения от тела замыкания:

numbers.map({

    (number: Int) -> Int in

    let result = 3 \* number

    return result

})

У вас есть несколько опций для написания замыканий более кратко. Когда тип замыкания уже известен, например обратный вызов делегата (callback), вы можете пропустить тип его параметров, тип возвращаемого значения или и то и другое. Однострочное замыкание в примере ниже возвращает значение своего единственного выражения:

let mappedNumbers = numbers.map({ number in 3 \* number })

mappedNumbers

Вы можете ссылаться на параметры по номеру, вместо его имени — этот подход особенно удобен в очень коротких замыканиях. Замыкание, переданное как последний аргумент для функции, может появляться сразу после скобок:

let sortedNumbers = sorted(numbers) { $0 > $1 }

sortedNumbers

Задание 13

Напишите функцию, которая принимает массив целых и возвращает кортеж, содержащий сумму, среднее значение и максимальное

Напишите функцию, которая возвращает другую функцию.

Напишите функцию, которая может принимать другую функцию, как ее аргумент.

Напишите замыкание

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

**Объекты и классы**

Используйте слово class с именем класса для создания класса. Объявление свойства класса делается так же, как объявление обычной переменной или константы — с тем лишь исключением, что оно теперь находится в контексте класса. Аналогично, методы и функции задаются внутри класса:

class Shape {

    var numberOfSides = 0

    func simpleDescription() -> String {

        return "A shape with \(numberOfSides) sides."

    }

}

Создается объект из класса, используя скобки после имени класса. Используйте синтаксис с точкой для доступа к свойствам и методам объекта:

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

Эта версия класса Shape не имеет одного важного свойства — инициализатора, который настроит объект при его создании. Используем метод init для этого:

class NamedShape {

    var numberOfSides: Int = 0

    var name: String

    init(name: String) {

        self.name = name

    }

    func simpleDescription() -> String {

        return "A shape with \(numberOfSides) sides."

    }

}

Обратите внимание на то, что для доступа к своим свойствам класс использует self. Аргументы инициализатору передаются так же, как при вызове функции:

var shape = NamedShape("My shape")

Каждое свойство должно иметь значение — заданное или при объявлении (как в случае numberOfSides), или в инициализаторе (как в случае name).

Используйте метод deinit для создания деинициализатора, если нужно перед уничтожением объекта выполнить какую-то очистку.

Классы могут наследоваться. Наследующие классы включают имя суперкласса (их родителя) после своего имени. При этом, классу не обязательно наследоваться от какого-то другого класса, в этих случаях имя родителя можно пропустить.

Класс-наследник может определять методы своего родителя, используя слово override — в противном случае, компилятор сообщит об ошибке.

class Square: NamedShape {

    var sideLength: Double

    init(sideLength: Double, name: String) {

        self.sideLength = sideLength

        super.init(name: name)

        numberOfSides = 4

    }

    func area() ->  Double {

        return sideLength \* sideLength

    }

    override func simpleDescription() -> String {

        return "A square with sides of length \(sideLength)."

    }

}

let test = Square(sideLength: 5.2, name: "my test square")

test.area()

test.simpleDescription()

В дополнение к простым хранимым свойствам, свойства могут также иметь getter и setter методы, т.е. методы, используемые для получения и установки их значения.

Например:

class EquilateralTriangle: NamedShape {

    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {

        self.sideLength = sideLength

        super.init(name: name)

        numberOfSides = 3

    }

    var perimeter: Double {

        get {

            return 3.0 \* sideLength

        }

        set {

            sideLength = newValue / 3.0

        }

    }

    override func simpleDescription() -> String {

        return "An equilateral triangle with sides of length \(sideLength)."

    }

}

var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")

println(triangle.perimeter)

triangle.perimeter = 9.9

println(triangle.sideLength)

В сеттере для свойства perimeter, newValue — это псевдоним для нового значения, которое передается. Можно предоставить другой псевдоним, указав его в скобках после set.

Обратите внимание, что инициализатор для EqualiateralTriangle имеет три разных шага:

1 Устанавливает значение свойств, объявляемый подклассом;

2 Вызывает инициализатор класса-предка;

3 Изменяет значение свойств, объявленных предком.

Любые дополнитеьлные действия так же можно сделать на последнем шаге.

Если вам не нужно вычислять свойство, но по прежнему нужно предоставить код, который будет запущен до и после получения нового значения, используйте willSet и didSet. Например, класс ниже проверяет, что длина стороны треугольника всегда такая же, как длина стороны квадрата:

class TriangleAndSquare {

    var triangle: EquilateralTriangle {

        willSet {

            square.sideLength = newValue.sideLength

        }

    }

    var square: Square {

        willSet {

            triangle.sideLength = newValue.sideLength

        }

    }

    init(size: Double, name: String) {

        square = Square(sideLength: size, name: name)

        triangle = EquilateralTriangle(sideLength: size, name: name)

    }

}

var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")

println(triangleAndSquare.square.sideLength)

println(triangleAndSquare.triangle.sideLength)

triangleAndSquare.square = Square(sideLength: 50, name: "larger square")

println(triangleAndSquare.triangle.sideLength)

Методы в классах имеют одно важное отличие от функций. Имена параметров в функциях используются только внутри функции, но имена параметров в методах используются так же при вызове метода (кроме первого параметра). По умолчанию, метод имеет одно и то же имя параметра при вызове и внутри самого метода. Но можно указать второе имя, используемое в самом методе:

class Counter {

    var count: Int = 0

    func incrementBy(amount: Int, numberOfTimes times: Int) {

        count += amount \* times

    }

}

var counter = Counter()

counter.incrementBy(2, numberOfTimes: 7)

Работая с опциональными значениями, вы можете написать ? перед операциями типа методов, свойств и обращений к элементам массива или словаря. Если значение перед ? — nil, то все после ? игнорируется и значение всего выражения — nil. Иначе, опциональное значение используется для вычисления выражения. В обоих случаях, значение всего выражение также является опциональным значением:

let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")

let sideLength = optionalSquare?.sideLength

Задание 14

Напишите класс Student со свойствами name и course, функцией и инициализатором. Создайте объект. Поменяйте его свойства.

Используя наследование от Student создайте SuperStudent с дополнительным свойством value, которое имеет set и get. Переопределите метод из Student. Создайте объект и вызовите его метод.

**Перечисления (enumerations) и структуры**

enum используется для создания перечисления. Как классы и другие именованные типы, перечисления могут иметь ассоциированные с ними методы:

enum Rank: Int {

    case Ace = 1

    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten

    case Jack, Queen, King

    func simpleDescription() -> String {

        switch self {

        case .Ace:

            return "ace"

        case .Jack:

            return "jack"

        case .Queen:

            return "queen"

        case .King:

            return "king"

        default:

            return String(self.rawValue)

        }

    }

}

let ace = Rank.Ace

let aceRawValue = ace.rawValue

В примере выше, исходное значение перечисления имеет тип Int, поэтому мы указали только первое — Ace равное 1. Остальные были добавлены по очереди (2, 3, 4 и т.д.). Можно также использовать строки или дробные числа в качестве исходных значений перечисления.

Используйте init?(rawValue:) для перехода от исходного значения к значению перечисления и обратно.

if let convertedRank = Rank(rawValue: 3) {

    let threeDescription = convertedRank.simpleDescription()

}

Значения элементов перечисления — это настоящие значения, а не просто другой способ записи исходных значений. По сути, в случаях, где нет разумного исходного значения, его не обязательно подставлять:

enum Suit {

    case Spades, Hearts, Diamonds, Clubs

    func simpleDescription() -> String {

        switch self {

        case .Spades:

            return "spades"

        case .Hearts:

            return "hearts"

        case .Diamonds:

            return "diamonds"

        case .Clubs:

            return "clubs"

        }

    }

}

let hearts = Suit.Hearts

let heartsDescription = hearts.simpleDescription()

Обратите внимание на то, как мы используем Hearts в двух разных случаях: когда присваиваем значение константе hearts, мы используем полное имя Suit.Hearts, т.к. константа не имеет определенного типа. Внутри switch’a, мы используем .Hearts, т.к. находимся внутри self. Можно использовать эту сокращенную форму всегда, когда тип значения уже известен.

Используйте struct, чтобы создать структуру. Структуры поддерживают многое из поведения классов, включая методы и инициализаторы. Одно из самых важных отличий между структурами и классами заключается в том, что структуры всегда копируются, когда передаются внутри кода, а классы — всегда передаются по ссылке.

struct Card {

    var rank: Rank

    var suit: Suit

    func simpleDescription() -> String {

        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"

    }

}

let threeOfSpades = Card(rank: .Three, suit: .Spades)

let threeOfSpadesDescription = threeOfSpades.simpleDescription()

Объект перечисления может иметь значения, ассоциирующиеся с объектом. Объекты одного и того же члена перечисления могут иметь разные значения, ассоциированные с ними. Вы предоставляете ассоциированные значения, когда создаете объект. Ассоциированные значения и исходные значения различны: исходное значение члена перечисления одинаковы для всех его объектов, вы задаете его при определении перечисления. Например, рассмотрим случай запроса времени заката и рассвета с сервера. Сервер отвечает или информацией, или ошибкой.

enum ServerResponse {

    case Result(String, String)

    case Error(String)

}

let success = ServerResponse.Result("6:00 am", "8:09 pm")

let failure = ServerResponse.Error("Out of cheese.")

switch success {

case let .Result(sunrise, sunset):

    let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)."

case let .Error(error):

    let serverResponse = "Failure...  \(error)"

Задание 15

Создайте перечисление с месяцами. Напишите в перечислении функцию, которая возвращает по месяцу пору года.

Используйте перечисление для установки месяца. Определите пору года.

Создайте перечисление ассоциируемое с объектами Student и SuperStudent

**Протоколы и расширения**

Используйте слово protocol для объявления протокола.

protocol ExampleProtocol {

    var simpleDescription: String { get }

    mutating func adjust()

}

Классы, перечисления и структуры могут соответствовать протоколам:

class SimpleClass: ExampleProtocol {

    var simpleDescription: String = "A very simple class."

    var anotherProperty: Int = 69105

    func adjust() {

        simpleDescription += "  Now 100% adjusted."

    }

}

var a = SimpleClass()

a.adjust()

let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {

    var simpleDescription: String = "A simple structure"

    mutating func adjust() {

        simpleDescription += " (adjusted)"

    }

}

var b = SimpleStructure()

b.adjust()

let bDescription = b.simpleDescription

Обратите внимание на ключевое слово mutating, которое обозначает метод, модифицирующий структуру. Объявление класса не требует добавления слова mutating, т.к. методы класса всегда могут модифицировать класс.

Используйте слово extension (расширение) для добавления функциональности существующему типу, например новых методов или вычисленных свойств. Вы можете использовать расширение для добавления совместимости с протоколом типу, который объявлен в другом месте, или даже типу, который вы импортировали из библиотеки или фреймворка:

extension Int: ExampleProtocol {

    var simpleDescription: String {

        return "The number \(self)"

    }

    mutating func adjust() {

        self += 42

    }

}

7.simpleDescription

Вы можете использовать имя протокола, как любой другой именованный тип — например, создать коллекцию объектов, которые имеют разные типы, но все соответствуют одному протоколу. Когда вы работаете со значениями, тип которых — протокол, методы вне объявления протокола недоступны.

let protocolValue: ExampleProtocol = a

protocolValue.simpleDescription

Хотя переменная protocolValue будет иметь тип SimpleClass во время исполнения, компилятор работает с ней как с переменной типа ExampleProtocol — это значит, что вы не сможете случайно обратиться к методам или свойствам, которые класс реализует в дополнение к протоколу.

Задание 16

Объявите протокол.

Класс Student должен соответствовать протоколу

Используя слово extension добавьте функциональность к типу Int для совместимости с вашим протоколом.

**Общие функции и типы**

Напишите имя внутри <>, чтобы создать общую функцию или тип:

func repeat<Item>(item: Item, times: Int) -> [Item] {

    var result = [Item]()

    for i in 0..<times {

        result.append(item)

    }

    return result

}

repeat("knock", 4)

Вы можете создать общие формы функций и методов, так же как и классов, перечислений и структур.

// Переопределяем опциональный тип из стандартной библиотеки Swift

enum OptionalValue<T> {

    case None

    case Some(T)

}

var possibleInteger: OptionalValue<Int> = .None

possibleInteger = .Some(100)

Используйте where после имени типа, чтобы указать список требований — например, потребовать, чтобы тип реализовывал протокол, потребовать чтобы два типа были одинаковы или потребовать, чтобы класс имел определенный суперкласс:

func anyCommonElements <T, U where T: SequenceType, U: SequenceType, T.Generator.Element: Equatable, T.Generator.Element == U.Generator.Element> (lhs: T, rhs: U) -> Bool {

    for lhsItem in lhs {

        for rhsItem in rhs {

            if lhsItem == rhsItem {

                return true

            }

        }

    }

    return false

}

anyCommonElements([1, 2, 3], [3])

В простых случаях, where можно пропустить и просто написать имя класса или протокола после двоеточия. Т.е. написать <T: Equatable> — это то же самое, что и <T where T: Equatable>.

Задание 16

Создайте общий тип и общую функцию.

Для общей функции задайте список требований.