Содержание

Перевод книги The Swift Programming language

Автор перевода: Сергей Югай

Соавторы: Александр Махатма

Перевод книги еще не закончен. Обновления доступны на сайте http://swift-info.ru

По вопросам сотрудничества, помощи в переводе и всем остальном - sergey@hellomrbrown.ru

---

Глава О языке Swift: О языке Swift

О языке Swift

Swift - это новый язык разработки iOS и OS X приложений, который включает в себя все лучшее из C и Objective C, при этом не ограничивая себя совместимостью с C.

Swift применяет безопасные практики программирования и добавляет современные функции, чтобы сделать разработку более простым, гибким и веселым занятием.

Написанный с чистого листа, Swift, опираясь на зрелый и любимый всеми фреймворк Cocoa (и CocoaTouch) - это возможность для разработчиков пересмотреть подход к разработке ПО.

Swift разрабатывался несколько лет. Компания Apple прокладывала фундамент для языка, работая над существующим компилятором, дебаггером и инфраструктурой фреймворков.

Apple упростили управление памятью, используя автоматический подсчет ссылок (ARC). Стэк фреймворков, построенный на мощном основании Foundation и Cocoa, модернизировался и стандартизировался.

Сам Objective-C эволюционировал, добавив поддержку блоков, литералы для коллекций, модули и другие функции, не нарушая существующего кода. Благодаря этой работе, Apple теперь имеет возможность представить новый язык для будущего разработки приложений.

Swift покажется знакомым для разработчиков Objective-C. Он перенимает читаемость именованных параметров из Objective-C и силу его динамической объектной модели. Он предоставляет простой доступ к существующим фреймворкам Cocoa и возможность смешивать код с Objective-C кодом.

Основываясь на этой общей площадке, Swift предоставляет многие новые возможности и унифицирует процедурные и объектно-ориентированные части языка.

Swift дружественен к новым программистам. Это первый язык для разработки систем индустриального качества, который так же выразителен и прост, как скриптовый язык. Он поддерживает "песочницу" - инновационную функцию, позволяющую разработчикам экспериментировать с кодом и сразу видеть результат, без сборки и запуска приложения.

Swift совмещает лучшие идеи из современных языков с мудростью инженерной культуры Apple. Компилятор оптимизирован для производительности и язык оптимизирован для разработки - без компромиссов с той или другой стороны. Он разработан для приложений от "Hello, world" до масштабов целой операционной системы. Все это делает Swift солидным вкладом для разработчиков и для самой Apple.

Swift - это фантастический способ разработки iOS и OS X приложений и продолжит развиваться с новыми функциями и возможностями.

"У нас(Apple) амбициозные цели для Swift. И мы очень ждем ваших приложений, созданных с его помощью."

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Тур по Swift

Тур по Swift

Традиционно, рассказ о новом языке начинается с Hello, world!

В Swift это делается так:

```
println("Hello, world!")
```

Если вы писали код в C или Objective-C, этот синтаксис выглядит знакомым - в Swift, эта строка кода является завершенной программой. Не нужно импортировать библиотеку для работы с вводом/выводом или работой со строками. Код, написанный в глобальной области видимости (global scope), используется как входная точка для приложения, поэтому функция main() не нужна. Также не нужно ставить точки с запятой в конце каждого оператора.

В этом туре вы узнаете достаточно информации для того, чтобы начать писать код в Swift - мы решим ряд типичных задач. Не волнуйтесь, если что-то будет непонятно - все, о чем будет рассказано в этом

туре, далее в книге будет объяснено еще более подробно.

Для лучшего результата, советуем открыть эту главу в песочнице XCode. Она позволит вам редактировать код и сразу видеть результат.

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. Это значит, что вы можете использовать константу для обозначения значения, определяемого единажды, но используемого во многих местах.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

Константа или переменная должна иметь тот е 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 implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

Попробуйте сами создать константу типа Float со значением 4.

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)
```

Попробуйте удалить конвертацию из String в последнией строке. Какую ошибку вы получите?

Есть еще более 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."
```

Попробуйте использовать \() для включения дробных чисел в строку и для вставки чьего-нибудь имени в приветствие.

Массивы и словари создаются с помощью [], как и доступ к их элементу (по ключу или по индексу):

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"
var occupations = [
"Malcolm" : "Captain",
"Kaylee": "Mechanic"
]
occupations["Jayne"] = "Public relations"
```

Чтобы создать пустой массив или словарь, используйте синтаксис инициализации:

```
let emptyArray = String[]()
```

```
let emptyDictionary = Dictionary<String, Float>()
```

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)"
}
```

Измените optionalName на nil. Какое приветствие вы получите? Добавьте else, отображающий другое приветствие для случаев, когда optionalName является nil.

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 монжество 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"
}
```

Попробуйте убрать default случай. Какую ошибку вы получите?

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
```

Добавьте другую переменную для хранения информации о том, какого типа является самое большое число

While используется для повторения блока кода, пока не измениься условие. Можно перенести условие в конеч 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
```

Используйте .. для создания диапазона, *не включающего* верхнее значение (т.е. цикл идет для i = 0, 1, 2) и … для диапазона, включающего оба значения (т.е. for i in 0…3 - это i = 0, 1, 2, 3).

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Функции и замыкания

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")
```

Удалите параметр day. Добавьте параметр для включения сегодняшнего особого блюда в приветствие.

Используйте кортеж (tuple) для возврата нескольких значений из функции:

```
func getGasPrices() -> (Double, Double, Double) {
return (3.59, 3.69, 3.79)
}
getGasPrices()
```

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)
```

Напишите функцию, которая считает среднее значение для ее аргументов

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
    fund 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
})
```

Перепишите замыкание так, чтобы оно возвращало 0 для всех нечетных номеров

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:

```
numbers.map({ number in 3 * number })
```

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:

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Объекты и классы

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."
    }
}
```

Добавьте константное свойство с помощью let; добавьте метод, который принимает аргумент.

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 отличие от Objective-C, где нужно было наследоваться от NSObject - примечание переводчика), в этих случаях имя родителя можно пропустить.

Класс-наследник может определять методы своего родителя, используя слово override - в противном случае, компилятор сообщит об ошибке. Компилятор также определяет методы, которые помечены как override, но на самом деле ничего не переопределяют.

Проиллюстрируем все это с помощью кода:

```
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 \(sidesLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

Создайте другой подкласс NamedShape - например, Circle, который принимает в качестве параметров радиус и имя. Напишите метод area и describe для описания класса Circle

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")
triangle.perimeter
triangle.perimeter = 9.9
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, объявляемых 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")
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50,
name: "larger square")
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
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Перечисления (enumerations) и структуры

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, Eigth, 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.toRaw())
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.toRaw()
```

Напишите функцию, которая сравнивает два значения Rank, используя их "исходные" (toRaw()) значения

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 toRaw и fromRaw для перехода от исходного значения к значению перечисления и обратно.

```
if let convertedRank = Rank.fromRaw(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 "hears"
             case .Diamonds:
                 return "diamonds"
             case .Clubs:
                 return "clubs"
        }
    }
}
let hearts = Suit.Hearts
let heartsDescription = hearts.simpleDescription()
```

Добавьте метод color к Suit, который возвращает "black" для Spades и Clubs и "red" для hearts и diamonds

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()
```

Добавьте к Card метод, который создает полную колоду карт, с одной картой для каждой комбинации номера и рубашки

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)"
}
```

Добавьте третий случай для ServerResponse к switch'у

Обратите внимание, как время заката и рассвета достаются из ServerResponse как пара значений, соответствующих случаям case.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Протоколы и расширения

Протоколы и расширения

Используйте слово protocol для объявления протокола.

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating fund adjust()
}
```

Классы, перечисления и структуры могут соответствовать протоколам:

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 66923
    fund adjust() {
        simpleDescription += " Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription
struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating fund adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

Напишите перечисление, которое будет отвечать этому протоколу

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
    }
}
simpleDescription
```

Напишите расширение для типа Double, которое добавляет свойство absoluteValue

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
// protocolValue.anotherProperty // раскомментируйте и посмотрите на ошибку
```

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.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Тур по Swift: Общие функции и типы

Общие функции и типы

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

```
func repeat<ItemType>(item : ItemType, times: Int) ->
ItemType[] {
    var result = ItemType[]()
    for i in 0..times {
        result += 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: Sequence, U: Sequence,
    T.GeneratorType.Element: Equatable, T.GeneratorType.Element == U.GeneratorType.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])
```

Модифицируйте anyCommonElements, чтобы сделать функцию, возвращающую массив элементов, общих для обоих последовательностей

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>.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Основы

Основы

Swift - новый язык программирования для создания iOS и OS X приложений. Однако, многие части Swift могут быть вам знакомы из опыта разработки на C и Objectve-C.

Swift предоставляет свои собственные версии всех фундаментальных типов C и Objective-C, включая Int для целых, Double и Float для чисел с плавающей запятой, Bool для логических значений и String для текстовых данных. Swift также предоставляет мощные версии двух основных типов коллекций - Array (массив) и Dictionary (словарь), это описано в типах коллекций.

Как C, Swift использует переменные для хранения и ссылки на значения, используя имя. Swift также широко использует переменные, чьи значения не могут быть изменены. Они известны как константы, и они гораздо более мощные, чем константы в Си. Константы используются в Swift для того, чтобы сделать код безопаснее и чище в случаях, когда вы рабоатете со значениями, которые не должны меняться.

В дополнение к известным типам, Swift добавляет новые типы, которых нет в Objective-C. Они включают в себя кортежи (tuple), которые позволяют создавать и передавать группы значений. Кортежи могут возвращать множество значений из функции как одно составное значение.

Swift также добавляет опциональные типы, которые позволяют работать с отсутствием значения. Опциональные типы говорят или "у меня есть значение и оно равно x", или "у меня вообще нет значения". Опциональные значения схожи в использовании с nil и указателями в Objective C, но они работают для любого типа данных, а не только для классов. Опциональные значения безопаснее и выразительнее, чем nil-указатели в Objective-C и часто используются во многих мощных функциях Swift.

Опциональные типы - это пример того факта, что Swift является языком с типовой безопасностью (type safe). Swift помогает вашему коду быть ясным о типах значений, с которыми он работает. Если часть вашего кода ожидает строку, типовая безопасность предотвращает вас от передачи туда целого числа по ошибке. Это позволяет вам ловить и исправлять ошибки как можно раньше в процессе разработки.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Константы и переменные

Константы и переменные

Константы и переменные ассоциируют имя (например, maximumNumberOfLoginAttempts или welcomeMessage) со значением определенного типа (например, число 10 или строка "Hello"). Значение *константы* не может быть изменено после того, как было задано, тогда как переменная может быть изменена на другое значение в будущем.

Объявление констант и переменных

Константы и переменные должны быть объявлены, прежде чем они будут использованы. Константы объявляются с помощью ключевого слова let, переменные - слова var. Вот пример того, как константы и переменные могут быть использованы для отслеживания количества попыток логина, предпринятого пользователем:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Код можно прочитать как "объявите новую константу, названную maximumNumberOfLoginAttempts, и задайте ей значение 10. Затем объявите новую переменую, названную currentLoginAttempt, и дайте ей начальное значение 0."

В этом примере, максимальное количество разрешенных попыток входа объявляется константой, т.к. оно никогда не меняется. Текущее количество попыток объявляется как переменная, т.к. это значение должно увеличиваться после каждой неудачной попытки.

Вы можете объявить несколько констант или несколько переменных на одной строке, разделив их запятой:

```
var x = 0.0, y = 0.0, z = 0.0
```

Если значение, которое вы храните, не будет меняться - всегда объявляйте его как константу, используя слово *let*. Используйте переменные только для хранения значений, которые должны меняться.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Аннотации(обозначения) типов

Аннотации(обозначения) типов

Вы можете добавить аннотацию типа, когда объявляете константу или переменную, чтобы точно обозначить тип значений, которые она может хранить. Аннотация типа отделяется двоеточием от названия переменной или константы. Приведем пример:

```
var welcomeMessage: String
```

Эту строку можно прочитать как "Объявите переменную, названную welcomeMessage, чтобы она хранила текстовые (String) значения". Таким образом, мы показываем, какие именно значения будет содержать переменная (в данном случае - текст/строку).

Переменной welcomeMessage теперь можно присовить любое строковое значение, и это не вызовет ошибок:

```
welcomeMessage = "Hello"
```

На практике, обозначения типов придется писать редко. Если вы предоставите начальное

знаение для переменной или константы, Swift почти всегда сможет сам распознать тип значений, которые она должна содержать. Это будет описано в разделе "Безопасность типов и подбор типов". В примере с welcomeMessage, мы не предоставили начального значения, поэтому мы указываем тип явно.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Имена для констант и переменных

Имена для констант и переменных

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

```
let π = 3.14159
let     = "     "
let 🐮 = "dogcow"
```

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, хотя в целом цифры можно использовать в любой другой части имени.

Как только вы объявили константу или переменную определенного типа, вы не можете переопределить ее снова с тем же именем или поменять тип хранимых данных. Вы так же не можете превратить константу в переменную или наоборот.

Если вам необходимо дать переменной или константе название, которое является ключевым словом в Swift, вы можете сделать это, окружив это слово символами ``, однако этого следует избегать.

Вы можете поменять значение существующей переменной на другое значение того же типа. Например, поменяем friendlyWelcome с "Hello!" на "Bonjour!":

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome теперь содержит "Bonjour!"
```

В отличие от переменной, значение константы нельзя поменять после установки - в противном случае, компилятор выдаст ошибку:

```
let languageName = "Swift"
languageName = "Swift++"
// тут мы получим ошибку, т.к. нельзя менять значение константы
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Вывод значений констант и переменных на экран

Вывод значений констант и переменных на экран

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

```
println(friendlyWelcome)
// получим на экране Bonjour!
```

println - это глобальная функция, которая выводит на экран значение, после чего делает переход на новую строку. Если вы работаете в XCode, к примеру, то 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*

Вы можете напечатать с помощью println любую строку:

```
println("This is a string")
```

Функция println может печатать и более сложные сообщения, как и функция NSLog в Cocoa (знакомая для разработчиков Objective-C). Эти сообщения могут включать значения констант и переменных.

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:

```
println("The current value of friendlyWelcome is \(friendlyWelcome)")
// печатает The current value of friendlyWelcome is Bonjour!
```

Об остальных возможностях строкой интерполяции читайте в разделе "Интерполяция строк" на нашем сайте

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Комментарии

Комментарии

Используйте комментарии, чтобы включить неисполняемый текст в ваш код, в качестве пометки или напоминания для себя. Компилятор игнорирует их при компиляции приложения.

Комментарии в Swift аналогичны комментариям в C. 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 C, 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.

Глава Основы: Точка с запятой

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

```
let cat = "cat"; println(cat);
//печатает cat
```

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Целые числа (тип Integer)

Целые числа (тип Integer)

Целые числа (Integer) - это числа, не содержащие дробной части, например 42 или -23. Они так же могут быть знаковыми (положительными, отрицательными или нулем) или беззнаковыми (положительным или нулем).

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 8битное 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.

Границы целых чисел

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 (UInt8 в нашем случае), соответственно minValue и maxValue автоматом становятся типа UInt8.

Int

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 размеров 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

Используйте UInt только когда вам действительно нужен беззнаковый целый тип размером со слово (word) платформы. В других случаях, предпочтительнее использовать Int, даже если известно, что значения будут неотрицательными. Последовательное использование Int для целых чисел позволяет содержать код совместимым, избегает нужды в конвертации между разными типаци чисел и соответствует определению целого типа, как описано в разделе "Безопасность типов и подбор типов" на нашем сайте.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Числа с плавающей точкой

Числа с плавающей точкой

Числа с плавающей точкой - это числа, у которых есть дробная часть, например 3.14159, 0.1 и -273.15.

Числа с плавающей точкой могут представлять гораздо более широкий диапазон значений, чем целые числа, а также хранить числа, которе гораздо больше или меньше, чем те, что хранятся в Int. В Swift есть два типа чисел с плавающей точкой:

- Double является 64битным числом с плавающей точкой - его следует использовать, когда значения должны быть очень большими или особенно точными
- Float является 32битным числом с плавающей точкой - используйте во всех остальных случаях, когда высокая точность не требуется

Double имеет точность в минимум 15 десятичных знаков, тогда как Float - только 6. Подходящий для использования тип зависит от природы и диапазона значений, с которыми вам требуется работать.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Безопасность типов и подбор(inference) типов

Безопасность типов и подбор(inference) типов

Swift является языком с типовой безопасностью. Типовая безопасность позволяет вам быть точным о типах значений, с которыми работает ваш код. Если кусок кода ожидает строковый тип, вы не сможете по ошибке передать туда число.

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.

Проверка типов позволяет вам избежать ошибок, когда вы работаете с разными типами значений. Однако, это не означает, что вам обязательно объявлять тип каждой константы или переменной - если вы не объявите тип сами, 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 C или Objective-C. Константы и переменные по прежнему явно типизированы, но большинство работы по определению типа сделано за вас.

Подбор типа особенно полезен, когда вы объявляете константу или переменную с начальным значением - это часто делается с помощью присвоения *литерального значения (литерала)* константе или переменной в момент ее объявления. Литерал - это просто значение, которое появляется в вашем коде, например 42 или 3.14159.

Например, если вы присваиваете литерал 42 новой константе без объявления ее типа, Swift автоматически догадывается, что тип вашей константы - Int, поскольку вы инициализируете ее числом, похожим на целое число:

```
let meaningOfLife = 42
// meaningOfLife автоматически становится Int
```

Точно так же, если вы не указываете тип для литерала с плавающей точкой, Swift предположит, что вы хотите использовать тип Double:

```
let pi = 3.14159
// pi имеет тип Double
```

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 3 не имеет заданного типа, зато компилятор видит присутствие плавающей точки - поэтому и назначает тип Double.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Числовые литералы

Числовые литералы

Целоисленные 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 10^{exp}:

- 1.25e2 означает 1.25 X 10^{2}, или 125.0
- 1.25e-2 означает 1.25 X 10^{-2}, или 0.0125

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

- 0xFp2 означает 15 X 2^{2}, или 60.0
- 0xFp-2 означает 15 X 2^{-2}, или 3.72

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
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Конвертация числовых типов

Конвертация числовых типов

Используйте тип Int для всех своих целочисленных констант и переменных, даже если известно, что они будут неотрицательными (разумеется, за исключением особых случаев). Использование стандартного целого типа в повседневных ситуациях означает, что целочисленные константы и переменные сразу могут взаимодействовать с автоподставляемым типом для целочисленных литеральных значений. (от переводчика - вообще, здесь говорится о том, что переменные становятся interoperable - в данном случае, имеется ввиду, что все наши константы и переменные, если они будут иметь тип Int, общепринятый в системе - смогут безопасно работать со всеми другими константами и переменными, зная, что они тоже Int).

Используйте другие целочисленные типы только в тех ситуациях, когда они действительно нужны для конкретной задачи, например когда используются данные с заданным размером из внешнего источника, или для производительности, уменьшения использования памяти или других оптимизаций. Использование типов с явно заданным размером в таких ситуациях позволяет поймат любые случайные переполнения данных и явно документирует природу используемых данных/значений.

Конвертация целых чисел

Диапазон чисел, которые можно хранить в целочисленной константе или переменной, отличается для

каждого типа - например, в Int8 можно хранить числа от -128 до 127, тогда как в UInt8 - числа от 0 до 255. Число, которое не помещается в диапазон приемлемых чисел, вызовет ошибку на этапе компиляции, например:

```
let cannotBeNegative: UInt8 = -1
// UInt8 не может хранить отрицательных значений, тут будет ошибка
let tooBig: Int8 = Int8.max + 1
// Int8 не может хранить число большее, чем максимальное значение, тут тоже ошибка
```

Поскольку каждый числовой тип может хранить разные диапазоны значений, вам в определенных случаях придется обращаться к явной конвертации числовых типов. Этот подход предотвращает от скрытых ошибок конвертации и позволяет ей быть явной в вашем коде.

Чтобы сконвертировать один числовой тип в другой, вам необходимо инициализировать новое число желаемого типа с использованием существующего значение. В примере ниже, константа twoThousand имеет тип UInt16, тогда как константа one имеет тип UInt8. Их нельзя напрямую добавлять друг к другу, поскольку они не одного типа. Вместо этого, мы вызываем UInt16(one), чтобы создать новое значение типа UInt16, используя значение константы one:

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Поскольку оба складываемых значения в последней строчке кода имеют тип UInt16, компилятор позволяет выполнить сложение - и выходная константа twoThousandAndOne будет иметь тип UInt16 - поскольку компилятор видит, что ей присваивается значение суммы двух UInt16-констант.

SomeType(ofInitialValue) (т.е. выражение вида НазваниеТипа(начальноеЗначение) - прим. переводчика) - это стандартный способ вызвать инциализатор типа в Swift и передать начальное значение. У типа UInt16 есть инициализатор, принимающий значение UInt8, и этот инициализатор используется, чтобы создать новый UInt16 из существующего UInt8. Однако, *любой* тип передать не получится - только тот тип, для которого UInt16 имеет соответствующий инициализатор. Впрочем, можно использовать расширения для создания своих инициализаторов - об этом будет рассказано в главе "Расширения" на нашем сайте.

Конвертация целых чисел и чисел с плавающей точкой

Конвертация между целыми числами и числами с плавающей запятой должна быть явной:

```
let three = 3
let pointOneFourOneFiveNine = 0.14519
let pi = Double(three) + pointOneFourOneFiveNine
// pi равно 3.14519 и он имеет автоматически определенный тип Double
```

Здесь, константа 3 используется для создания нового значения Double, таким образом в последней строке оба слагаемых имеют один тип. Без этой конвертации, сложение было бы невозможно (компилятор выдал бы ошибку).

Обратная конвертация также возможна, т.е. целочисленное значение можно инициализировать значением с плавающей точкой:

```
let integerPi = Int(pi)
// integerPi равен 3, имеет тип Int
```

Для дробных значений, при конвертации в целое, просто отбрасывается дробная часть - т.е. 4.75 станет 4, -3.9 станет -3.

Правила для совмещения числовых констант и переменных отличаются от правил для числовый литералов. Литеральное значение 3 может быть добавлено напрямую к литеральному значению 0.14159, т.к. числовые литералы не имеют определенного типа - их тип определяется лишь когда они оцениваются компилятором.

Глава Основы: Алиасы типов

Алиасы типов

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

Они очень полезны, когда вы хотите ссылаться на определенный тип названием, более подходящим в данном контексте. К примеру, вы работаете с данными определенного размера из внешнего источника:

```
typealias AudioSample = UInt16
```

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

```
var maxAmplitudeFound = AudioSample.min
//maxAmplitudeFound теперь 0
```

В данном примере, мы объявили AudioSample как алиас для UInt16 и, поскольку это алиас, мы можем обращаться к AudioSample.min, который на самом деле вызывает UInt16.min, предоставляющий, в свою очередь, значение для нашей переменной.

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Логический тип (boolean)

Логический тип (boolean)

Swift имеет базовый логический тип, называемый Bool. Логический тип (он же - булевый тип) называется так, поскольку он может содержать только два зачения - true(правда) или false(ложь). 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.

Логические значения особенно полезны при работе с условными выражениями, например if:

```
if turnipsAreDelicious {
 println("ммм, вкусная репа!")
} else {
 println("фуу, репа ужасна!")
}
// напечатает "репа ужасна" (а лично я люблю репу! - прим. переводчика)
```

Условные выражения, такие как if, будут рассмотрены в главе "Управление контролем".

Типовая безопасность языка предотвращает использование не-логических значений в выражениях, требующих логическое значение. Следующий пример (нормально компилирующийся в C 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".

Как и в других примерах типовой безопасности Swift'а, этот подход избегает случайных ошибок и позволяет убедиться, что намерения каждой части кода всегда ясны.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Кортежи (tuples)

Кортежи (tuples)

Кортеди (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. Код ответа HTTP - это специальное значение, которое веб-сервер возвращает вам каждый раз, когда вы запрашиваете веб-страницу. Код 404 Not found значит, что запрашиваемая страница не найдена.

```
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"
```

Кортежи особенно полезны, когда мы возвращаем значение из функций. ФФункция, которая скачивает веб страницу, может возвращать кортеж (Int, String) для обозначение успешности или неудачи совершенной операции. Возвращая кортеж с двумя значениями, каждое разного типа, функция предоставляет больше полезной информации о ее выполнении, чем если бы она возвращала какое-то одно значение. Для большей информации по теме, смотрите раздел "Функции с несколькими возвращемыми значениями" на нашем сайте.

Кортежи полезны для временных групп схожих значений. Они не предназначены для создания сложных структур данных. Если есть шанс, что ваши данные будут храниться в приложении некоторое время, лучше смоделировать их с помощью класса или структуры. Для большей информации на эту тему, почитайте раздел "Классы и структуры".

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Опциональные значения

Опциональные значения

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

- или "у меня есть значение и оно равно x"
- или " у меня нет никакого значения"

Концепции опциональных значений не существует в C или Objective-C. Самое ближайшее в Objective-C - это возможность возвращать nil из метода, который возвращает объекты. nil как бы обозначает, что "корректный объект отсутствует", однако это работает только объектов, но не для структур, базовых типов или перечислений. Для этих типов, Objective-C методы обычно возвращают какое-нибудь специальное значение (например, NSNotFound), чтобы обозначить отсутствие значения. Этот подход предполагает, что код, вызывающий метод, знает о том, что есть это специальное значение и что его нужно учитывать. ОПциональные значения в Swift позволяют вам указать возможность отсутствия значения для *любого типа*, без надобности использования дополнительных констант.

Приведем пример. Тип String имеет метод toInt, который пробует перевести содержимое строки в целочисленное значение типа Int. Однако, не каждая строка может быть сконвертирована вв 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).

Условные выражения и вынужденная распаковка (forced unwrapping)

Вы можете использовать if, чтобы выяснить, содержит ли опциональное значение какое-либо значение. Если оно содержит, то оно вернет true, иначе - false.

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 {
    println("\(possibleNumber) имеет целочисленное значение \(convertedNumber!)")
} else {
    println("\(possibleNumber) не может быть сконвертирован в целое число")
}
// печатает на экран "123 имеет целочисленное значение 123"
```

Для большей информации об условных конструкциях, обратитесь к главе "Контроль управления" на нашем сайте.

Попытка использовать ! для того, чтобы достать значение в случаях, когда его нет в опциональном значении, вызовет ошибку времени исполнения (runtime error), т.е. ошибку во время работы приложения. Поэтому всегда убеждайтесь, что значение содержится, прежде чем использовать восклицательный знак.

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. if и while описываются в разделе "Контроль управления" на нашем сайте.

Опциональные связки реализуются следующим образом:

```
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.

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
```

nil не может быть использован с не-опциональными константами и переменными. Если константа или переменная в вашем коде должна работать с отсутствием значений при определенных обстоятельствах, всегда определяйте ее как опциональную с подходящим типом.

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

```
var surveyAnswer: String?
// surveyAnswer автоматически является nil
```

В Swift nil - не то же самое, что nil в Objective C. Там nil является указателем на несуществующий объект, в Свифте же - это не указатель, а отсутствие значения определенного типа. Опциональные переменные/константы любого типа могут быть установлены в nil, а не только объекты.

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 слчаях, 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. Основное использование таких опциональных переменных в Swift - при инициализации класса, как описывается в разделе "Ссылки без владельцев и неявно развернутые опциональные свойства" (позже вставим сюда ссылку, когда переведем).

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 разрешения 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 {
    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.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Основы: Утверждения (они же ассерты, они же assertions)

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 дасть 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.

Если ваш код вызывает утверждение в отладочном окружении, т.е., например, при запуске в XCode, то вы сможете увидеть конкретное место и состояние, вызвавшее падение приложения. Утверждение так же позволяет вам предоставить подходящее отладочное сообщение, отражающее натуру утверждения.

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 функциии - 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.

Когда стоит использовать утверждения

Используйте их, когда условие потенциально может быть false, но должно обязательно быть true, чтобы код продолжил выполнение нормально. Вот пара примеров:

- Индекс для обращения к элементу коллекции должен быть в пределах коллекции, но может быть выше или ниже пределов
- Значение, передаваемое функции, некорректно и не позволяет нормально ее выполнить
- Опциональная переменная содержит nil, но должна быть не-nil для успешного выполнения кода.

Для более подробной информации смотрите также "Доступ к элементам коллекции" и "Функции" (как только мы их переведем..)

Утверждения завершают приложение, поэтому они являются эффективным способом, чтобы подсказать, что все необходимые условия всегда соблюдаются в течение процесса разработки и отловить нужные ошибки, но перед публикацией приложения от них лучше избавиться (поскольку не очень правильно просто брать и вырубать приложение, не сообщив ничего пользователю).

p.s. последнее предложение является вольной трактовкой того, что указано в туториале, но смысл отражает точно - впрочем, буду рад услышать о любых корректировках, пишите на email, указанный ниже

Глава Базовые операторы: Базовые операторы

Базовые операторы

Оператор - это специальный символ или фраза, которые Вы используете для проверки, изменения или комбинирования значений. Например, оператор сложения (+) складывает два числа вместе (как let i = 1 +1). Более сложные примеры включают логический оператор и && (как в if enteredDoorCode && passedRetinScan) и оператор инкремента ++i, который является упрощенным оператором (shortcut) увеличения значения i на 1.

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, которые описаны в 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.

Эта глава описывает самые частые операторы Swift. Продвинутые Операторы охватывает продвинутые операторы Swift и описывает как определить Ваши собственные операторы и написать стандартные операторы для Ваших типов.

Терминология

Операторы бывают унарные, бинарные и тернарные:

- Унарные оперируют одним значением (как например -a). Унарные префиксные операторы определяются сразу перед выражением (как !b) и унарные постфиксные операторы сразу после (i++)
- Бинарные оперируют двумя выражениями (2 + 3) и только
- Тернарные операторы используют три выражения. Как и в C, в Swift есть только один тернанрный оператор, оператор состояния (a ? b : c)

Значения, с которыми операторы взаимодействуют, называются *операндами* .
В выражении 1 + 2 , + это бинарный оператор, а 1 и 2 - это операнды.

Оператор Присваивания

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

```
let b = 10
var a = 5
a = b

//a 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 допустимо поскольку 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 в Вашем коде.

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Арифметические операторы

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) , которые описаны в Операторы Переполнения

Оператор сложения также поддерживает конкатенацию строк :

```
"hello," + "world" //равно "hello,world"
```

Два символьных(character) значения или одно символьное значение и одна строка(String) могут быть сложены вместе и создают новую строку :

```
let A: Character = "A"
let B: Character = "B"

let AB = A + B
//AB сейчас равно строке AB
```

Смотрите также Конкатенацию Строк и Символов

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Оператор Остатка

Оператор Остатка

Оператор Остатка (a % b) вычисляет сколько останется от деления a на b и возвращает это значение

Заметьте:
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 .

Вот как этот оператор работает. Чтобы вычислить 9 % 4, сначала посчитайте сколько 4ок поместится в 9ти:

```
    4       4 1
1 2 3 4 5 6 7 8 9
```

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

В Swift это может быть записано как:

```
9 % 4 //равно 1
```

Чтобы определить ответ для a % b , оператор % считает следующее уравнение и возвращает остаток как значение
a = (b x какой-то множитель) + остаток
Где "какой-то множитель" это самое большое количество множителей b, которое поместится в a.
Подставив 9 и 4 в это уравнение, получим:
9 = (4 x 2) + 1

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

```
-9 % 4 // равно - 1
```

Подставив -9 и 4 в это уравнение, получим:
-9 = (4 x 2) + -1
возвращаемый остаток = - 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

```
2.5  2.5  2.5 .5
1 2 3 4 5 6 7 8
```

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Операторы Инкремента и Декремента

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 сейчас равно 1
```

Каждый раз когда Вы вызываете ++i , значение i увеличивается на 1. По существу, ++i это сокращенный способ сказать i = i + 1. Также, --i используется для сокращения i = i - 1.
Символы ++ и -- могут быть использованы как префикс и постфикс. ++i и i++ - оба выражения допустимые способы увеличить значение i на 1. Точно как, --i и i-- -допустимые способы уменьшить значение i на 1.

Заметьте что эти операторы изменяют i и также возвращают значение. Если вы хотите только уменьшить или увеличить значение, хранимое в i, достаточно игнорировать возвращаемое значение. И если вы используете возвращаемое значение, оно будет различным, в зависимости от того, используете вы префиксную или постфиксную версию оператора, в соответствии со следующими правилами :

- Если оператор написан перед переменной, он инкрементирует переменную *перед* тем как вернуть её значение.
- Если оператор написан после переменной, он инкрементирует переменную *после* того как вернет значение.

```
var a = 0
let b = ++a
// a и b сейчас равны 1
let c = a++
// a рано 2 , c = 1
```

В примере выше, let b= ++a увеличивает a перед возвращением значения. Это почему a и b оба равны 1. Также , let c = a++ увеличивает a после возвращения значения . Т.е. c получает старое значение 1, инкрементится a и становится равным 2.
За исключением, когда вам нужно специфичное поведение i++, рекомендуется использовать ++i и --i во всех случай, потому что у них типичное поведение изменения и возвращения значения.

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Унарный Оператор Минус

Унарный Оператор Минус

Знак численных значений может быть переключен, используя префикс -, известный как унарный оператор минус:

```
let three = 3  // равно 3

let minusThree = -three  // равно -3

let plusThree = -minusThree // равно 3
```

Унарный оператор минус (-) добавляется сразу перед значением, которым он оперирует, без пробелов.

Унарный Оператор Плюс

Унарный оператор плюс (+) возвращает значение перед которым стоит, ничего не изменяя:

```
let minusSix = -6  // равно 6

let alsoMinusSix = +minusSix // тоже равно 6
```

Хотя оператор плюс ничего по-существу не делает, Вы можете использовать его чтобы обеспечить симметрию в Вашем коде для положительных чисел, когда используете оператор минус для отрицательных чисел.

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Составные операторы присваивания

Составные операторы присваивания

Как C, Swift предоставляет составные операторы присваивания, которые комбинируют операторы присваивания с другими операторами. К примеру, присваивание с суммированием(+=):

```
var a = 1

a += 2 //a сейчас равно 3
```

Выражение a += 2 это упрощение для a = a + 2. Фактически, сложение и присваивание скомбинированы в один оператор, который выполняет обе задачи сразу.

Заметьте
Составное присваивание не возвращает значение . Вы не можете написать let b = a += 2 , например. Это поведение отличается от операторов инкремента и декремента.

Полный список составных операторов приваивания можно найти в Выражения .

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Операторы сравнения

Операторы сравнения

Swift поддерживает все стандартные операторы сравнения C :

- Равно (a == b)
- Не равно (a !=b)
- Больше чем (a > b)
- Меньше чем (a < b)
- Больше чем или равно (a >= b)
- Меньше чем или равно (a <= b)

Заметьте
Swift также предоставляет два оператора равенства (=== и !==), которые Вы используете для проверки являются ли два объекта-ссылки ссылающимися на тот же экземпляр объекта. Больше информации в Классах и Структурах .

Каждый из операторов сравнения возвращает Булевое значение (Bool), которые показывает является ли выражение верным(true):

```
1 == 1  //true(верно) поскольку 1 равен 1
2 != 1  //true, поскольку 2 не равно 1
2 > 1   //true, поскольку 2 больше чем 1
1 < 2   //true, поскольку 1 меньше 2
1 >= 1  //true, поскольку 1 больше чем либо равно 1
2 <= 1  //false( не верно), поскольку 2 не меньше и не равно 1
```

Операторы сравнения часто используются в условных выражениях, таких как if :

```
let name = "world"

if name == "world"
{
    println("hello, world")
}
else
{
    println("Im sorry \(name),but I don't recognize you")
}

// Напечатается "hello, world", поскольку name == "world"
```

Для большего о if выражениях смотрите Control Flow .

Тернарный условный оператор

Тернарный условный оператор это специальный оператор с тремя частями, которые принимают форму question ? answer1 : answer2. Это упрощенный вариант для оценки одного из двух выражений, основанный на тот верно или нет выражение question . Если question верно (true), выполняется answer1 и возвращает значение; если не верно, то считается и возвращается значение answer2.
Тернарный условный оператор это упрощение для следующего кода:

```
if question
{
    answer1
}
else
{
    answer2
}
```

Здесь пример, который подсчитывает пиксели высоты для ячейки таблицы(table row). Высота должна быть выше на 50 пикселей, чем высота контента, если у ячейки есть заголовок, и на 20 пикселей выше,

если у ячейки нет заголовка:

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
//rowHeight равно 90
//прим.переводчика: если Вы поставите ? сразу после hasHeader,будет ошибка
```

Предшествующий пример это сокращенный вариант кода :

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader
{
    rowHeight = rowHeight + 50
}
else
{
    rowHeight = rowHeight + 20
}
```

Первый пример использования тернарного условного оператора значит что rowHeight может установлен в нужное значение в единственной строке кода. Это более кратко, чем второй пример и позволяет rowHeight быть не переменной, а константой, поскольку её значение не будет изменено внутри if.
Тернарный условный оператор предоставляет эффективный способ решения какое из двух выражений принять. Используйте тернарный оператор с осторожностью. Его сжатость может привести к трудночитаемости кода. Избегайте комбинирование нескольких экземпляров тернарного оператора в одном сравнении.

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Операторы Диапазона

Операторы Диапазона

Swift включает два оператора диапазона, которые упрощают обозначение диапазона значений

Оператор Закрытого Диапазона

Оператор закрытыго 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)")
    index
}

// 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
```

Для большей информации о for-in циклов смотрите Control Flow

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 диапазаон 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 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

Заметьте, что массив содержит четыре значения, но 0..count считает только до 3(индекс последнего элемента в массиве), поскольку используется полузакрытый диапазон.
Для большей информации о массивах, смотрите Массивы

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Базовые операторы: Логические Операторы

Логические Операторы

Логические Операторы изменяют или комбинируют Булевскиие логические значения true и false. Swift поддерживает три стандартных логических оператора из C-основанных языков:

- Логическое отрицание (!a)
- Логическое и (a && b)
- Логическое или (a || b)

Логическое Отрицание

Оператор логического отрицания (!a) инвертирует Булевское значение, т.е. true становится false, и false становится true.
Этот оператор является префиксным, и появляется сразу перед значение над которым он работает, без пробелов. Это можно прочесть как "не a". Например:

```
let allowedEntry = false
if !allowedEntry
{
    println("Access Denied")
}
```

```
// напечатается "Access Denied"
```

Фраза if !allowedEntry может звучать как "не allowedEntry". Следующая строка выполнится только, если "не allowedEntry" равен true; это так, поскольку allowedEntry = false.
Как и в этом примере внимательный выбор Булевской константы и имени переменной, могут помочь сохранить коду читаемость и краткость, пока избегается двойное отрицание или запутанные логические выражения.

Логический Оператор И

Логический оператор И (a && b) создает логические выражения, где оба выражения должны быть true, чтобы общее выражение тоже было true.
Если любое значение false, то общее выражение тоже будет false. По факту, если первое значение false, второе не будет даже вычисляться, поскольку невозможно чтобы общее выражение стало true . Это известно как *быстрая схема вычисления* .

В этом примере сравниваются две Булевые переменные и доступ разрешен, только если оба значения true:

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan
{
    println("Welcome!")
}
else
{
    println("Access Denied")
}

//напечатается "Access Denied"
```

Логический оператор ИЛИ

Логический оператор или (a || b) это оператор, который используется для создания выражений, в которых общее значение будет true, если хотя бы одно из значений true.
Как и логический оператор и, логический или использует *быструю схему вычисления.*
Если левая сторона выражения true, правая не вычисляется, поскольку правая уже не может изменить общего значения.
В примере ниже, первая Булевая переменная (hasDoorkey) равна false, однако второе значение (knowsOverridePassword) равна true. Поскольку одно значение true, общее выражение также считается true, и доступ разрешен:

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword
{
    println("Welcome")
}
else
{
    println("Access Denied")
}


//напечатается "Welcome!"
```

Комбинирование Логических Операторов

Вы можете комбинировать несколько логических операторов, чтобы увеличить количество сравнений:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword
{
    println ("Welcome")
}
else
{
    println("Access Denied")

}
```

Этот пример использует несколько && и || операторов чтобы создать длинное смешанное сравнение. Однако, && и || всё еще работает с двумя операндами, т.е. это три маленьких выражения связанных вместе. Это можно прочесть как :
Если enteredDoorCode и passedRetinaScan; или если hasDoorKey ; или если knowsOverridePassword, то выполнить println("Welcome"), иначе println("Access Denied").
Основываясь на enteredDoorCode, passedRetinaScan, and hasDoorKey, первые два выражения false. Однако, knowsOverridePassword true, а значит и всё выражение true.

Явные скобки

Иногда полезно использовать скобки, когда они не необходимы, чтобы сделать общее выражение более читаемым. В верхнем выражении полезно добавить скобки в первую часть общего выражения, чтобы явно выделить выражение:

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey || knowsOverridePassword
{
    println ("Welcome")
}
else
{
    println("Access Denied")

}
```

Скобки делают понятнее, что первые два значения являются частью общего разделенного выражения. Вывод объединного выражения не изменяется, однако общий замысел более понятен читающему. Читаемости всегда отдается предпочтение перед краткостью; используйте скобки там, где они помогают сделать идею более ясной.

Автор перевода данной главы - Александр Махатма, maxatma@mail.ua, оригинал

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Строки и символы

Строки и символы

Внимание! Перевод данной главы еще не завершен!

Строка - это упорядоченная коллекция (набор) символов, например "hello, world" или "альбатрос". Строки в Swift представлены типом String, который, в свою очередь, представляет собой коллекцию значений типа Character.

Типы String и Character в Swift предоставляют быстрый, совместимый с юникодом способ работать с текстом в вашем коде. Синтаксис для создания и манипуляций со строками легкий и читабельный, со

схожим синтаксисом для C-строк. Для конкатенации (сложения двух строк вместе) достаточно просто сложить их, используя оператор +, а изменяемость строк (возможность модифицировать существующую строку) определяется тем, хранится ли она в константе или в переменной - как и для любого другого типа в Swift.

Несмотря на простоту синтаксиса, String в языке имеет быструю и современную реализацию - каждая строка состоит из независимых от кодировки юникод-символов и предоставляет поддержку для доступа к этим символам в разных юникод-представлениях (unicode representations).

Строки также могут быьт использованы для вставки констант, переменных, литералов и выражений в более длинные строки - это называется "строковая интерполяция". Это позволяет легко создавать собственные строковые значения для вывода, хранения и печати.

String в Swift'e "внутри" бесшовно сшит (bridged) с классом NSString. Если вы работаете с Foundation в Cocoa или Cocoa Touch, весь API NSString доступен вам для каждой создаваемой вами String в свифте - и это помимо всех функций String, которые описываются в этой главе. Вы также можете использовать String для любых API, в которых используется NSString. Для более подробной информации об использовании String с Foundation и Cocoa, обратитесь к "Используем Swift с Cocoa и Objective-C".

Строковые литералы

Вы можете включить предопределенные строковые значения в вашем коде как строковые литералы. Строковый литерал - это какая-либо зафиксированная последовательность текстовых символов, окруженная парой кавычек.

Строковый литерал может быть использован в качестве начального значения для константы или переменной:

```
let someString = "Some string literal value"
```

Обратите внимание, что Swift сам автоматически определит тип String для someString, поскольку мы инициализировали ее с помощью строкового литерала.

Строковые литаралы могут включать следующие специальные символы:

- Специальные символы \0 (null), \\ (бэкслэш), \t (горизонтальная табуляция), \n (перенос строки), \r (возврат каретки), \" (двойная кавычка), \' (одиночная кавычка).
- Однобайтовые юникод-скаляры, записанные как \xNN, где NN - две шестнадцатеричных цифры.
- Двухбайтные юникод-скаляры, записанные как \uNNNN, где NNNN - четыре шестнадцатеричных цифры.
- Четыребайтные юникод-скаляры, записанные как \UNNNNNNNN, где NNNNNNNN - восемь шестнадцатеричных цифр.

Код ниже показывает пример кажого специального символа. Константа wiseWords содержит цитату в кавычках. doollarSign, blackHeart и sparklingHeart показывают использование разных юникод-скаляров. Примечание переводчика: чтобы посмотреть на скаляры, лучше запустить эту радость у себя в XCode. Я попозже выдерну сюда картинку, но пока я этого не сделал - лучше посмотреть в xcode :)

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\x24"         // $,  Unicode scalar U+0024
let blackHeart = "\u2665"       // ♥,  Unicode scalar U+2665
let sparklingHeart = "\U0001F496"  //  , Unicode scalar U+1F496
```

Глава Строки и символы: Инициализация пустой строки

Инициализация пустой строки

Чтобы создать пустое значение String как стартовую точку для создания более длинной строки - либо присвойте пустой строковый литерал переменной, либо создайте новый объект String с помощью синтаксиса инициализации:

```
var emptyString = "" // пустой строковый литерал
var anotherEmptyString = String() // синтаксис инициализации
// обе строки пусты и эквивалентны друг другу
```

Можно узнать, пуста ли строка, обращаясь к свойству isEmpty:

```
if emptyString.isEmpty {
    println("Nothing to see here")
}
// выведет Nothing to see here
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Изменение строк

Изменение строк

Вы можете решить, будет ваша строка модифицируемой или нет, присвоив ее переменной или константе (соответственно, модифицируется если переменная и нет, если константа):

```
var variableString = "Horse"
variableString += " and carriage"
// variableString теперь "Horse and carriage"

let constantString = "Highlander"
constantString += " and another Highlander"
// это вызовет ошибку компилятора, поскольку нельзя модифицировать константу
```

Этот подход отличается от модифицируемости строк в objective-c, где вам нужно было выбирать между двумя классами - NSString и NSMutableString - чтобы определить, будет ли строка модифицируемой.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Строки это тип-значение

Строки это тип-значение

В Swift тип String - это тип-значение (value type - не заморачивайтесь на тему самого термина, главное - это вывод, который сейчас из этого последует - прим. переводчика). Если вы создаете новую строку

String, ее значение *копируется*, когда оно передается в функцию или метод или когда присваивается константе или переменной. В любом случае, *копия* существующей строки создается и затем передается или присваивается.. а не ее оригинал. Типы-значения описаны в разделе "Структуры и перечисления - это типы-значения".

Это поведение отличается от NSString в Cocoa - там, при создании NSString-объекта и передаче его в функцию или при присвоении переменной, вы всегда передавали или присваивали *ссылку* на один и тот же объект NSString. Копирования не происходило, если только вы специально это не указывали.

Подход Swift "копируем по умолчанию" позволяет быть уверенным, что если функция или метод передали вам значение String, то это - ваше значение и что оно не будет модифицировано где-либо еще.

"Глубоко внутри" (при компиляции), компилятор языка оптимизирует использование строк так, чтобы само копирование происходило только когда оно абсолютно необходимо, что означает хорошую производительность при работе со строками как со значениями.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Работа с символами

Работа с символами

String представляет собой коллекцию значений Character с обозначенным порядок. Каждое значение Character представляет собой один юникод-символ. Вы можете получить доступ к каждому символу Character, например, используя итерацию по всей строке с помощью for-in:

```
for character in "Dawg!" {
    println(character)
}
// D
// a
// w
// g
// !
```

for-in описывается в разделе "Циклы for".

Как вариает, вы можете создать отдельный символ Character как константу или переменную, используя односимвольный строчный литерал:

```
let yenSign: Character = "¥"
```

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Подсчет символов

Подсчет символов

Чтобы получить количество символов в строке, вызовите глобальную функцию countElements и передайте ей строку в качестве параметра:

```
let unusualMenagerie = "Koala  , Snail  , Penguin  , Dromedary  "
println("unusualMenagerie имеет \(countElements(unusualMenagerie)) символов")
// печатает "unusualMenagerie имеет 40 символов"
```

Различные символы юникода и различные представления одного и того же символа юникода могут занимать различное количество памяти. Из-за этого, символы в Swift не занимают одинаковое количество памяти в строке и, в результате, длина строки не может быть подсчитана без итерации по всей строке и подсчета каждого символа. Если вы работаете с особенно длинными строками, помните, что countElements должна пройти по всем символам в строке, чтобы точно подсчитать количество символов в ней.

Помните также, что количество символов, возвращаемое функцией countElements, не всегда совпадает со свойством length объекта NSString, содержащего эти символы. Длина NSString базируется на количестве 16-битных еденицы кода внутри UTF-16 представления строки, а не на количестве юникод-символов в строке. Чтобы отобразить этот факт, свойство length из NSString вызывается с помощью utf16count объекта String

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Конкатенация строк и символов

Конкатенация строк и символов

String и Character могут быть сложены вместе (конкатенированы) с помощью знака +, чтобы создать новое значение String:

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"

let stringPlusCharacter = string1 + character1        // "hello!"
let stringPlusString = string1 + string2              // "hello there"
let characterPlusString = character1 + string1        // "!hello"
let characterPlusCharacter = character1 + character2  // "!?
```

Вы можете также добавить String или Character к существующей переменной String с помощью оператора +=:

```
var instruction = "look over"
instruction += string2
// instruction теперь "look over there"

var welcome = "good morning"
welcome += character1
// welcome теперь "good morning!
```

Вы не можете добавить String или Character к существующей переменной Character, т.к. она должна содержать только один символ.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Строки и символы: Интерполяция строк

Интерполяция строк

Интерполяция строк - это способ создать новое значение String из смеси констант, переменных, литералов и значений, включая их все в один строчный литерал. Каждый элемент, который вы включаете, заключается в \():

```
let multiplier = 3
let message = "\(multiplier) умножить на 2.5 равно \(Double(multiplier) * 2.5)"
// message теперь "3 умножить на 2.5 равно 7.5"
```

В примере выше, значение multiplier вставляется в строковый литерал как \(multiplier). Оно заменяется на значение самой переменной, когда выполняется строковая интерполяция для создания самой строки.

Значение multiplier также используется в выражении далее в строке - выражение вычисляет значение Double(multiplier) * 2.5 и вставляет результат (7.6) в строку. В данном случае оно тоже записывается внутри \(), т.е. \(Double(multiplier) * 2.5), когда вставляется в строковый литерал.

Выражения, которые вы пишите внутри скобок в интерполированной строке, не могут содержать кавычки ", бэкслэш \ и не могут содержать перенос строки.

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Типы коллекций:

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Поток управления:

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Замыкания (closures):

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Перечисления (enumerations):

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Классы и структуры:

---

http://swift-info.ru, автор перевода: Сергей Югай

Глава Свойства:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Методы:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Доступ к элементам коллекций (subscripts):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Наследование:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Инициализация:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Деинициализация:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Автоматический подсчет ссылок - ARC:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Опциональное связывание (chaining):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Приведение типов (type casting):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Вложенные типы (nested types):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Расширения (extensions):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Протоколы:

http://swift-info.ru, автор перевода: Сергей Югай

Глава Общие (шаблоны, generics):

http://swift-info.ru, автор перевода: Сергей Югай

Глава Продвинутые операторы:

http://swift-info.ru, автор перевода: Сергей Югай

Автор перевода - Сергей Югай

Контактный email - sergey@hellomrbrown.ru

Язык Swift, все права на него, ровно как и на айфон, айпод, айпад, objective c и остальные вещи принадлежат компании Apple. Данный сайт лишь содержит перевод информации о языке Swift.